The SP71 computer model

J Gerard Wolff*

January 1, 2015

Abstract

This is the source code and associated notes for the SP71 computer model, a successor to the SP70 model described in *Unifying Computing and Cognition*.

1 Introduction

This is mainly the source code and associated notes for the SP71 computer model, a successor to the SP70 model described in *Unifying Computing and Cognition* [1, Sections 3.9.2 and 9.2].

Notes for the development are in Section 2.6.

2 Source code

2.1 SP71.cpp

```
// SP71: a program for unsupervised learning in the SP framework.

// Copyright (C) 2005 J G Wolff.

// This program is released to the public domain without any restrictions.
// You can redistribute it and/or modify it as you wish.

// In any publication that refers to any version of this program, please
// acknowledge J G Wolff as the originator of the program.

// A description of SP70, the precursor of this program, may be found
// in "Unsupervised learning in a framework of information compression
// by multiple alignment, unification and search", J G Wolff, March 2002.
// A copy may be obtained from http://uk.arxiv.org/abs/cs.AI/0302015.
// See also http://www.cognitionresearch.org.uk/papers/ul/ul.htm.

// Please retain this notice on all versions.

// Dr J G Wolff, www.CognitionResearch.org.uk.
```

*Dr Gerry Wolff, BA (Cantab), PhD (Wales), CEng, MBCS (CITP); CognitionResearch.org, Menai Bridge, UK; jgw@cognitionresearch.org; +44 (0) 1248 712962; +44 (0) 7746 290775; *Skype*: gerry.wolff; *Web*: www.cognitionresearch.org.

```
// Email: jgw@cognitionresearch.org.uk.

// April 2005.

// SP71.cpp : Defines the entry point for the console application.
//

#include "SP71_head.h"

int main(int argc, char* argv[])
{
        // Make sure the program has been invoked with
        // the correct arguments.

        // NOTE: If this program is developed in some environment other than
        // MS Visual C++, then 'SP_main()' should probably become 'main()' and the
        // following lines of code (up to 'SP_main()') should be moved to
        // the beginning of main().

        if (argc < 4)
        {
                fprintf(stderr, "Usage:\n\t%s<in-file><parameters-file><out-file> \
                        <latex-file><plot-file>\n") ;
                return 1;
    }

        in_filename = *++argv ; // Name of file containing element
                          // syntax and descriptions.
        parameters_filename = *++argv ; // Name of file containing parameters.
        out_filename = *++argv ; // Name of file where element
                           // syntax and descriptions are written.
        latex_filename = *++argv ; // Name of file where the best
                                                        // sequence found is written.
        plot_filename = *++argv ; // Name of file where plotting
                                                        // information will be written.
        SP_main() ;

        return 0;
}
```

2.2 SP71_comp.cpp

```
// SP71: a program for unsupervised learning in the SP framework.

// Copyright (C) 2005 J G Wolff.

// This program is released to the public domain without any restrictions.
// You can redistribute it and/or modify it as you wish.

// In any publication that refers to any version of this program, please
// acknowledge J G Wolff as the originator of the program.

// A description of SP70, the precursor of this program, may be found
// in "Unsupervised learning in a framework of information compression
// by multiple alignment, unification and search", J G Wolff, March 2002.
// A copy may be obtained from http://uk.arxiv.org/abs/cs.AI/0302015.
// See also http://www.cognitionresearch.org.uk/papers/ul/ul.htm.

// Please retain this notice on all versions.

// Dr J G Wolff, www.CognitionResearch.org.uk.
// Email: jgw@cognitionresearch.org.uk.

// April 2005.

#define PROGRAM "SP71"

#define VERSION "9.0"

/*
**
** ABSTRACT:        Development of the SP model for unsupervised inductive
**                                learning.
**
** ENVIRONMENT: The module requires the header file "SP71_head.h"
**                                to be on the 'include' path. It also makes use of
**                                routines defined in "SP71_lib.cpp"
**
```

```
** AUTHOR:               Gerry Wolff
**
** MODIFIED BY:
**
** SP71:
**
** 1.0 JGW 18/6/03 -       Derived from SP70, v 9.2. The main objectives in this
**                                       version are described in sp71_od, %1.
**
**                                       Since mis-matches do not occur with simple examples, the
**                                       first step will be to try to implement item 2 in that
**                                       discription. Because learning during parsing results
**                                       in the creation of 'composite' alignments, each of which
**                                       is also a 'full' alignment, it should not be necessary to
**                                       re-parse the PFNs in order to derive frequency values needed
**                                       for the process of compiling grammars. After a single parsing
**                                       of each PFN, it should be possible to move straight on to
**                                       compiling alternative grammars. In this version, compiling
**                                       of grammars is done after the parsing of each PFN. To allow
**                                       for the fact that early parses are poor (because the system
**                                       has not yet acquired much knowledge), compiling of grammars
**                                       is based on a subset of the PFNs and these are always the
**                                       most recently parsed.
** 1.1 JGW 12/7/03 -       After each PFN has been processed, the best one or two grammars
**                                       are selected, the patterns in them are cleaned up by removing
**                                       unnecessary ID-symbols, ID-symbols are renamed in a tidy manner
**                                       and Old is purged of all patterns that are not in those grammars.
**                                       The variables for numbering ID-symbols are reset in an appropriate
**                                       manner. See sp71_OD, %4.
** 1.2 JGW 20/10/03 -       Re-introduces re-parsing as a way of achieving accurate
**                                       frequency values for patterns prior to the compiling
**                                       of grammars (see sp71_od, %6).
** 1.3 JGW 27/10/03 -       This version attempts to keep track of discontinuous
**                                       dependencies by including within each grammar, the
**                                       encoded form of each of the original patterns. The program
**                                       minimises (G + E), as before.
** 1.4 JGW 4/11/03 -       New patterns are created from a partial match between
**                                       *two* patterns, as described in sp71_od, %11.
** 1.5 JGW 9/11/03 -       New patterns created from an alignment of two
**                                       patterns as described in sp71_od, %12.
** 1.6 JGW 10/11/03 -       When a mis-match between Old patterns is detected
**                                       during parsing in Phase 1, the program invokes
**                                       extract_patterns() to create new patterns to be
**                                       added to old_patterns. In this case, the alignment
**                                       containing the mis-match is treated as an alignment
**                                       between the driving pattern and the target pattern
**                                       and each of these are treated as simple patterns
**                                       (see sp71_od, %13).
** 1.7 JGW 11/11/03 -       Continues development of 1.6 (which now includes
**                                       functions for displaying hit sequences in
**                                       horizontal or vertical format) and revises
**                                       extract_patterns() to work with hit sequences
**                                       rather than alignments. This saves having to
**                                       construct alignments which are then discarded.
** 1.8 JGW 5/12/03 -       Returns to alignments that omit columns for
**                                       non-hit New symbols. Continues development.
**                                       Tries a new version of compare_patterns that
**                                       ensures that, if two left brackets are matched,
**                                       then the corresponding pair of right brackets
**                                       must be matched. This attempt fails: checking
**                                       the validity of bracket matches at the time
**                                       hits are recorded is not straightforward.
** 1.9 JGW 8/12/03 -       This version checks the validity of bracket
**                                       matchings just before a hit sequence is
**                                       converted into an alignment. If the hit sequence
**                                       contains invalid matchings of brackets, it is
**                                       rejected.
** 1.10 JGW 9/12/03 -       Modified so that brackets no longer have the
**                                       status IDENTIFICATION and discrimination symbols
**                                       are added only when they are actually needed
**                                       (see sp71_od, %16).
** 2.0 JGW 9/12/03 -       Creates version 3 of extract_patterns().
** 2.1 JGW 12/12/03 -       Introduces index for matches between brackets, as
**                                       described in sp70_od, %18.
** 2.2 JGW 22/1/04 -       Scraps the system for checking legal matches between
**                                       brackets in v 2.1 and replaces it with a system that
**                                       checks left and right brackets in any one pattern (see
**                                       sp71_od, %20). This version turns out to be very
**                                       inefficient owing to the multiplicity of hit
**                                       sequences arising from unconstrained matching of
**                                       brackets.
** 2.3 JGW 22/1/04 -       This enforces legal matches between brackets in
**                                       the same way as in SP62 - by only matching 'outer'
```

```
**                                      brackets with a legal pair of 'inner' brackets.
** 2.4 JGW 23/1/04 -       Refinements as described in sp71_od, %21.
** 2.5 JGW 28/1/04 -       Refinement of cleaning up process for grammars,
**                                      as described in sp71_od, %23.
** 3.0 JGW 13/5/04 -       1 Any kind of mismatch or incomplete matching of
**                                      a pattern during parsing should lead to the
**                                      creation of new coded patterns to be added to Old.
**                                      2 After each pattern from new is parsed, with
**                                      the creation of new patterns for Old, all the
**                                      New patterns up to and including the current
**                                      New pattern are reparsed, sifting and sorting
**                                      is applied, and all Old patterns other than those
**                                      in the best one or two grammars are discarded.
**                                      This should help to reduce the size of Old and
**                                      speed up later processing. It also enables one
**                                      to get an idea of how the grammars are developing.
**                                      See sp71_od, %27.
** 4.1 JGW 2/6/04 -                Derived from v 3.0. This version attempts to
**                                      concatenate parsings when the main parsing
**                                      process is finished and then to learn new
**                                      structures from such parsings. See sp71_od, %31.
** 5.0 JGW 7/6/04 -                Only 'full' alignments are accepted at the end
**                                      of each cycle for passing on to the next cycle,
**                                      ie alignments in which all the C-symbols are
**                                      matched. At the end of each cycle, the system
**                                      tries to combine sub-alignments to find about 2
**                                      'good' combinations of sub-alignments.
**                                      See sp71_od, %33.
** 5.1 JGW 11/6/04 -       Modifies the functions for writing out alignments
**                                      so that all symbols are seen in their correct
**                                      positions relative to other symbols.
**                                      See sp71_od, %34.
** 5.2 JGW 12/6/04 -       Re-writes compare_patterns_with_brackets() as
**                                      described in sp71_od, %35.
** 5.3 JGW 15/6/04 -       Sorting out problems with parsing and formation
**                                      of composite alignments as described in
**                                      sp71_od, %36.
** 5.4 JGW 15/6/04 -       Creates abstract patterns and composite alignments
**                                      from 'good' combinations (see sp71_od, %37).
** 5.5 JGW 21/6/04 -       Distinguishes 'driving' alignments from all of
**                                      current_alignments and adds code pattern to Old
**                                      for every completed alignment, as described in
**                                      sp71_od, %38 and %39.
** 5.6 JGW 25/6/04 -       Re-uses CLASS_SYMBOLs in different contexts.
**                                      (see sp71_od, %41.)
** 6.0 JGW 23/7/04 -       Reorganisation of SP71 as described in sp71_od,
**                                      %45 (10, step 1). The overall aim is integrate
**                                      recursive checking of newly-created patterns
**                                      with the recursive building of multiple
**                                      alignments. Explicit building of multiple
**                                      alignments will be replaced by the successive
**                                      processing of code patterns. If explicit
**                                      multiple alignments are required, they
**                                      can be derived from the encodings as required.
**                                      This version does parsing with code patterns but
**                                      does not yet integrate the generation of code
**                                      patterns with parsing (see sp71_od, %47).
** 6.1 JGW 28/7/04 -       Attempts to integrate the creation of code patterns
**                                      with parsing, as described in sp71_od, %47.
** 6.2 JGW 30/7/04 -       Continues integration and implements composite rule
**                                      described in sp71_od, %50.
** 6.3 JGW 3/8/04 -                Develops the idea that each unified pattern assumes
**                                      the role of the abstract pattern and a succession
**                                      of unmatched portions of an alignment are encoded
**                                      as a single sequence, not a sucession of discrete
**                                      sequences. See sp71_od, %52.
** 7.0 JGW 5/8/04 -                Returns to alignments as the basis for learning
**                                      rather than code patterns (see sp71_od, %53). This
**                                      version builds on the developments in v 6.3.
** 7.1 JGW 10/8/04 -       Re-introduces combine_alignments(), retains the
**                                      requirement that only 'full' alignments can proceed
**                                      to the next cycle of recognise() and postpones
**                                      learning until recognise() is finished. (See
**                                      sp71_od, %55 to %59). This version not completed.
** 7.2 JGW 6/12/04 -       A new start as described in sp71_od, %62.
** 7.3 JGW 17/12/04 -       Further development with the main goal to provide
**                                      *alternative* encodings of null elements (sp71_od,
**                                      %65) with subsidiary goals as described in that
**                                      section.
** 7.4 JGW 20/12/04 -       Creates new abstract patterns when required rather
**                                      than in response to null patterns (see sp71_od, %67).
** 7.5 JGW 28/12/04 -       Tries to sort out apparent anomalies in the
**                                      sifting and sorting part of the program as
```

```
**                                          described in sp71_od, %72.
** 7.6 JGW 5/1/05 -               Rationalisation of class symbols and discrimination
**                                          symbols (to become context symbols and unique
**                                          identifiers) as described in sp71_od, %77.
** 7.7 JGW 10/1/05 -        Tries learning by merging all the Old patterns in
**                                          the reference alignment as described in sp71_od, %79.
**                                          This version was not completed and contains
**                                          programming errors.
** 7.8 JGW 12/1/05 -        Attempts learning that preserves intermediate-level
**                                          structures, as described in sp71_od, %81.
** 7.9 JGW 25/1/05 -        Further development as described in sp71_od, %83
**                                          (3): To ensure lossless compression,
**                                          create code patterns for each New pattern and include
**                                          them with the grammars for these patterns.
**                                          After reflection, this item is dropped
**                                          (see sp71_od, %84).
**                                          This version also introduces the status
**                                          BOUNDARY_MARKER for LEFT_BRACKET or RIGHT_BRACKET
**                                          at the beginnings and ends of patterns (as in
**                                          SP62). In Phase 2, only FULL_B alignments are selected
**                                          as described in sp71_od, %84.
** 7.10 JGW 31/1/05 -        Further development as described in sp71_od, %83
**                                          (4 and 5):  4 To ensure that the abstract
**                                          patterns have a function in compression, only
**                                          introduce '#' symbols when they are actually
**                                          needed to differentiate alternatives, re-use
**                                          the same symbols in different contexts, and give
**                                          these symbols the minimum number of bits needed
**                                          to differentiate them from each other. On reflection,
**                                          unique ID symbols are better than discrimination
**                                          symbols and they will be retained (see sp71_od,
**                                          %86). 5 Clean up grammars more often.
** 7.12 JGW 31/1/05 -        Derived from v 7.10. This version has two separate
**                                          learning procedures for creating new patterns to be
**                                          added to Old: one applies *within* the scope of
**                                          an alignment, as before, and the other applies
**                                          *outside* the scope of one or more alignments and
**                                          can be applied to composite alignments as well as
**                                          single alignments. The program also enforces the
**                                          rule that an alignment can only proceed from one
**                                          parsing cycle to the next if all its CONTENTS symbols
**                                          are matched (as before) and all the New hit symbols
**                                          within the scope of the alignment form a coherent
**                                          sequence with no gaps in the New pattern.
**                                          (see sp71_od, %91).
** 7.13 JGW 21/2/05 -        Modifies compiling of grammars so that, for any
**                                          given pattern from New, an alignment that is
**                                          contained within another alignment is not used.
**                                          (see sp71_od, %97). Unique ID symbols are only
**                                          added to patterns if there is a genuine choice
**                                          between two alternatives in a given context or
**                                          if the pattern is a top-level pattern (see
**                                          sp71_od, %99).
** 8.0 JGW 1/3/05 -               Attempts to introduce generalisation and the
**                                          correction of over-generations (see sp71_od,
**                                          %102).
** 8.1 JGW 17/3/05 -        Attempts to fix problems when short sentences
**                                          come before long ones. This version successfully
**                                          abstracts 'correct' grammars when short sentences
**                                          come before long ones and vice versa (see sp71_od,
**                                          %109).
** 8.2 JGW 22/3/05 -        Further refinement of the program. [Converted
**                                          into v 9.0].
** 9.0 JGW 30/3/05 -        Attempts recursive application of matching
**                                          two patterns and deriving new patterns from
**                                          partial matches, as outlined in sp71_od, %112
**                                          and %113.
**
*/

#include "SP71_head.h"

#define PRINT_CD_VALUES FALSE
#define MAX_DR_PATTS_ALIGNMENT 10
#define CONTAINED_IN FALSE
#define NUMBER_OF_PRESENTATION_GRAMMARS 3

double size_of_best_grammar ;
char *in_filename ;
char *parameters_filename ;
char *out_filename ;
char *latex_filename ;
char *plot_filename ;
```

```
FILE *input_file ;
FILE *parameters_file ;
FILE *output_file ;
FILE *latex_file ;
FILE *plot_file ;

group *corpus ;
group *new_patterns ; // New patterns supplied at the start of processing
group *old_patterns ; // Old patterns supplied at the start of processing.
        // In 'learning' mode, there may not be any patterns in old_patterns.
group *original_symbols_in_corpus ;
group *symbol_types_in_old ;

group *parsing_alignments ; // This holds alignments for the
        // current New pattern.
group *cumulative_parsing_alignments ; // In Phase 2, this holds
        // alignments for all the new patterns up to and including
        // the current New pattern.
group *selected_cumulative_parsing_alignments ;
group *symbols_in_new ; // A list of the types of symbols (symbols with
        // a given name) found in new_patterns.
group *set_of_grammars ;
group *set_of_combinations ;
group *best_combinations ;
sequence *current_new_pattern ;
grammar *naive_grammar ; // Contains the patterns from New, each one
        // with added code symbols so that it can be used for encoding
        // data.
hit_node *hit_root ; /* A pointer to the root of the hit structure */
hit_node *hn_master ; /* Used in add_hit(). */
int original_alphabet_size ; // The size of the alphabet of symbols
        // used in the original corpus.
int current_alphabet_size ; // The size of the alphabet after learning
        // has been done, including the creation of new code symbols.
int group_ID_number = 1 ;
int sequence_ID_number = 1 ;
int hit_node_ID_number = 1 ;
int combination_ID_number = 1 ;
int grammar_ID_number = 1 ;
int cycle ;
int phase ;
int new_patterns_counter ;
double score_for_best_grammar, G_for_best_grammar,
        E_for_best_grammar, G_naive = 0, E_naive = 0, GE_naive,
        raw_data = 0, compression ;

/*** ARRAYS, CONSTANTS AND INDICES FOR STORING SYMBOL COUNTS FOR
EACH SYMBOL IN NEW AND FOR STORING THE HIT STRUCTURE ***/

/*** STRUCTS, ARRAYS, CONSTANTS AND INDICES ***/

struct pnc_entry
{
        sequence *al1 ; // The alignment from which the pnc comes.
        sequence *pnc ; // A 2-row alignment which represents the pnc.
        int next_set ; // An index pointer to the next set of pncs.
        int next_pnc ; // An index pointer to the next pnc in
                                // given set.
} ; // Used in probabilities_of_inferences

hit_node **leaf_array ; /* leaf_array[] is used for recording the
        leaf nodes of the hit structure. */

int *sort_array ; /* sort_array[] is used for sorting the
        entries in leaf_array[]. */

int fe_sort ; /* The first empty entry in sort_array[]. */

sequence **alignments_array ;

sequence *original_patterns_array[MEDIUM_SCRATCH_ARRAY_SIZE] ;

int half_hit_structure_rows ;

struct patterns_counts_record
{
        sequence *pattern1 ;
        int frequency, max_cioa, count_in_one_alignment ;
                // max_cioa = "maximum count in one alignment".
                // frequency is the cumulative value of max_cioas
                // for successive New patterns.
} ;
```

```
struct patterns_counts_record *patterns_counts_array ;

int size_of_old_patterns ;

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        For the grammars in set_of_grammars, this function writes
**        out each grammar.
**
** CALLING SEQUENCE:
**
**        void write_grammars(int limit, bool write_derived_from)
**
** FORMAL ARGUMENTS:
**
**        Return value:                        void
**
**        limit:                                        The number of grammars to be written.
**
**        write_derived_from:               If true, 'derived_from' field is written,
**                                                     otherwise it is not.
**
** IMPLICIT INPUTS:
**
**        set_of_grammars
**
** IMPLICIT OUTPUTS:
**
**        Cleaned up grammars.
**
** SIDE EFFECTS:
**
**        NONE
*/

void write_grammars(int limit, bool write_derived_from)
{
        grammar *grammar1 ;
        int counter = 0 ;

        list_for(grammar1, grammar, set_of_grammars)
        {
                grammar1->write_grammar(write_derived_from, false) ;
                grammar1->write_grammar(write_derived_from, true) ;
                if (++counter >= limit) break ;
        }
} // write_grammars

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        For each pattern in old_patterns, this function counts the number
**        of times it is recognised in full_alignments (all the
**        full alignments formed for ONE pattern from New). This
**        count is the maximum number of times it is 'completely' matched
**        in any ONE alignment for a given pattern from New. In this context,
**        'complete' matching means that all the CONTENTS symbols of
**        the given pattern are matched and that all the CONTENTS
**        symbols of all the other patterns in the alignment are matched.
**
**        Notice that this method of counting avoids spurious double counting
**        that might arise from the fact that any given pattern from New
**        may appear in several different alternative alignments. Amongst
**        these alternative alignments, we are only interested in the
**        *maximum* count in any *one* alignment for a given pattern from New.
**        These maximum counts for each pattern from New are summed across
**        the one or more patterns in New.
**
** CALLING SEQUENCE:
**
**        void count_patterns_and_symbol_types(sequence *cnp)
**
**        cnp:                The last New pattern to be processed in
**                                the main program. If the value is NIL,
**                                all patterns are to be processed.
**
** FORMAL ARGUMENTS:
**
```

```
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        old_patterns, full_alignments
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void count_patterns_and_symbol_types(sequence *cnp)
{
        sequence *pattern1, *pattern2, *alignment1, *pattern3, *pattern4 ;
        int sequence_depth, row ;

        delete symbol_types_in_old ; // This deletes the group and
                // all the symbols in the group. The next function
                // re-creates everything from scratch.

        symbol_types_in_old =
                old_patterns->compile_alphabet(&current_alphabet_size) ;

        size_of_old_patterns = old_patterns->count_number_of_children() ;
        patterns_counts_array =
                new patterns_counts_record[size_of_old_patterns] ;

        int index1 = 0 ;

        list_for(pattern1, sequence, old_patterns)
        {
                patterns_counts_array[index1].pattern1 = pattern1 ;
                patterns_counts_array[index1].frequency = 0 ;
                index1++ ;
        }

        // For each 'full' alignment, count the number of occurrences
        // of each pattern from Old in the alignment. For each New pattern,
        // and any given Old pattern, find the maximum number of times that
        // Old pattern appears any alignment that parses the New pattern. Add
        // that value to the frequency value for the Old pattern.

        // Print alignments where the New pattern and all the Old patterns
        // are fully matched.

        fprintf(output_file, "CUMULATIVE 'FULL_A' ALIGNMENTS FROM PARSING (") ;
        print_pattern_cycle(false, cnp) ;
        fprintf(output_file, "):\n\n") ;

        list_for(alignment1, sequence, cumulative_parsing_alignments)
        {
                if (alignment1->get_degree_of_matching() != FULL_A) continue ;

                fprintf(output_file, "%s%d%s",
                        "ID",
                        alignment1->get_ID(),
                        ": ") ;
                alignment1->write_tree_object(PRINT_SEQUENCE_FREQUENCY) ;
        }

        cumulative_parsing_alignments->sort_by_compression_difference() ;

        // For each of the patterns in new_patterns, obtain values
        // for max_cioa and add them to the frequency values of
        // relevant patterns.

        list_for(pattern3, sequence, new_patterns)
        {
                for (index1 = 0; index1 < size_of_old_patterns; index1++)
                        patterns_counts_array[index1].max_cioa = 0 ;

                // Obtain values for count_in_one_alignment and max_cioa.

                list_for(alignment1, sequence, cumulative_parsing_alignments)
                {
                        if (alignment1->get_degree_of_matching() != FULL_A) continue ;
                        pattern4 = alignment1->get_row_pattern(0) ;
                        if (pattern4 != pattern3) continue ;
```

```
                        // We now processing only those alignments that have
                        // pattern3 as their New pattern.

                        // For each of the patterns in alignment1
                        // (except the current New pattern), update
                        // count_in_one_alignment and max_cioa.

                        for (index1 = 0; index1 < size_of_old_patterns; index1++)
                                patterns_counts_array[index1].count_in_one_alignment = 0 ;

                        sequence_depth = alignment1->get_sequence_depth() ;
                        for (row = 1; row < sequence_depth; row++)
                        {
                                pattern1 = alignment1->get_row_pattern(row) ;
                                for (index1 = 0; index1 < size_of_old_patterns; index1++)
                                {
                                        pattern2 = patterns_counts_array[index1].pattern1 ;
                                        if (pattern1 == pattern2) break ;
                                }

                                if (index1 >= size_of_old_patterns)
                                        abort_run("pattern1 not found in \
                                                count_patterns_and_symbol_types()") ;

                                (patterns_counts_array[index1].count_in_one_alignment)++ ;
                                if (patterns_counts_array[index1].count_in_one_alignment >
                                        patterns_counts_array[index1].max_cioa)
                                                patterns_counts_array[index1].max_cioa =
                                                patterns_counts_array[index1].count_in_one_alignment ;
                        }
                }

                // Now add max_cioa values to frequency counts for the patterns
                // in old_patterns.

                for (index1 = 0; index1 < size_of_old_patterns; index1++)
                        patterns_counts_array[index1].frequency +=
                                patterns_counts_array[index1].max_cioa ;

                if (pattern3 == cnp) break ;
        } // Get next New pattern.

        // Transfer final frequency values to patterns in Old and print
        // them out at the same time.

        fprintf(output_file, "PATTERNS IN OLD (") ;
        print_pattern_cycle(false, cnp) ;
        fprintf(output_file, "):\n\n") ;

        int frequency ;
        for (index1 = 0; index1 < size_of_old_patterns; index1++)
        {
                pattern1 = patterns_counts_array[index1].pattern1 ;
                frequency = patterns_counts_array[index1].frequency ;
                pattern1->set_frequency(frequency) ;
                pattern1->write_pattern(true, true) ;
        }

        // Find the frequency of each symbol type by scanning
        // over the patterns in the corpus and adding the frequency
        // of the pattern to the frequency of the symbol type for
        // each occurrence of each symbol in the pattern in which
        // it occurs.

        find_symbol_frequencies(symbol_types_in_old, false, true,
                cnp) ;

        // Compute bit_cost for each symbol type. Assign costs to
        // instances of the symbols in new_patterns and old_patterns.

        calculate_and_assign_frequencies_and_costs(symbol_types_in_old, false) ;

        delete[] patterns_counts_array ;

} // count_patterns_and_symbol_types

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks whether or not an alignment has all its New symbols matched
```

```
**        and all CONTENTS symbols in its Old patterns matched. If all New symbols
**        and all CONTENTS symbols in Old patterns are matched, the alignment is
**        marked as FULL_A. If all CONTENTS symbols in Old patterns are matched
**        but not all New symbols are matched, the alignment is marked as FULL_B.
**        Otherwise, the alignment is marked as PARTIAL.
**
** CALLING SEQUENCE:
**
**        int sequence::find_degree_of_matching(sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:               FULL_A, FULL_B or PARTIAL.
**
**        cnp:                                The New pattern being processed.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

int sequence::find_degree_of_matching(sequence *cnp)
{
        symbol *col1, *symbol1 ;
        int i, return_value ;
        bool all_new_symbols_are_matched,
                all_old_contents_symbols_matched = true ;

        // Check to see whether all the New symbols are matched.

        all_new_symbols_are_matched = this->all_new_symbols_matched(cnp) ;

        // Check whether all the CONTENTS symbols of all the Old
        // patterns are matched.

        list_for(col1, symbol, this)
        {
                if (col1->is_a_hit()) continue ;

                // Find the single symbol in an Old pattern in this column.
                // There should not be any New symbol because, in general,
                // unmatched New symbols are not entered into alignments.

                symbol1 = col1->find_unmatched_symbol() ;

                if (symbol1 == NIL)
                        abort_run("Empty column in sequence::find_degree_of_matching().") ;

                if (symbol1->get_status() == CONTENTS)
                {
                        all_old_contents_symbols_matched = false ;
                        break ;
                }
        }

        // If all the New symbols are not matched, check to see whether
        // any of the unmatched symbols fall within the scope of
        // of any one Old pattern. Strictly speaking, it should only
        // ever be necessary to check the bottom Old pattern in any
        // alignment because the rules for the formation of alignments
        // require that only FULL_B alignments can proceed from one
        // parsing cycle to the next or be involved in the formation
        // of composite alignments.

        int row, sequence_length = count_number_of_children(),
                index1, index2, new_int_pos1, new_int_pos2 ;
        symbol **col_array, *new_symbol, *old_symbol ;
        bool backwards_symbol_found, forwards_symbol_found,
                unmatched_new_within_scope = false ;
        if (all_new_symbols_are_matched == false)
        {
                // Set up the sequence as an array.

                col_array = new symbol*[sequence_length] ;
```

```
        i = 0 ;
        list_for(col1, symbol, this)
        {
                col_array[i] = col1 ;
                i++ ;
        }

        // Now search for gaps in the New-symbol hit sequence ;

        new_int_pos2 = NULL_VALUE ;
        for (index1 = 0; index1 < sequence_length; index1++)
        {
                col1 = col_array[index1] ;
                new_symbol = col1->get_row_symbol(0) ;
                if (new_symbol == NIL) continue ;

                new_int_pos1 = col1->get_row_orig_int_pos(0) ;
                if (new_int_pos2 == NULL_VALUE) goto L1 ;

                if (new_int_pos1 - new_int_pos2 > 1)
                {
                        // For each row, search backwards from index2
                        // and forwards from index1 to see whether there
                        // is any non-NIL symbol in the row. If there is
                        // in both directions, then the gap falls within
                        // the scope of the Old pattern on this row.

                        for (row = sequence_depth - 1; row >= 1; row--)
                        {
                                backwards_symbol_found =
                                        forwards_symbol_found = false ;

                                // Search backwards first.

                                for (i = index2; i >= 0; i--)
                                {
                                        col1 = col_array[i] ;
                                        old_symbol = col1->get_row_symbol(row) ;
                                        if (old_symbol != NIL)
                                        {
                                                backwards_symbol_found = true ;
                                                break ;
                                        }
                                }

                                // Now search forwards.

                                for (i = index1; i < sequence_length; i++)
                                {
                                        col1 = col_array[i] ;
                                        old_symbol = col1->get_row_symbol(row) ;
                                        if (old_symbol != NIL)
                                        {
                                                forwards_symbol_found = true ;
                                                break ;
                                        }
                                }

                                if (backwards_symbol_found
                                        && forwards_symbol_found)
                                {
                                        // We have found a break in the New hit
                                        // sequence within the scope of an
                                        // Old pattern

                                        unmatched_new_within_scope = true ;
                                        if (row < sequence_depth - 1)
                                                abort_run("Anomalous alignment in \
                                                        sequence::find_degree_of_matching().") ;
                                        goto L2 ;
                                }
                        }
                }

                L1: new_int_pos2 = new_int_pos1 ;
                index2 = index1 ;
        }

        delete[] col_array ;
}

L2: ;
```

```
        if (all_new_symbols_are_matched && all_old_contents_symbols_matched)
        {
                set_degree_of_matching(FULL_A) ;
                return_value = FULL_A ;
        }
        else if (all_old_contents_symbols_matched
                && unmatched_new_within_scope == false)
        {
                set_degree_of_matching(FULL_B) ;
                return_value = FULL_B ;
        }
        else if (all_old_contents_symbols_matched
                && unmatched_new_within_scope == true)
        {
                set_degree_of_matching(FULL_C) ;
                return_value = FULL_C ;
        }
        else
        {
                set_degree_of_matching(PARTIAL) ;
                return_value = PARTIAL ;
        }

        return(return_value) ;

} // sequence::find_degree_of_matching

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks whether or not an alignment has all the DATA_SYMBOLs in
**        it Old patterns matched to symbols in New.
**
** CALLING SEQUENCE:
**
**        bool sequence::has_all_data_symbols_matched()
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if all DATA_SYMBOLs are matched to symbols in New,
**                                    false otherwise.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

bool sequence::has_all_data_symbols_matched()
{
        symbol *col1, *symbol1, *new_symbol ;
        int row ;

        list_for(col1, symbol, this)
        {
                // Look in rows below row 0 to see that any Old symbols have
                // type DATA_SYMBOL. Any such symbol must be matched to
                // a New symbol, otherwise the function returns false.

                for (row = 1; row < sequence_depth; row++)
                {
                        symbol1 = col1->get_row_symbol(row) ;
                        if (symbol1 == NIL) continue ;
                        if (symbol1->get_type() != DATA_SYMBOL) break ;

                        // We have an Old symbol that is a DATA_SYMBOL.

                        new_symbol = col1->get_row_symbol(0) ;
                        if (new_symbol == NIL) return(false) ; // We have found
                                // an Old DATA_SYMBOL that is not matched with a
                                // New symbol.
                        break ; // There is no need to carry on testing
                                // other symbols in the column.
```

```
                }
        }

        return(true) ; // All the DATA_SYMBOLs in the alignment are matched
                // to New symbols.

} // sequence::has_all_data_symbols_matched

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        For the 'full' alignments stored in
**        full_alignments, re-compute codes and
**        encoding costs. Compute total_cost for each of the patterns in
**        new_patterns and old_patterns.
**
** CALLING SEQUENCE:
**
**        void compute_scores(sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        cnp:                        The last New pattern to be processed. If the
**                                        value is NIL, all New patterns are to be
**                                        processed.
**
** IMPLICIT INPUTS:
**
**        full_alignments
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void compute_scores(sequence *cnp)
{
        sequence *al1, *pattern1 ;
        list_element *el_pos1 = NIL ;

        list_for_el_pos(al1, sequence, cumulative_parsing_alignments, el_pos1)
        {
                if (al1->get_degree_of_matching() != FULL_A) continue ;
                al1->correct_column_values() ; // Each column receives the
                        // frequency and bit_cost values for the symbol type
                        // within the column (changed as a result of the
                        // application of count_patterns_and_symbol_types()).
                al1->make_code(false) ;
        }

        // Compute costs for patterns in new_patterns

        fprintf(output_file, "Encoding costs for New patterns:\n\n") ;

        el_pos1 = NIL ;
        list_for_el_pos(pattern1, sequence, new_patterns, el_pos1)
        {
                pattern1->compute_costs() ;

                fprintf(output_file, "%s%d%s%1.2f%s",
                        "ID",
                        pattern1->get_ID(),
                        ": ",
                        pattern1->get_encoding_cost(),
                        " ") ;
                pattern1->write_tree_object(PRINT_NO_FREQUENCIES) ;

                if (pattern1 == cnp) break ;
        }

        // Compute costs for patterns in old_patterns.

        fprintf(output_file, "Encoding costs for Old patterns:\n\n") ;
```

```
        el_pos1 = NIL ;
        list_for_el_pos(pattern1, sequence, old_patterns, el_pos1)
        {
                pattern1->compute_costs() ;

                fprintf(output_file, "%s%d%s%1.2f%s",
                        "ID",
                        pattern1->get_ID(),
                        ": ",
                        pattern1->get_encoding_cost(),
                        " ") ;
                pattern1->write_tree_object(PRINT_SEQUENCE_FREQUENCY) ;
        }

} // compute_scores

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        This function copies the patterns in the three best grammars
**        into three new grammars and then cleans out unnecessary symbols,
**        and renumbers the symbols tidily. Copying is necessary because it
**        often happen that a given pattern appears in more than one of the
**        best grammars and cleaning and tidying may give a different result
**        in each case.
**
** CALLING SEQUENCE:
**
**        void clean_and_tidy_best_grammars()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void clean_and_tidy_best_grammars()
{
        grammar *grammar1, *grammar2 ;
        int counter = 0 ;

        fprintf(output_file, "CLEANING AND TIDYING OF BEST GRAMMARS:\n\n") ;

        list_for (grammar1, grammar, set_of_grammars)
        {
                if (++counter > NUMBER_OF_PRESENTATION_GRAMMARS) break ;
                grammar2 = new grammar ;
                grammar2->copy_details_and_patterns(grammar1) ;
                grammar2->clean_up(grammar1->get_ID()) ;
                grammar2->tidy_up_code_symbols(grammar1->get_ID()) ;
                grammar2->delete_patterns() ;
                delete grammar2 ;
        }

} // clean_and_tidy_best_grammars

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Given a list of 'full' alignments from the parsing of patterns in New
**        (in full_alignments), this function compiles
**        a list of alternative grammars for the patterns in New and, for
**        each one, calculates a value for G (the size of the grammar)
**        and E (the size of New when it has been encoded in terms of
**        the grammar).
**
**        This function uses two constraints: full_alignment_limit (which
**        restricts the number of alternative alignments that may be
```

```
**        considered for any one pattern from New; and grammar_limit (which
**        restricts the number of alternative grammars that may be developed
**        at any one time.
**
**        This function also measures G and E for the best grammars found
**        for successive patterns from New and outputs this information in
**        a form that is suitable for plotting.
**
**        The code patterns used to derive E are included with the patterns
**        that are used to derive G.
**
** CALLING SEQUENCE:
**
**        void compile_alternative_grammars(sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        cnp:                         The last New pattern to be processed in
**                                         the main program.
**
** IMPLICIT INPUTS:
**
**        full_alignments
**
** IMPLICIT OUTPUTS:
**
**        set_of_grammars
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void compile_alternative_grammars(sequence *cnp)
{
        sequence *pattern_new, *coded_pattern_new, *pattern_row_0,
                *full_alignment ;
        int alignment_counter, grammar_counter, np_counter ;
        group *temp_set_of_grammars = new group ;
        grammar *grammar1, *grammar2, *best_grammar, *grammar3,
                *grammar_tree = new grammar ; // A NULL grammar that
                        // forms the root of the tree of grammars.
        double lowest_score, temp_score ;
        bool identical_grammars_found ;

        fprintf(output_file, "START OF COMPILING ALTERNATIVE GRAMMARS:\n\n") ;

        compute_scores(cnp) ; // For the 'full' alignments
                // stored in cumulative_parsing_alignments, re-compute codes and
                // encoding costs. Compute total_cost for each of the patterns in
                // new_patterns and old_patterns. This function also corrects
                // the values for frequency and bit_cost for each column
                // in each alignment in full_alignments and it generates
                // explicit code patterns from each full alignment.

        // Now compile grammars.

        cumulative_parsing_alignments->sort_by_compression_difference() ;

        fprintf(output_file, "Cumulative parsing alignments in order of CD:\n\n") ;

        list_for(full_alignment, sequence, cumulative_parsing_alignments)
        {
                fprintf(output_file, "%s%d%s%3.2f%s",
                        "ID",
                        full_alignment->get_ID(),
                        ", CD = ",
                        full_alignment->get_compression_difference(),
                        "\n") ;
        }

        fprintf(output_file, "\n") ;

        naive_grammar->initialise() ;
        list_element *el_pos1 = NIL ;
        np_counter = 0 ;
        list_for_el_pos(pattern_new, sequence, new_patterns, el_pos1)
        {
                np_counter++ ;
                coded_pattern_new = naive_grammar->get_next_pattern() ;
```

```
alignment_counter = 0 ;
list_for(full_alignment, sequence, selected_cumulative_parsing_alignments)
{
        if (full_alignment->is_composite_alignment()) continue ;

        if (full_alignment->get_degree_of_matching() != FULL_A) continue ;

        // Select alignments that contain pattern_new in row 0.

        pattern_row_0 = full_alignment->get_row_pattern(0) ;
        if (pattern_row_0 != pattern_new) continue ;

        alignment_counter++ ;
        if (alignment_counter > full_alignment_limit)
        {
                fprintf(output_file, "%s%d%s",
                        "FULL_ALIGNMENT_LIMIT (",
                        full_alignment_limit,
                        ") has been reached for New pattern:\n") ;
                pattern_new->write_pattern(true, false) ;
                break ;
                        // This cut-off assumes that full alignments in
                        // full_alignments have been sorted in
                        // descending order of CD score.
        }
        // full_alignment has pattern_new in its row 0.

        if (set_of_grammars->is_empty())
        {
                grammar1 = new grammar ;
                grammar_tree->add_derived_grammar(grammar1) ;
                grammar1->
                        set_derived_from_alignment(full_alignment) ;
                temp_set_of_grammars->add_child(grammar1) ;
                grammar1->compile_grammar(full_alignment) ;

                grammar1->compute_G() ;
                grammar1->increment_E(full_alignment->
                        get_encoding_cost()) ;
                grammar1->sum_G_and_E() ;

                // Diagnostic

                #if DIAGNOSTIC1
                fprintf(output_file, "DIAGNOSTIC:\n\n") ;
                grammar1->write_grammar(true) ;
                fprintf(output_file, "END DIAGNOSTIC\n\n") ;
                fflush(output_file) ;
                #endif
        }
        else
        {
                // Step through the list of grammars as it was
                // at the end of processing the previous pattern
                // from New.

                list_for(grammar1, grammar, set_of_grammars)
                {
                        grammar2 = new grammar(*grammar1) ;
                        grammar1->add_derived_grammar(grammar2) ;
                        grammar2->set_derived_from_grammar(grammar1->get_ID()) ;
                        grammar2->
                                set_derived_from_alignment(full_alignment) ;
                        grammar2->compile_grammar(full_alignment) ;

                        grammar2->compute_G() ;
                        grammar2->increment_E(full_alignment->
                                get_encoding_cost()) ;
                        grammar2->sum_G_and_E() ;

                        // Diagnostic

                        #if DIAGNOSTIC1
                        fprintf(output_file, "DIAGNOSTIC:\n\n") ;
                        grammar2->write_grammar(true) ;
                        fprintf(output_file, "END DIAGNOSTIC\n\n") ;
                        fflush(output_file) ;
                        #endif

                        // Compare grammar2 with the grammars that are
                        // already in temp_set_of_grammars. If a match
                        // is found with any of them, determine which one
                        // has the worst (highest) score and
```

```
                        // remove it from further consideration.

                        identical_grammars_found = false ;
                        list_for(grammar3, grammar, temp_set_of_grammars)
                        {
                                if (grammar2->matches(grammar3))
                                {
                                        identical_grammars_found = true ;
                                        break ;
                                }
                        }

                        if (identical_grammars_found)
                        {
                                if (grammar2->get_score() > grammar3->get_score())
                                {
                                        #if DIAGNOSTIC2
                                        fprintf(output_file, "%s%d%s%3.2f%s%d%s%3.2f%s",
                                                "New grammar GR",
                                                grammar2->get_ID(),
                                                " (score = ",
                                                grammar2->get_score(),
                                                ") is the same as the previous grammar\nGR",
                                                grammar3->get_ID(),
                                                " (score = ",
                                                grammar3->get_score(),
                                                ") and is not considered further.\n\n") ;
                                        #endif
                                        // delete grammar2 ;
                                }
                                else
                                {
                                        #if DIAGNOSTIC2
                                        fprintf(output_file, "%s%d%s%3.2f%s%d%s%3.2f%s",
                                                "Previous grammar GR",
                                                grammar3->get_ID(),
                                                " (score = ",
                                                grammar3->get_score(),
                                                ") is the same as the new grammar\nGR",
                                                grammar2->get_ID(),
                                                " (score = ",
                                                grammar2->get_score(),
                                                ") and is not considered further.\n\n") ;
                                        #endif
                                        temp_set_of_grammars->extract_child(grammar3) ;
                                        // delete grammar3 ;
                                        temp_set_of_grammars->add_child(grammar2) ;
                                }
                        }
                        else temp_set_of_grammars->add_child(grammar2) ;
                }
        }
}

// Release all the grammars in set_of_grammars

set_of_grammars->release_children() ;

// Compute the score of each grammar in temp_set_of_grammars
// as G + E. At the same time, count the number of grammars
// that have been formed.

grammar_counter = 0 ;
list_for(grammar1, grammar, temp_set_of_grammars)
{
        grammar_counter++ ;
        grammar1->sum_G_and_E() ;

        // Diagnostic

        #if DIAGNOSTIC2
        fprintf(output_file, "DIAGNOSTIC:\n\n") ;
        grammar1->write_grammar(true) ;
        fprintf(output_file, "END DIAGNOSTIC\n\n") ;
        fflush(output_file) ;
        #endif
}

if (verbose)
        fprintf(output_file, "%s%d%s%d%s",
                "Number of grammars formed for New pattern ID",
                pattern_new->get_ID(),
                " = ",
```

```
                        grammar_counter,
                        "\n\n") ;

        // Now transfer newly-updated grammars from temp_set_of_grammars
        // to set_of_grammars in ascending order of their scores (smaller
        // scores are better than bigger ones). When
        // the number of grammars has reached grammar_limit, stop
        // transfering grammars and release the remaining grammars
        // instead. Print out grammars as they are transferred.

        grammar_counter = 0 ;
        #if DIAGNOSTIC2
        fprintf(output_file, "Grammars created for New pattern:\n") ;
        pattern_new->write_pattern(true, false) ;
        #endif
        while (temp_set_of_grammars->is_empty() == false)
        {
                if (grammar_counter >= grammar_limit)
                {
                        temp_set_of_grammars->release_children() ;

                        fprintf(output_file, "%s%d%s%d%s",
                                "GRAMMAR_LIMIT (",
                                grammar_limit,
                                ") reached for New pattern, ID",
                                pattern_new->get_ID(),
                                "\n\n") ;

                        break ;
                }
                else
                {
                        lowest_score = HIGH_VALUE ;
                        list_for(grammar1, grammar, temp_set_of_grammars)
                        {
                                temp_score = grammar1->get_score() ;
                                if (temp_score < lowest_score)
                                {
                                        lowest_score = temp_score ;
                                        best_grammar = grammar1 ;
                                }
                        }

                        #if DIAGNOSTIC2
                        fprintf(output_file, "%s%d%s",
                                "GR",
                                best_grammar->get_ID(),
                                "\n") ;
                        #endif
                        temp_set_of_grammars->extract_child(best_grammar) ;

                        set_of_grammars->add_child(best_grammar) ;
                        grammar_counter++ ;
                        if (verbose) best_grammar->write_grammar(true, false) ;
                }
        }

        if (pattern_new == cnp) break ;
} // Repeat the cycle for the next pattern from New.

if (set_of_grammars->is_empty() == false)
{
        best_grammar = (grammar *)set_of_grammars->get_first_child() ;
        G_for_best_grammar = best_grammar->get_G() ;
        E_for_best_grammar = best_grammar->get_E() ;
        score_for_best_grammar = best_grammar->get_score() ;
        G_naive += coded_pattern_new->get_total_cost() ;
        raw_data += pattern_new->get_total_cost() ;
        E_naive += coded_pattern_new->get_encoding_cost() ;
        GE_naive = G_naive + E_naive ;
        naive_grammar->increment_E(E_naive) ;
        naive_grammar->increment_G(G_naive) ;
        naive_grammar->sum_G_and_E() ;
        compression = score_for_best_grammar / raw_data ;

        fprintf(plot_file,
                "%d%s%3.2f%s%3.2f%s%3.2f%s%3.2f%s%3.2f%s%3.2f%s%3.2f%s%3.2f%s",
                new_patterns_counter,
                ", ",
                G_for_best_grammar,
                ", ",
                E_for_best_grammar,
                ", ",
```

```
                        score_for_best_grammar,
                        ", ",
                        G_naive,
                        ", ",
                        E_naive,
                        ", ",
                        GE_naive,
                        ", ",
                        raw_data,
                        ", ",
                        compression,
                        "\n") ;
        }

        // Now write out the grammars in set_of_grammars.

        fprintf(output_file, "FINAL GRAMMARS (") ;
        print_pattern_cycle(false, cnp) ;
        fprintf(output_file, "):\n\n") ;

        list_for(grammar1, grammar, set_of_grammars)
                grammar1->sort_by_ID() ;

        if (set_of_grammars->is_empty())
                fprintf(output_file, "No grammars formed.\n\n") ;
        else write_grammars(3, false) ;

        if (new_patterns_counter < number_of_patterns_in_new)
        {
                // Delete all Old patterns and replace them with patterns
                // in the n_grammars best grammars.

                purge_old_patterns(n_grammars, cnp) ;
        }
        else clean_and_tidy_best_grammars() ; // This is for final presentation.

        // Release the grammars in set_of_grammars. They are still linked
        // in to the grammar_tree and will be deleted when that tree is
        // deleted (below).

        set_of_grammars->release_children() ;

        #if DIAGNOSTIC1
        fprintf(output_file, "DIAGNOSTIC:\n\n") ;

        // Write out a trace of how the grammars were formed.

        grammar_tree->write_trace(0) ;
        fprintf(output_file, "\n") ;

        fprintf(output_file, "END DIAGNOSTIC\n\n") ;
        fflush(output_file) ;
        #endif

        // Delete all the grammars in the grammar tree recursively.

        delete grammar_tree ;

} // compile_alternative_grammars

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Makes a selection of cumulative_parsing_alignments and puts them
**        in selected_cumulative_parsing_alignments. The alignments that are
**        selected are those that are FULL_A and are not driving alignments
**        for any other alignment.
**
** CALLING SEQUENCE:
**
**        void select_alignments()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        cumulative_parsing_alignments
**
** IMPLICIT OUTPUTS:
```

```
**
**        selected_cumulative_parsing_alignments
**
** SIDE EFFECTS:
**
**        NONE
*/

void select_alignments()
{
        sequence *alignment1, *al1 ;
        list_element *el_pos1 ;
        bool is_driving_alignment ;
        int full_alignment_ID ;

        cumulative_parsing_alignments->sort_by_ID() ;

        list_for(alignment1, sequence, cumulative_parsing_alignments)
        {
                if (alignment1->get_degree_of_matching() != FULL_A) continue ;

                // Check that full_alignment is not the driving pattern
                // for another alignment that is also in
                // cumulative_parsing_alignments.

                el_pos1 = NIL ;
                is_driving_alignment = false ;
                full_alignment_ID = alignment1->get_ID() ;
                list_for_el_pos(al1, sequence, cumulative_parsing_alignments,
                        el_pos1)
                {
                        if (al1->get_driving_ID() == full_alignment_ID)
                        {
                                is_driving_alignment = true ;
                                break ;
                        }
                }

                if (is_driving_alignment) continue ;

                // alignment1 is FULL_A and it is not a driving alignment
                // for any other alignment.

                selected_cumulative_parsing_alignments->add_child(alignment1) ;
        }

} // select_alignments

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Looks for alignments in which are 'fully' matched in the sense that
**        all the New symbols are matched and all the CONTENTS symbols are matched.
**        From these, the function counts frequencies of patterns and symbols.
**        The function ends by computing a set of alternative grammars.
**
** CALLING SEQUENCE:
**
**        void sifting_and_sorting(sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:               void
**
**        cnp:                                The last New pattern to be processed in
**                                                the main program.
**
** IMPLICIT INPUTS:
**
**        patterns in New
**
** IMPLICIT OUTPUTS:
**
**        Sets of alternative grammars
**
** SIDE EFFECTS:
**
**        NONE
**
*/
```

```
void sifting_and_sorting(sequence *cnp)
{
        fprintf(output_file, "START OF SIFTING AND SORTING (") ;
        print_pattern_cycle(false, cnp) ;
        fprintf(output_file, ")\n\n") ;

        select_alignments() ;

        count_patterns_and_symbol_types(cnp) ;

        fprintf(output_file, "OLD PATTERNS WITH DETAILS (") ;
        print_pattern_cycle(false, current_new_pattern) ;
        fprintf(output_file, ")\n\n") ;
        old_patterns->write_patterns_with_details() ;

        compile_alternative_grammars(cnp) ;

        fprintf(output_file, "END OF SIFTING AND SORTING (") ;
        print_pattern_cycle(false, cnp) ;
        fprintf(output_file, ")\n\n") ;

} // sifting_and_sorting

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Holding function for recognise() for one New pattern.
**
** CALLING SEQUENCE:
**
**        void recognition_cycles(sequence *pattern1, bool derive_patterns)
**
** FORMAL ARGUMENTS:
**
**        Return value:               void
**
**        pattern1:                       A new pattern or derived pattern to be recognised.
**        derive_patterns:        If true, new patterns are derived from partial
**                                                alignments, otherwise they are not.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        Alignments in parsing_alignments.
**
** SIDE EFFECTS:
**
**        NONE
*/

void recognition_cycles(sequence *pattern1, bool derive_patterns)
{
        sequence *al1 ;
        int counter, degree_of_matching_value ;

        cycle = 0 ;

        while (recognise(pattern1)) ;

        set_of_combinations->delete_children() ;

        if (derive_patterns == false) goto L1 ;

        // Derive new patterns from partial alignments.

        counter = 0 ;

        parsing_alignments->sort_by_compression_difference() ;

        fprintf(output_file, "Start of learning:\n(") ;
        print_pattern_cycle(true, pattern1) ;
        fprintf(output_file, ")\n\n") ;

        list_for(al1, sequence, parsing_alignments)
        {
                // if (al1->is_composite())
                // {
                //         fprintf(output_file, "%s%d%s",
```

```
                //                "Alignment ID",
                //                al1->get_ID(),
                //                " is composite and is excluded from learning.\n\n") ;
                //        continue ;
                // }

                degree_of_matching_value = al1->get_degree_of_matching() ;
                if (degree_of_matching_value != FULL_A)
                {
                        if (++counter > extraction_limit) break ;

                        #if DIAGNOSTIC3
                        fprintf(output_file, "DIAGNOSTIC 3A:\n\n") ;
                        old_patterns->write_patterns("OLD PATTERNS:",
                                NULL_VALUE, NULL_VALUE) ;
                        fprintf(output_file, "END DIAGNOSTIC 3A:\n\n") ;
                        fflush(output_file) ;
                        #endif

                        al1->create_patterns() ;
                }
        }

        L1: parsing_alignments->delete_children() ;

} // recognition_cycles

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Scans the Old patterns looking for ID-symbols that have no function,
**        either as identifiers within another pattern, or as discrimination
**        symbols, or as top-level identifiers. These symbols are removed.
**
** CALLING SEQUENCE:
**
**        void find_and_remove_unnecessary_ID_symbols()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        old_patterns
**
** IMPLICIT OUTPUTS:
**
**        Modification of one or more patterns in old_patterns.
**
** SIDE EFFECTS:
**
**        NONE
*/

void find_and_remove_unnecessary_ID_symbols()
{
        symbol *symbol1, *symbol2 ;
        sequence *pattern1 ;
        int status ;
        group *ID_symbols_without_any_function = new group ;

        list_for(pattern1, sequence, old_patterns)
        {
                list_for(symbol1, symbol, pattern1)
                {
                        status = symbol1->get_status() ;
                        if (status == CONTENTS) break ;
                        if (status != IDENTIFICATION) continue ;

                        // Keep record of ID symbols that have no function
                        // amongst the Old patterns.

                        if (symbol1->has_no_function())
                                ID_symbols_without_any_function->add_child(symbol1) ;
                }

                if (ID_symbols_without_any_function->is_empty())
                        continue ;

                // Now remove symbols from ID_symbols_without_any_function
```

```
                // one by one, remove them from pattern1 and delete them.

                while(symbol2 = (symbol *)
                        ID_symbols_without_any_function->extract_first_child())
                {
                        pattern1->extract_child(symbol2) ;
                        delete symbol2 ;
                }

                fprintf(output_file, "PATTERN HAS BEEN REDUCED:\n\n") ;
                pattern1->write_pattern(true, false) ;
        }

        delete ID_symbols_without_any_function ;

} // find_and_remove_unnecessary_ID_symbols

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Compress the patterns in New in terms of patterns in Old (if any).
**
** CALLING SEQUENCE:
**
**        void compress_new_into_old()
**
** FORMAL ARGUMENTS:
**
**        Return value:                        void
**
** IMPLICIT INPUTS:
**
**        The corpus
**
** IMPLICIT OUTPUTS:
**
**        Alignments and new patterns
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void compress_new_into_old()
{
        sequence // *copy_of_receptacle_pattern,
                *cnp ;
        list_element *el_pos1, *el_pos2 ;

        fprintf(output_file, "START OF PARSING AND LEARNING\n\n") ;

        el_pos1 = NIL ;
        new_patterns_counter = 0 ;
        list_for_el_pos(current_new_pattern, sequence, new_patterns, el_pos1)
        {
                new_patterns_counter++ ;
                fprintf(output_file, "NEXT PATTERN FROM NEW:\n\n") ;
                current_new_pattern->write_with_details(true) ;

                // PHASE 1

                phase = 1 ;

                fprintf(output_file, "%s%d%s",
                        "PHASE ",
                        phase,
                        "\n\n") ;

                // Prepare a "receptacle_pattern".

                receptacle_pattern = new sequence(BASIC_PATTERN) ; // Receives
                        // symbols from current_new_pattern, one by one, after
                        // each one has been matched against symbols in Old
                        // (including any symbols in receptacle_pattern).

                old_patterns->add_child(receptacle_pattern) ;

                // Setting up.

                // Do the current_new_pattern.
```

```
recognition_cycles(current_new_pattern, true) ;

// Print out receptacle pattern now that it has been
// completed. Also, count the number of children in the pattern.

fprintf(output_file, "%s%d%s",
        "From current_new_pattern ID",
        current_new_pattern->get_ID(),
        ",\nthe 'receptacle' pattern stored in Old is:\n\n") ;

receptacle_pattern->write_with_details(true) ;
receptacle_pattern->count_number_of_children() ;

naive_grammar->add_pattern(receptacle_pattern) ;

fprintf(output_file, "End of learning: (") ;
print_pattern_cycle(false, current_new_pattern) ;
fprintf(output_file, ")\n\n") ;

old_patterns->write_patterns("OLD PATTERNS AFTER LEARNING AND WITH GENERALISATIONS:", NULL_VALUE,
        NULL_VALUE) ;
fprintf(output_file, "OLD PATTERNS WITH DETAILS: (") ;
print_pattern_cycle(false, current_new_pattern) ;
fprintf(output_file, ")\n\n") ;
old_patterns->write_patterns_with_details() ;

// PHASE 2

phase = 2 ;

fprintf(output_file, "%s%d%s",
        "PHASE ",
        phase,
        "\n\n") ;

fprintf(output_file, "CURRENT PATTERN FROM NEW:\n\n") ;
current_new_pattern->write_with_details(true) ;

find_and_remove_unnecessary_ID_symbols() ;

sequence *al1 ;

el_pos2 = NIL ;
list_for_el_pos(cnp, sequence, new_patterns, el_pos2)
{
        recognition_cycles(cnp, false) ;

        parsing_alignments->sort_by_ID() ;

        fprintf(output_file, "%s%d%s",
                "Full alignments for New pattern ID",
                cnp->get_ID(),
                " (") ;
        print_pattern_cycle(true, cnp) ;
        fprintf(output_file, ")\n\n") ;

        parsing_alignments->
                write_patterns("FULL_A", NULL_VALUE, FULL_A) ;
        parsing_alignments->
                write_patterns("FULL_B", NULL_VALUE, FULL_B) ;
        parsing_alignments->
                write_patterns("FULL_C", NULL_VALUE, FULL_C) ;

        // Transfer alignments from parsing_alignments to
        // cumulative_parsing_alignments.

        while (al1 = (sequence *)
                parsing_alignments->extract_first_child())
                        cumulative_parsing_alignments->add_child(al1) ;

        if (cnp == current_new_pattern) break ;
}

// Using frequencies of patterns and symbols,
// re-compute codes and encoding costs for 'full' alignments.
// From this information, compile one or more sets of 'good'
// grammars describing the New patterns.

sifting_and_sorting(current_new_pattern) ;

fprintf(output_file,
        "All alignments are deleted ready for next New pattern.\n\n") ;
```

```
                selected_cumulative_parsing_alignments->release_children() ;
                cumulative_parsing_alignments->delete_children() ;

                fprintf(output_file, "END OF ") ;
                print_pattern_cycle(false, current_new_pattern) ;
                fprintf(output_file, "\n\n") ;

                old_patterns->write_patterns("OLD PATTERNS:", NULL_VALUE,
                        NULL_VALUE) ;

        } // Get next current_new_pattern from New.

} // compress_new_into_old

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Assigns status and types to symbols in corpus, compiles an alphabet
**        of symbols in the corpus and prints out its size,
**        finds frequencies of symbols, computes corresponding
**        information costs and assigns these values to symbols in the
**        corpus.
**
**        This function also scans the corpus looking for pre-assigned
**        IDENTIFICATION symbols. Amongst those that are numeric, it looks
**        for the highest number and then sets the global variable 'context_number'
**        to be one higher. This ensures that new IDENTIFICATION symbols that
**        are assigned by the program have higher numbers than any pre-assigned
**        numeric ID symbols.
**
** CALLING SEQUENCE:
**
**        void process_symbols()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        The corpus, original_alphabet_size
**
** IMPLICIT OUTPUTS:
**
**        Sets the value of context_number
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void process_symbols()
{
        if (!identification_symbols_marked)
                assign_status_to_symbols() ;
        else
        {
                // Make sure that left and right brackets at the beginnings
                // and ends of patterns are marked as having
                // BOUNDARY_MARKER status.

                sequence *pattern1 ;
                symbol *symbol1 ;

                list_for(pattern1, sequence, new_patterns)
                {
                        symbol1 = (symbol *)pattern1->get_first_child() ;
                        if (*(symbol1->get_name()) == '<')
                                symbol1->set_status(BOUNDARY_MARKER) ;
                        symbol1 = (symbol *)pattern1->get_last_child() ;
                        if (*(symbol1->get_name()) == '>')
                                symbol1->set_status(BOUNDARY_MARKER) ;
                }

                list_for(pattern1, sequence, old_patterns)
                {
                        symbol1 = (symbol *)pattern1->get_first_child() ;
                        if (*(symbol1->get_name()) == '<')
                                symbol1->set_status(BOUNDARY_MARKER) ;
```

```
                symbol1 = (symbol *)pattern1->get_last_child() ;
                if (*(symbol1->get_name()) == '>')
                        symbol1->set_status(BOUNDARY_MARKER) ;
        }
    }

    assign_type_to_symbols() ;

    original_symbols_in_corpus =
        corpus->compile_alphabet(&original_alphabet_size) ;
        // This function needs to follow the previous one to ensure
        // that type and status information is on each symbol in
        // the alphabet.

    fprintf(output_file, "%s%d%s",
        "Original alphabet size = ",
        original_alphabet_size,
        "\n\n") ;

    // find_symbol_frequencies(original_symbols_in_corpus,
    //        true, true, NIL) ; /* Find the frequency of
    //        each symbol type by scanning over the patterns in the corpus
    //        and adding the frequency of the pattern to the frequency of the
    //        symbol type for each occurrence of each symbol in the
    //        pattern in which it occurs. */

    // At the beginning, every New symbol has a frequency of 1,
    // assigned by default when it is created. The system derives
    // frequencies of symbols only from Old patterns created
    // as learning proceeds.

    calculate_and_assign_frequencies_and_costs(original_symbols_in_corpus, true) ;
        /* From the frequencies of symbol types this
        function computes an information cost for each symbol type
        in the corpus. Calculates average_symbol_type_cost.
        In addition, the function assigns the frequency of its type
        to each symbol in the corpus and the associated cost,
        calculated by the Shannon-Fano-Elias method. */

    // Until some Old patterns have been generated, the symbol types in
    // the Old patterns are to be the same as original_symbols_in_corpus.
    // This will be corrected as learning proceeds.

    symbol *symbol1, *symbol2 ;
    symbol_types_in_old = new group ;
    list_for(symbol1, symbol, original_symbols_in_corpus)
    {
        symbol2 = new symbol(*symbol1) ;
        symbol_types_in_old->add_child(symbol2) ;
    }

    // Process the old_patterns looking for numeric IDENTIFICATION symbols.
    // Out of these, find the highest number. Set context_number to be one
    // higher than that.

    sequence *pattern1 ;
    int highest_number = 0, string_number ;
    char *symbol_name ;

    list_for(pattern1, sequence, old_patterns)
    {
        list_for(symbol1, symbol, pattern1)
        {
            if (symbol1->get_status() != IDENTIFICATION) continue ;
            symbol_name = symbol1->get_name() ;

            // Check whether all characters are digits.

            if (isnumeric(symbol_name) == false) continue ;

            string_number = atoi(symbol_name) ;
            if (string_number > highest_number)
                    highest_number = string_number ;
        }
    }

    context_number = highest_number + 1 ;
    unique_id_number = 1 ;

} // process_symbols

/*****************************************************************/
```

```
/*
** FUNCTIONAL DESCRIPTION:
**
**        Creates new patterns and grammars by parsing.
**
** CALLING SEQUENCE:
**
**        void create_patterns_and_grammars()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void create_patterns_and_grammars()
{
        grammar *grammar1 ;

        // Preliminaries.

        // Set up arrays and clear them.

        sort_array = new int[hit_structure_rows] ;
        leaf_array = new hit_node*[hit_structure_rows] ;
        half_list = new hit_node*[half_hit_structure_rows] ;
        alignments_array = new sequence*[hit_structure_rows] ;

        int index, row1, col1 ;

        for (index = 0; index < hit_structure_rows; index++)
                sort_array[index] = NULL_VALUE ;

        for (index = 0; index < hit_structure_rows; index++)
                leaf_array[index] = NIL ;

        for (index = 0; index < half_hit_structure_rows;index++)
                half_list[index] = NIL ;

        for (index = 0; index < hit_structure_rows; index++)
                alignments_array[index] = NIL ;

        // Clear hit_node_array[].

        for (index = 0; index < HIT_NODE_ARRAY_SIZE; index++)
        {
                hit_node_array[index].new_hit_node = NIL ;
                hit_node_array[index].hn_parent = NIL ;
        }

        // Clear symbol_selection_array[][].

        for (row1 = 0; row1 < MAX_NEW_ARRAY_ROWS; row1++)
        {
                for (col1 = 0; col1 < MCOLS; col1++)
                        symbol_selection_array[row1][col1] = NIL ;
        }

        // Make counts.

        number_of_symbols_in_new =
                new_patterns->count_number_of_symbols() ; // Count the
                        // number of symbols in New.

        number_of_symbols_in_old =
                old_patterns->count_number_of_symbols() ; // Count the
        // number of symbols in Old.

        // Find the numbers of patterns in New and Old.

        number_of_patterns_in_new =
```

```
        new_patterns->count_number_of_children() ;

original_number_of_patterns_in_old =
        old_patterns->count_number_of_children() ;

// Print out the corpus.

sequence *pattern1 ;

fprintf(output_file,
        "THE WHOLE CORPUS (INCL. FREQUENCIES GREATER THAN ONE)\n\n") ;

if (new_patterns->is_empty())
        fprintf(output_file, "NEW PATTERNS: None\n\n") ;
else
{
        fprintf(output_file, "NEW PATTERNS:\n\n") ;
        list_for(pattern1, sequence, new_patterns)
        {
                pattern1->print_ID() ;
                fprintf(output_file, " ") ;
                pattern1->write_tree_object(PRINT_ALL_FREQUENCIES) ;
        }
}

if (old_patterns->is_empty())
        fprintf(output_file, "OLD PATTERNS: None\n\n") ;
else old_patterns->write_patterns("OLD PATTERNS:", NULL_VALUE,
        NULL_VALUE) ;

process_symbols() ;

// Compute the size of the corpus in bits.

double size_of_corpus_at_start = corpus->total_of_symbol_sizes() ;

fprintf(output_file, "%s%1.2g%s",
        "Size of corpus at start: ",
        size_of_corpus_at_start,
        "\n\n") ;

// Now find EC for patterns in Old.

list_for(pattern1, sequence, old_patterns)
        pattern1->compute_costs() ;

// Notice that the following block must come after the function
// assign_status_to_symbols() (called in process_symbols(), above)
// because the former depends on the original order of the symbols.

write_new_and_old() ;

/* Compress the corpus. */

compress_new_into_old() ; // This is where the real work is done.

delete sort_array ;
delete leaf_array ;
delete half_list ;
delete alignments_array ;

fprintf(output_file, "%s%1.2f%s%1.2f%s",
        "Size of corpus at start = ",
        size_of_corpus_at_start,
        " bits.\n\nSize of best grammar = ",
        size_of_best_grammar,
        " bits.\n\n") ;

grammar1 = (grammar *)set_of_grammars->get_first_child() ;
if (grammar1 == NIL)
        fprintf(output_file, "No grammar produced in repeat_cpas().\n\n") ;
else
{
        fprintf(output_file, "NAIVE GRAMMAR:\n\n") ;

        naive_grammar->write_grammar(false, false) ;
}

final_number_of_patterns_in_old =
        old_patterns->count_number_of_children() ;

/* Now write out statistics */
```

```
        fprintf(output_file, "STATISTICS\n\n") ;
        fprintf(output_file, "%s%d%s%d%s%d%s%d%s",
                "Original size of alphabet = ", original_alphabet_size,
                "\n\nFinal size of alphabet = ", current_alphabet_size,
                "\n\nOriginal number of patterns in Old = ",
                original_number_of_patterns_in_old,
                "\n\nFinal number of patterns in Old = ",
                final_number_of_patterns_in_old,
                "\n\n") ;

} // create_patterns_and_grammars

/*****************************************************************/

/* The main part of SP */

/*
** FUNCTIONAL DESCRIPTION:
**
**        Entry point to the SP system.
**
** CALLING SEQUENCE:
**
**        void SP_main()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        'output_file', 'latex_file'
**
** SIDE EFFECTS:
**
**        If the session logfile cannot be successfully opened, the function
**        exits with an error code.
**        As described in 'show_menu', the order of options displayed in
**        that function is implied here.        If the order in 'show_menu' were
**        to change then the 'switch (selection)' case labels would have to
**        be adjusted accordingly. Selecting menu option 1 (edit) has no effect.
*/

void SP_main()
{
        int i ;

        if ((input_file = fopen(in_filename, "r")) == NIL)
                abort_run("Fatal: Unable to open input_file\n") ;
        if ((parameters_file = fopen(parameters_filename, "r")) == NIL)
                abort_run("Unable to open parameters file.\n");
        if ((output_file = fopen(out_filename, "w")) == NIL)
                abort_run("Unable to open write file.\n");
        if ((latex_file = fopen(latex_filename, "w")) == NIL)
                abort_run("Fatal: Unable to open latex_file\n") ;
        if ((plot_file = fopen(plot_filename, "w")) == NIL)
                abort_run("Fatal: Unable to open plot_file\n") ;

        fprintf(output_file, "%s%s%s",
                "SP71 version ", VERSION, ". Copyright (C) ") ;
        year() ;
        fprintf(output_file, "%s%s%s%s%s%s", " J G Wolff.\n",
                "This program is released to the public domain without\n",
                "any restrictions. You can redistribute it and/or modify\n",
                "it as you wish. Please retain this notice on all\n",
                "versions and acknowledge J G Wolff as the originator\n",
                "of the program in all publications.\n\n") ;

        fprintf(output_file, "%s%s%s%s%s",
                "RESULTS FROM ", PROGRAM, ", VERSION ", VERSION, "\n\n") ;

        time_now() ;

        fprintf(output_file, "%s%s%s",
                "INPUT FILE: ", in_filename, "\n\n") ;

        // Read parameters.

        read_parameters() ;
```

```
half_hit_structure_rows = hit_structure_rows / 2 ;

// Print out parameters.

fprintf(output_file, "PARAMETERS:\n\n") ;

fprintf(output_file, "%s%d%s",
        "PROBABILITIES = ", probabilities, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "SHOW_AL_STRUCTURE = ", show_al_structure, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "SHOW_HIT_STRUCTURE = ", show_hit_structure, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "SHOW_UNSELECTED_ALIGNMENTS = ",
        show_unselected_alignments, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "VERBOSE = ",
        verbose, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "USE_ABBREVIATIONS = ",
        use_abbreviations, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "SHOW_ALL_PARSING_ALIGNMENTS = ",
        show_all_parsing_alignments, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "ID_C_SYMBOL_CONSTRAINT = ",
        id_c_symbol_constraint, "\n\n") ;
fprintf(output_file, "%s%c%s",
        "ALIGNMENT_FORMAT = ",
        alignment_format, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "HIT_STRUCTURE_ROWS = ", hit_structure_rows, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "FAIL_SCORE = ", fail_score, "\n\n") ;
fprintf(output_file, "%s%1.2f%s",
        "COST_FACTOR = ", cost_factor, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "KEEP_ROWS = ", keep_rows, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "FULL_ALIGNMENT_LIMIT = ",
        full_alignment_limit, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "GRAMMAR_LIMIT = ",
        grammar_limit, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "EXTRACTION_LIMIT = ",
        extraction_limit, "\n\n") ;
fprintf(output_file, "%s%s%s",
        "FIGURE_ID = ",
        figure_ID, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "COMBINATION_LIMIT = ",
        combination_limit, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "MAX_OLD_GAP = ",
        max_old_gap, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "MAX_DRIVING_GAP = ",
        max_driving_gap, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "MAX_TARGET_GAP = ",
        max_target_gap, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "MAX_UNSUPPORTED_CYCLES = ",
        max_unsupported_cycles, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "MAX_ALIGNMENTS_IN_ONE_CYCLE = ",
        max_alignments_in_one_cycle, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "ORIENTATION = ",
        orientation, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "WRITE_AL_CHARS_LENGTH = ",
        write_al_chars_length, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "WRITE_SECTION_CHARS_LENGTH = ",
        write_section_chars_length, "\n\n") ;
fprintf(output_file, "%s%1.2f%s",
        "MINIMUM_FONT_SIZE = ",
        minimum_font_height, "\n\n") ;
fprintf(output_file, "%s%1.2f%s",
        "MAXIMUM_FONT_SIZE = ",
```

```
        maximum_font_height, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "N_GRAMMARS = ",
        n_grammars, "\n\n") ;
fprintf(output_file, "%s%d%s",
        "CREATE_PATTERNS_DIAGNOSTIC = ",
        create_patterns_diagnostic, "\n\n") ;

/* Load SP structure */

load() ;

/* The first object in the corpus is taken to be 'New' */

corpus->initialise() ;
new_patterns = (group *)corpus->get_next_child() ;
if (new_patterns == NIL) abort_run("main(): invalid New object") ;

old_patterns = (group *)corpus->get_next_child() ;
if (old_patterns == NIL) abort_run("main(): invalid Old object") ;

// Internal constants not set in input file.

fprintf(output_file, "%s%d%s",
        "MAX_ALIGNMENT_LENGTH = ",
        max_alignment_length, "\n\n") ;

fflush(output_file) ;

if (fe_key_array == 0 && use_abbreviations != OFF) // fe_key_array
        // is used to detect whether a key to the abbreviations
        // of symbols has been supplied or not.
{
        if (use_abbreviations == LETTERS)
                make_letter_abbreviations() ;

        // if (use_abbreviations == DIGITS)
        //        make_digit_abbreviations() ; Development of the
        // use of digit abbreviations has been halted.
}

write_key() ; // Writes out the key to abbreviations of symbols
// used in the corpus.

// Make global objects and arrays.

hn_master = new hit_node ;
hit_root = new hit_node ;
parsing_alignments = new group ;
cumulative_parsing_alignments = new group ;
selected_cumulative_parsing_alignments = new group ;
symbols_in_new = new_patterns->symbol_set() ;
set_of_grammars = new group ;
naive_grammar = new grammar ;
brackets_list = new group ;
set_of_combinations = new group ;
best_combinations = new group ;
created_patterns = new group ;
generalisation_list1 = new group ;
generalisation_list2 = new group ;

write_al_chars = new char*[MAX_ALIGNMENT_DEPTH] ;
write_al_vacant_slots = new char*[MAX_ALIGNMENT_DEPTH] ;

for (i = 0; i < MAX_ALIGNMENT_DEPTH; i++)
{
        write_al_chars[i] = new char[write_al_chars_length] ;
        write_al_vacant_slots[i] = new char[write_al_chars_length] ;
}

// Put headings into plot_file.

fprintf(plot_file, "%s%s%s%s%s%s%s%s%s",
        "new pattern, ",
        "G, ",
        "E, ",
        "G + E, ",
        "G (naive), ",
        "E (naive), ",
        "G + E (naive), ",
        "O, ",
        "(G + E) / O",
        "\n") ;
```

```
        create_patterns_and_grammars() ;

        exit_routine(0) ;
} /* main */
```

2.3 SP71_head.h

```
// SP71: a program for unsupervised learning in the SP framework.

// Copyright (C) 2005 J G Wolff.

// This program is released to the public domain without any restrictions.
// You can redistribute it and/or modify it as you wish.

// In any publication that refers to any version of this program, please
// acknowledge J G Wolff as the originator of the program.

// A description of SP70, the precursor of this program, may be found
// in "Unsupervised learning in a framework of information compression
// by multiple alignment, unification and search", J G Wolff, March 2002.
// A copy may be obtained from http://uk.arxiv.org/abs/cs.AI/0302015.
// See also http://www.cognitionresearch.org.uk/papers/ul/ul.htm.

// Please retain this notice on all versions.

// Dr J G Wolff, www.CognitionResearch.org.uk.
// Email: jgw@cognitionresearch.org.uk.

// April 2005.

/*
** TITLE:                SP71_head.h
**
** VERSION:         SP71, v 9.0
**
** ABSTRACT:        The header file to be inherited by all modules in
**                                the SP system. It includes the class structure for the
**                                program.
**
** ENVIRONMENT: This file should be placed on the 'include' path of
**                                the C++ compiler so that it may be located correctly.
**                                This file produces no executable code.
**
** AUTHOR:                Gerry Wolff
**
** MODIFIED BY:
**
** SP71:
**
** 1.0 JGW 18/6/03 -        Derived from SP70, v 9.2.
** 1.1 JGW 12/7/03 -        See SP71_comp.cpp.
** 1.2 JGW 20/10/03 -        ditto.
** 1.3 JGW 27/10/03 -        ditto.
** 1.4 JGW 3/11/03 -        ditto.
** 1.5 JGW 9/11/03 -        ditto.
** 1.6 JGW 10/11/03 -        ditto.
** 1.7 JGW 11/11/03 -        ditto.
** 1.8 JGW 5/12/03 -        ditto.
** 1.9 JGW 8/12/03 -        ditto.
** 1.10 JGW 9/12/03 -        ditto.
** 2.0 JGW 9/12/03 -        ditto.
** 2.1 JGW 12/12/03 -        ditto.
** 2.2 JGW 22/1/04 -        ditto.
** 2.3 JGW 22/1/04 -        ditto.
** 2.4 JGW 23/1/04 -        ditto.
** 2.5 JGW 28/1/04 -        ditto.
** 3.0 JGW 13/5/04 -        ditto.
** 4.1 JGW 2/6/04 -                ditto.
** 5.0 JGW 7/6/04 -                ditto.
** 5.1 JGW 11/6/04 -        ditto.
** 5.2 JGW 12/6/04 -        ditto.
** 5.3 JGW 15/6/04 -        ditto.
** 5.4 JGW 15/6/04 -        ditto.
** 5.5 JGW 21/6/04 -        ditto.
** 5.6 JGW 25/6/04 -        ditto.
** 6.0 JGW 23/7/04 -        ditto.
** 6.1 JGW 28/7/04 -        ditto.
** 6.2 JGW 30/7/04 -        ditto.
** 6.3 JGW 3/8/04 -                ditto.
```

```
** 7.0 JGW 5/8/04 -               ditto.
** 7.1 JGW 10/8/04 -        ditto.
** 7.2 JGW 6/12/04 -        ditto.
** 7.3 JGW 17/12/04 -        ditto.
** 7.4 JGW 20/12/04 -        ditto.
** 7.5 JGW 28/12/04 -        ditto.
** 7.6 JGW 5/1/05 -               ditto.
** 7.7 JGW 10/1/05 -        ditto.
** 7.8 JGW 12/1/05 -        ditto.
** 7.9 JGW 25/1/05 -        ditto.
** 7.10 JGW 31/1/05 -        ditto.
** 7.12 JGW 31/1/05 -        ditto.
** 7.13 JGW 21/2/05 -        ditto.
** 8.0 JGW 1/3/05 -               ditto.
** 8.1 JGW 17/3/05 -        ditto.
** 8.2 JGW 22/3/05 -        ditto.
** 9.0 JGW 30/3/05 -        ditto.
*/

#include <stdio.h>
#include <ctype.h>
#include <time.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>

// Definitions.

#define DIAGNOSTIC1 FALSE
#define DIAGNOSTIC2 FALSE
#define DIAGNOSTIC3 FALSE
#define DIAGNOSTIC4 FALSE
#define DIAGNOSTIC5 FALSE
#define DIAGNOSTIC6 FALSE

#define TRUE 1
#define FALSE 0
#define A_DIFFERENT_FROM_B 2
#define A_SUBSET_B -1
#define A_SAME_AS_B 0
#define B_SUBSET_A 1
#define FULL_A 1 // The New pattern and all Old patterns are fully matched.
#define FULL_B 2 // The Old patterns are fully matched but the New pattern
        // is not fully matched. There are no unmatched New symbols within
        // the scope of any one Old pattern.
#define FULL_C 3 // The Old patterns are fully matched but the New pattern
        // is not fully matched. There may be unmatched New symbols within
        // the scope of any one Old pattern.
#define PARTIAL 4 // What is left after FULL_A, FULL_B and FULL_C have been removed.
#define BEST_ALIGNMENT_PRINT_LIMIT 10
#define BASIC_PATTERN 1
#define ABSTRACT_PATTERN 2
#define CODE_PATTERN 3
#define SUB_ALIGNMENT 4
#define COMPOSITE_ALIGNMENT 5
#define ALIGNMENT_FROM_PARSING 6
#define OUTPUT_COUNTER_1 5
#define OUTPUT_COUNTER_2 10
#define GAP_BETWEEN_AL_COLS 3
#define H 1
#define V 2
#define HIT_NODE_ARRAY_SIZE 10000
#define CODE_BIT_COST 4.0
#define BRACKET_BIT_COST 2.0
#define VERY_LARGE_NUMBER 1000000000
#define ALIGNMENTS_FOR_LEARNING 3 // Controls the
        // number of alignments fed into 'learning' at the end
        // of the 'recognition' phase for each pattern from New.
#define PRINTING_POINTS_PER_MM 2.8452756
#define CHARACTER_HEIGHT_WIDTH_RATIO 1.95
#define FONT_SET_SIZE 10
#define PRINTING_SPACE_LONG_SIDE 247 // The length (in mm) of the long
        // side of the space available for printing on an A4 sheet.
#define PRINTING_SPACE_SHORT_SIDE 160 // The length (in mm) of the short
        // side of the space available for printing on an A4 sheet.
#define LANDSCAPE_CHARS_PER_LINE 116 // Characters (12 pt) in one line of
        // an A4 page in landscape orientation (25 mm margins).
#define LANDSCAPE_LINES_PER_PAGE 21 // Lines in one A4 page in
        // landscape orientation (12 pt characters, 25 mm margins).
#define KEY_ARRAY_SIZE 1000
#define ELLIPSIS_MARGIN 5 // write_al_chars_limit must be at least
        // ELLIPSIS_MARGIN smaller than WRITE_ALIGNMENT_CHAR_LENGTH
        // to allow space for inserting "..." at the end of truncated
```

```
        // alignments.
#define HIGH_VALUE 1000000000 // An arbitarily high integer value used
        // in unify_best and elsewhere.
#define LOW_VALUE -1000000000 // An arbitarily low integer value used
        // in unify_best and tree_object::sort_children().
#define CONTEXT_SYMBOL 1 // Value for the type of a symbol.
#define DATA_SYMBOL 2 // Value for the type of a symbol.
#define LEFT_BRACKET 3 // Value for the type of a symbol.
#define RIGHT_BRACKET 4 // Value for the type of a symbol.
#define UNIQUE_ID_SYMBOL 5 // Value for the type of a symbol.
#define IDENTIFICATION 1 // Value for the status of a symbol.
        // See sp70_od, %47.
#define CONTENTS 2 // Value for the status of a symbol.
        // See sp70_od, %47.
#define BOUNDARY_MARKER 3 // Value for the status of a symbol.
        // Only applies to brackets at the extreme ends of a pattern.
        // Brackets that are internal to a pattern have the status CONTENTS.
        // See sp62_od, %19.
#define ORIGINAL 1
#define COPY 2
#define MAX_ALIGNMENT_DEPTH 50
#define MAX_NEW_ARRAY_ROWS 200
#define SCALING_FACTORS_SIZE 100
#define SMALL_SCRATCH_ARRAY_SIZE 10
#define MEDIUM_SCRATCH_ARRAY_SIZE 100
#define LARGE_SCRATCH_ARRAY_SIZE 1000
#define AL_CONTENTS_ARRAY_WIDTH 50
#define WRITE_ALIGNMENT_C_COLS 20
#define MCOLS 200 /* The number of columns in the best_hit_structure[] */
#define START 0
#define CONTINUE 1
#define SEQUENCE 1
#define GROUP 2
#define ON 1
#define OFF 0
#define LETTERS 2
#define DIGITS 3
#define LANDSCAPE 0
#define PORTRAIT 1
#define ALREADY_VISITED_1 1
#define ALREADY_VISITED_2 2
#define UNKNOWN 2
#define BUFSIZE 5000
#define NIL 0
#define NULL_VALUE -1
#define STRING_LENGTH 20
#define LONG_STRING_LENGTH 100
#define ESCAPE 033
#define DEL '\177'
#define PRINT_ALL_FREQUENCIES 3
#define PRINT_GROUP_FREQUENCIES 2
#define PRINT_SEQUENCE_FREQUENCY 1
#define PRINT_NO_FREQUENCIES 0
#define MAX_WRITE_LINE 75

// Macro definitions

#define list_for(X, Y, Z) for (X = (Y *)Z->get_first_child();X != NIL ; \
        X = (Y *)Z->get_next_child())

#define list_for_el_pos(X, Y, Z, P) for (X = (Y *)Z->get_next_child_by_el_pos(&P); \
        X != NIL; X = (Y *)Z->get_next_child_by_el_pos(&P))

// Definitions of classes and declarations of methods.

// CLASS STRUCTURE

//        base_element
//                list_element
//                alignment_element

//        base_object
//                symbol
//                combination
//                grammar
//                tree_object
//                        group
//                        cost_tree_object
//                                sequence
//                                hit_node

class base_element ;
class list_element ;
```

```
class alignment_element ;

class base_object ;
class symbol ;
class combination ;
class grammar ;
class tree_object ;
class group ;
class cost_tree_object ;
class sequence ;
class hit_node ;

struct key_entry
{
        char *abbreviated_symbol ;
        char *complete_symbol ;
} ;

struct row_entry
{
        sequence *row_pattern ;
        list_element *patt_last_el_pos ; // This is used to record the
                // last (ie, most recent) el_pos for row_pattern where the
                // corresponding symbol has been written into
                // write_al_chars[][].
        int fe_col_in_write_al_chars ; // Keeps track of the first empty
                // column ***for this row*** in write_al_chars[][].
        bool pattern_is_finished ; // true when all the symbols in
                // the row_pattern have been written, false otherwise.
} ;

struct hn_entry
{
        hit_node *new_hit_node ;
        hit_node *hn_parent ;
} ;

// Definitions of global variables

extern group *generalisation_list1 ;
extern group *generalisation_list2 ;
extern int new_patterns_counter ;
extern double average_symbol_type_cost ;
extern double average_CONTEXT_SYMBOL_type_cost ;
extern int id_c_symbol_constraint ;
extern group *created_patterns ;
extern group *set_of_combinations ;
extern group *best_combinations ;
extern group *brackets_list ;
extern int cycle ;
extern int phase ;
extern bool right_bracket_expected ;
extern int number_of_patterns_in_new ;
extern int original_number_of_patterns_in_old ;
extern int final_number_of_patterns_in_old ;
extern int number_of_symbols_in_new ;
extern int number_of_symbols_in_old ;
extern int fe_hit_node_array ;
extern struct hn_entry hit_node_array[] ;
extern sequence *receptacle_pattern ;
extern int half_hit_structure_rows ;
extern int total_number_of_alignments ;
extern sequence *symbol_selection_array[][MCOLS] ;
extern clock_t time1 ;
extern clock_t time2 ;
extern int original_alphabet_size ;
extern int current_alphabet_size ;
extern char *in_filename ;
extern char *parameters_filename ;
extern char *out_filename ;
extern char *latex_filename ;
extern char *plot_filename ;
extern FILE *input_file ;
extern FILE *parameters_file ;
extern FILE *output_file ;
extern FILE *latex_file ;
extern FILE *plot_file ;
extern int context_number ;
extern int unique_id_number ;
extern bool identification_symbols_marked ;
extern hit_node **half_list ;
extern char **write_al_chars ;
extern char **write_al_vacant_slots ;
```

```
extern double font_heights[] ;
extern int fe_key_array ;
extern struct key_entry key_array[] ;
extern struct row_entry row_array[] ;
extern const int max_alignment_depth ; /* The maximum depth of an sequence
        that can be written (symbols). */
extern const int max_alignment_length ;
extern int write_al_chars_length ;
extern int write_section_chars_length ;
extern double minimum_font_height ;
extern double maximum_font_height ;
extern int n_grammars ;
extern int create_patterns_diagnostic ;
extern int show_all_parsing_alignments ;
extern char alignment_format ;
extern int hit_structure_rows ;
extern int patterns_to_keep ;
extern double cost_factor ;
extern hit_node **leaf_array ; /* leaf_array[] is used for recording the
        leaf nodes of the hit structure. */
extern int *sort_array ; /* sort_array[] is used for sorting the
        entries in leaf_array[]. */
extern hit_node *hit_root ;
extern hit_node *hn_master ;
extern int fe_sort ; /* The first empty entry in sort_array[]. */
extern group *corpus ;
extern group *old_patterns ;
extern group *symbols_in_new ;
extern group *new_patterns ;
extern group *set_of_sequences ;
extern sequence *current_new_pattern ;
extern group *original_symbols_in_corpus ;
extern group *symbol_types_in_old ;
extern group *parsing_alignments ;
extern group *cumulative_parsing_alignments ;
extern group *selected_cumulative_parsing_alignments ;
extern group *set_of_grammars ;
extern grammar *naive_grammar ;
extern bool probabilities ;
extern bool show_al_structure ;
extern bool show_hit_structure ;
extern bool show_unselected_alignments ;
extern bool verbose ;
extern int use_abbreviations ;
extern int keep_rows ;
extern int driving_keep_rows ;
extern int full_alignment_limit ;
extern int grammar_limit ;
extern int extraction_limit ;
extern int not_used ;
extern int fail_score ;
extern char figure_ID[] ;
extern int combination_limit ;
extern int max_old_gap ;
extern int max_driving_gap ;
extern int max_target_gap ;
extern int max_unsupported_cycles ;
extern int max_alignments_in_one_cycle ;
extern int orientation ;
extern int group_ID_number ;
extern int sequence_ID_number ;
extern int hit_node_ID_number ;
extern int combination_ID_number ;
extern int grammar_ID_number ;
extern sequence **alignments_array ;

/* Declarations of global functions. */

extern void generalise_patterns() ;
extern void add_symbol_to_generalisation_list1(int) ;
extern bool isnumeric(char *) ;
extern void purge_old_patterns(int, sequence *) ;
extern void extract_patterns_1() ;
extern void extract_patterns_2() ;
extern symbol *make_bracket_symbol(int, int, int) ;
extern void increment_index(int *, int, char *) ;
extern sequence *recognise_select_and_write() ;
extern void probabilities_of_inferences(sequence *) ;
extern void write_new_and_old() ;
extern void assign_symbol_frequencies_and_costs(group *) ;
extern void find_symbol_frequencies(group *, bool, bool, sequence *) ;
extern void assign_type_to_symbols() ;
extern void assign_status_to_symbols() ;
```

```
extern bool recognise(sequence *) ;
extern void calculate_and_assign_frequencies_and_costs(group *, bool) ;
extern void modified_sizes_of_sfe_codes(group *) ;
extern bool is_unique(sequence *, symbol *,
        sequence **) ;
extern sequence *unify_best(int, sequence *) ;
extern void set_up_hit_structure() ;
extern sequence *make_one_alignment(hit_node *, sequence *) ;
extern bool mismatch_found(hit_node *, sequence *) ;
extern void prepare_hit_sequence(hit_node *) ;
extern symbol *make_column(int, sequence *, symbol *,
        sequence *, symbol *, int, int, bool) ;
extern void print_pattern_cycle(bool, sequence *) ;
extern void SP_main() ;
extern void time_now() ;
extern void year() ;
extern void output_alignment(sequence *) ;
extern void sort_leaf_array(int, int) ;
extern bool is_better_than(double, double) ;
extern void make_letter_abbreviations() ;
extern void make_digit_abbreviations() ;
extern void write_key() ;
extern double log_2(double) ;
extern void plus_one(int *, int, char *) ;
extern void write_hit_structure(hit_node *, int) ;
extern base_object *get_child(list_element **) ;
extern void exit_routine(int) ;
extern void clear_nodes(hit_node *) ;
extern void read_parameters() ;
extern void write_lines(FILE *, char *, int) ;
extern void abort_run(char *) ;
extern void write_message(char *) ;
extern void load() ;
extern hit_node *get_leaf_nodes_in_order(int *, sequence **, int *) ;

// CLASSES

class base_element
{
        public:
                base_element(base_object *obj1=NIL) {el_obj = obj1 ;}
                base_element(base_element &el) {el_obj = el.el_obj ;}
                ~base_element() {}

                void set_el_obj(base_object *obj1){el_obj = obj1 ;}
                base_object *get_el_obj() {return(el_obj) ;}
        protected:
                base_object *el_obj ; // Points to a symbol when used
                        // within a basic pattern. Points to an object
                        // containing an array of alignment_elements
                        // when used within an alignment.
} ; // class base_element

class alignment_element: public base_element
{
        public:
                alignment_element(base_object *obj1=NIL) : base_element(obj1)
                        {same_column_above = same_column_below = NULL_VALUE ;
                        original_pattern = NIL ;}
                alignment_element(alignment_element &al_el1) :
                                base_element(al_el1)
                        {same_column_above = al_el1.get_same_column_above() ;
                        same_column_below = al_el1.get_same_column_below() ;
                        original_pattern = al_el1.get_original_pattern() ;}
                ~alignment_element() {} // Notice that
                        // any el_obj which attached to the alignment
                        // element is *not* deleted.

                void set_same_column_above(int row) {same_column_above = row ;}
                int get_same_column_above() {return(same_column_above) ;}
                void set_same_column_below(int row) {same_column_below = row ;}
                int get_same_column_below() {return(same_column_below) ;}
                void set_original_pattern(sequence *orig_patt)
                        {original_pattern = orig_patt ;}
                sequence *get_original_pattern() {return(original_pattern) ;}
                int get_orig_patt_int_pos() ;
                void show_al_structure() ;
        protected:

                // The el_obj field (at the level of base_element) is used
                // to enter the address of the symbol for this al_el.

                int same_column_above ; // An index to to an al_el in a row
```

```
                        // above which contains a matching symbol.
                int same_column_below ; // An pointer to to an al_el in a row
                        // below which contains a matching symbol.
                sequence *original_pattern ; // The pattern from which
                        // this row of the sequence and this al_el
                        // was derived.

                // Note: the column position of each al_el is the 'position'
                // field of the column in which the al_el appears.
                // The row position of each al_el is the index value
                // for the al_el within the column in which the al_el
                // appears.
} ; // class alignment_element

class list_element: public base_element
{
        public:
                list_element(base_object *obj1=NIL) : base_element(obj1)
                        {parent = NIL ; position = NULL_VALUE ; next = NIL ;}
                list_element(list_element &el1) : base_element(el1)
                        {parent = NIL ; position = NULL_VALUE ; next = NIL ;}
                ~list_element() {}

                bool this_is_last_position()
                        {if (next == NIL) return(true) ; else return(false) ;}
                void set_parent(base_object *parent1) {parent = parent1 ;}
                base_object *get_parent() {return(parent) ;}
                void set_position(int pos) {position = pos ;}
                int get_position(){return(position) ;}
                void set_next(list_element *el1){next = el1 ;}
                list_element *get_next(){return(next) ;}
        protected:
                base_object *parent ; // The object containing a pointer
                        // to the start of the list.
                int position ; // The position in the list.
                list_element *next ; // The next list_element in the list.
} ; // class list_element

class base_object
{
        public:
                base_object() {frequency = 1 ;}
                base_object(base_object &obj) {frequency = obj.frequency ;}
                virtual ~base_object() {}
                virtual base_object *clone() {return(NIL) ;}

                virtual bool is_symbol() {return(false) ;}
                void increment_frequency() {frequency++ ;}
                void add_to_frequency(int fr) {frequency += fr ;}
                void set_frequency(int fr) {frequency = fr ;}
                int get_frequency() {return(frequency) ;}
                virtual void mark_parent_and_int_positions_recursive() {}
                virtual int count_number_of_symbols() {return(0) ;}
                virtual double total_of_symbol_sizes() {return(0) ;}
                virtual void invert_symbol_status() {}
                virtual void symbol_set_recursion(group *) {}
                virtual void wto(int) {}
                virtual void set_encoding_cost(double) {}
        protected:
                int frequency ;
} ; // class base_object

class symbol: public base_object // This is a class for simple
        // symbols and for columns of an alignment.
{
        public:
                symbol(char *nm=NIL, int al_depth=1, int hn_ID=NULL_VALUE) ;
                symbol(symbol &s) ;
                ~symbol() ;

                base_object *clone() {symbol *result = new symbol(*this) ;
                        return(result) ;}
                symbol *shallow_copy() ; // Makes a copy of a symbol
                        // without any information about rows (assumes a depth of 1).

                void assign_frequency_and_cost() ;
                bool is_symbol() {return(true) ;}
                bool name_matches(symbol *s)
                        {if (strcmp(name, s->get_name()) == 0)
                                return(true) ;
                        else return(false) ;}
                bool name_matches_string(char *str)
                        {if (strcmp(name, str) == 0)
```

```
                return(true) ;
        else return(false) ;}
bool name_is(char *nm)
        {if (strcmp(nm, name) == 0) return(true) ;
        else return(false) ;}
void set_name(char *nm)
        {delete name ; name = new char[strlen(nm) + 1] ;
                strcpy(name, nm) ;}
char *get_name() {return(name) ;}
int count_number_of_symbols() {return(1) ;}
double total_of_symbol_sizes() {return(bit_cost) ;}
void wto(int) ;
void set_encoding_cost(double e_cost) {bit_cost = e_cost ;}
void set_bit_cost(double x) {bit_cost = x ;}
double get_bit_cost() {return(bit_cost) ;}
void set_type(int tp) {if (tp != NULL_VALUE
        && tp != CONTEXT_SYMBOL && tp != DATA_SYMBOL
        && tp != LEFT_BRACKET && tp != RIGHT_BRACKET
        && tp != UNIQUE_ID_SYMBOL)
                abort_run("Invalid type for symbol::set_type") ;
        type = tp ;}
int get_type() {return(type) ;}
void set_status(int st) {if (st != NULL_VALUE
        && st != IDENTIFICATION && st != CONTENTS
        && st != BOUNDARY_MARKER)
                abort_run("Invalid status for symbol::set_status().") ;
        status = st ;}
int get_status() {return(status) ;}
void mark_parent_and_int_positions_recursive() {}
bool has_no_function() ;

// Alignment methods

symbol *find_unmatched_symbol() {if (is_a_hit() == true)
        return(NIL) ; if (sequence_depth == 1) return(NIL) ;
        int i ; symbol *s1 ; for (i = 0; i < sequence_depth; i++)
        {s1 = get_row_symbol(i) ; if (s1 != NIL) return(s1) ;}
        return(NIL) ;}
bool contains_contents_symbol() ;
void set_symbol_matches() ;
symbol *get_row_symbol(int row) {if (sequence_depth == 1 ||
        row > sequence_depth || row < 0)
                abort_run("Invalid value for row in get_row_symbol") ;
        alignment_element *al_el1 = this->get_al_el(row) ;
        return((symbol *)al_el1->get_el_obj()) ;}
int get_row_orig_int_pos(int row) {if (sequence_depth == 1 ||
        row > sequence_depth || row < 0)
                abort_run("Invalid value for row in get_row_symbol") ;
        alignment_element *al_el1 = this->get_al_el(row) ;
        return(al_el1->get_orig_patt_int_pos()) ;}
sequence *get_row_pattern(int row) {if (sequence_depth == 1 ||
        row > sequence_depth || row < 0)
                abort_run("Invalid value for row in get_row_symbol") ;
        alignment_element *al_el1 = this->get_al_el(row) ;
        return(al_el1->get_original_pattern()) ;}
void set_row_pattern(int row, sequence *pattern1)
        {if (sequence_depth == 1 ||
                row > sequence_depth || row < 0)
                        abort_run("Invalid value for row in set_row_pattern") ;
        alignment_element *al_el1 = this->get_al_el(row) ;
        al_el1->set_original_pattern(pattern1) ;}
void set_row_symbol(int row, symbol *symbol1)
        {if (sequence_depth == 1 ||
                row > sequence_depth || row < 0)
                        abort_run("Invalid value for row in set_row_symbol") ;
        alignment_element *al_el1 = this->get_al_el(row) ;
        al_el1->set_el_obj(symbol1) ;}
void show_al_structure() ;
alignment_element *get_al_el(int row)
        {if (sequence_depth == 1 ||
                row > sequence_depth || row < 0)
                abort_run("Invalid value for row in get_al_el") ;
        return(al_el_array + row) ;}
void set_sequence_depth(int al_depth)
        {sequence_depth = al_depth ;}
int get_sequence_depth() {return(sequence_depth) ;}
void make_al_el_array(int sequence_depth)
        {al_el_array = new
                alignment_element[sequence_depth] ;}
void set_al_el_array(alignment_element *al_el)
        {al_el_array = al_el ;}
alignment_element *get_al_el_array() {return(al_el_array) ;}
void set_h_node_ID(int ID)
```

```
                {h_node_ID = ID ;}
        int get_h_node_ID() {return(h_node_ID) ;}
        void set_is_a_hit(bool is_hit)
                {symbol_is_a_hit = is_hit ;}
        bool is_a_hit() {return(symbol_is_a_hit) ;}
        symbol *get_first_original_symbol()
                {symbol *s ; for (int i = 0; i < sequence_depth; i++)
                        {s = get_row_symbol(i) ; if (s != NIL) return(s) ;}
                        if (i >= sequence_depth)
                                abort_run("Error in get_first_original_symbol()") ;
                        return(NIL) ;}
        bool contains_matching_symbols()
                {symbol *s ; int counter = 0 ;
                        for (int i = 0; i < sequence_depth; i++)
                        {s = get_row_symbol(i) ; if (s == NIL) continue ;
                        counter++ ;} if (counter > 1) return(true) ;
                        else return(false) ;}
protected:
        void symbol_set_recursion(group *set) ;
        char *name ;
        double bit_cost ;
        int type ; // Records whether a given symbol is
                // CONTEXT_SYMBOL, LEFT_BRACKET, RIGHT_BRACKET or DATA_SYMBOL.
        int status ; // Records whether a given symbol is IDENTIFICATION
                // or CONTENTS or BOUNDARY_MARKER for the parent pattern.

        // Alignment fields

        int sequence_depth ;
        alignment_element *al_el_array ;
        int h_node_ID ; // The ID number of the hit node (if any)
                // which corresponds to this symbol. The ID is
                // stored rather than the hit node itself so that
                // the hit structure can be properly cleared at the
                // end of each cycle and memory is not cluttered
                // with hit nodes from previous cycles.
        bool symbol_is_a_hit ; // If a column/symbol contains two or
                // more symbols then it is a 'hit' column.
} ; // class symbol

class tree_object: public base_object
{
public:
        tree_object() : base_object() {first_element = NIL ;
                current_el_pos = NIL ; number_of_children = 0 ;}
        tree_object(tree_object &tn) ;

        bool has_member(base_object *obj1)
                {list_element *pos1 = NIL ; base_object *obj2 ;
                 list_for_el_pos(obj2, base_object, this, pos1)
                        if (obj1 == obj2) return(true) ;
                return(false) ;}
        void transfer_children(tree_object *recipient)
                {recipient->set_first_element(first_element) ;
                        first_element = NIL ;}
        void sort_children() ; // Sorts the children of a node by
                // compression_difference.
        void delete_children() {if (first_element == NIL) return ;
                base_object *child ;
                while (child = this->extract_first_child())
                        delete child ;}
        void delete_child(base_object *child) {extract_child(child) ;
                delete child ;}
        void sort_by_name() ;
        bool is_symbol() {return(false) ;}
        base_object *get_child_by_el_pos(list_element **position)
                {if ((*position) == NIL)
                        abort_run("Invalid parameter in get_child_by_el_pos()") ;
                        return((*position)->get_el_obj()) ;}
        base_object *get_child_by_int_pos(int position) ;
        base_object *get_next_child() ;
        base_object *get_next_child_by_el_pos(list_element **position) ;
        base_object *get_preceding_child_by_el_pos(list_element **position) ;
        base_object *get_preceding_child(base_object *child) ;
        list_element *get_preceding_el_pos(list_element *position) ;
        list_element *get_next_el_pos(list_element **position)
                {return((*position)->get_next()) ;}
        base_object *get_first_child() {if (first_element == NIL)
                return(NIL) ; current_el_pos = first_element ;
                return(first_element->get_el_obj()) ;}
        base_object *get_last_child() ;
        list_element *get_el_pos_by_int_pos(int position) ;
        list_element *get_el_pos_by_child(base_object *obj) ;
```

```
        list_element *get_first_el_pos() {return(first_element) ;}
        list_element *get_last_el_pos()
                {list_element *el_pos1 = first_element, *el_pos2 = NIL ;
                while (el_pos1 != NIL) {el_pos2 = el_pos1 ;
                el_pos1 = el_pos1->get_next() ;} return(el_pos2) ;}
        list_element *get_next_el_pos()
                {if (current_el_pos == NIL) return(first_element) ;
                return(current_el_pos->get_next()) ;}
        int get_first_int_pos()
                {return(first_element->get_position()) ;}
        int get_last_int_pos()
                {list_element *el_pos1 = get_last_el_pos() ;
                        return(el_pos1->get_position()) ;}
        void initialise() {current_el_pos = NIL ;}
        void set_current_el_pos(list_element *el_pos)
                {current_el_pos = el_pos ;}
        list_element *get_current_el_pos()
                {return(current_el_pos) ;}
        int get_current_int_pos()
                {if (current_el_pos == NIL) return(NULL_VALUE) ;
                else return(current_el_pos->get_position()) ;}
        int get_int_pos_by_child(base_object *child) ; // Obtain integer
                // position of child from the value stored in its list element.
        int find_int_pos_by_child(base_object *child) ; // Obtain integer
                // position of child by counting.
        void add_child(base_object *child) ;
        void add_child_at_start(base_object *child) ;
        void extract_child(base_object *child) ;
        base_object *extract_first_child()
                {if (base_object *child = this->get_first_child())
                        {this->extract_child(child) ; return(child) ;}
                else return(NIL) ;}
        void precede(base_object *child1, base_object *child2) ;
        void follow(base_object *child1, base_object *child2) ;
        void release_children() ;
        bool this_is_last_child(base_object *child) ;
        bool is_empty() {if (first_element == NIL) return(true) ;
                        else return(false) ;}
        symbol *contains_copy_of(symbol *symbol1) ;
        void mark_parent_and_int_positions_recursive() ;
        void mark_parent_and_int_positions_non_recursive() ;
        void set_first_element(list_element *el1)
                {first_element = el1 ;}
        list_element *get_first_element(){return(first_element) ;}
        void write_tree_object(int print_code) ;
        int count_number_of_children() ;
        void increment_child_count() {number_of_children++ ;}
        void set_number_of_children(int no_of_children)
                {number_of_children = no_of_children ;}
        int get_number_of_children() {return(number_of_children) ;}
        int count_number_of_symbols()
                {int no_of_symbols = 0 ; base_object *child ;
                        list_for(child, base_object, this)
                                no_of_symbols += child->count_number_of_symbols() ;
                        return(no_of_symbols) ;}
        double total_of_symbol_sizes()
                {double cumulative_size = 0 ; base_object *child ;
                        list_for(child, base_object, this)
                                cumulative_size += child->total_of_symbol_sizes() ;
                        return(cumulative_size) ;}
        void invert_symbol_status()
                {base_object *child ; list_for(child, base_object, this)
                        child->invert_symbol_status() ;}
        void symbol_set_recursion(group *set) // Called from
                                // symbol_set()
                {base_object *child ; this->initialise() ;
                        while (child = this->get_next_child())
                        child->symbol_set_recursion(set) ;}
        virtual void print_ID() = 0 ;
        int get_ID() {return(ID) ;}
        group *symbol_set() ;
        bool contains(base_object *obj1)
                {list_element *el_pos1 = NIL ; base_object *obj2 ;
                list_for_el_pos(obj2, base_object, this, el_pos1)
                        if (obj1 == obj2) return(true) ;
                return(false) ;}
    protected:
        list_element *first_element ;
        list_element *current_el_pos ;
        int number_of_children ; // Number of children of this node.
        void finish_deletion() ;
        int ID ;
} ; // class tree_object
```

```
class group: public tree_object
{
        public:
                group() : tree_object() {ID = group_ID_number++ ;}
                group(group &gr) : tree_object(gr) {ID = group_ID_number++ ;}
                ~group() {this->finish_deletion() ;}
                base_object *clone() {group *result = new group(*this) ;
                        return(result) ;}

                void add_or_merge_patterns(sequence *) ;
                bool group_matches(group *gr1) ;
                void write_patterns(char *heading, int origin, int selection) ;
                void write_patterns_with_details() ;
                void print_ID() {fprintf(output_file, "%S%d", "ID", ID) ;}
                void wto(int print_code) ;
                void compile_frequencies(group *symbol_set,
                        sequence *last_pattern) ;
                group *compile_alphabet(int *alphabet_size) ;
                void sort_by_ID() ;
                void sort_by_compression_difference() ;
                void write_IDs(int, sequence *cnp) ;
                void insert_sequence_in_order_of_CD(sequence *al1) ;
                bool contains_duplicates() ;
                bool add_pattern_without_duplicates(sequence *seq1) ;
                bool add_pattern_without_copies_or_duplicates(sequence *seq1) ;
                bool add_symbol_without_copies_or_duplicates(symbol *symbol1) ;
} ; // class group

class cost_tree_object: public tree_object
{
        public:
                cost_tree_object() : tree_object()
                        {new_symbols_cost = encoding_cost = compression_ratio =
                                compression_difference = NULL_VALUE ;}
                cost_tree_object(cost_tree_object &cto) : tree_object(cto)
                        {new_symbols_cost = cto.get_new_symbols_cost() ;
                                encoding_cost = cto.get_encoding_cost() ;
                                compression_ratio = cto.get_compression_ratio() ;
                                compression_difference =
                                        cto.get_compression_difference() ;}
                ~cost_tree_object() {}

                void set_new_symbols_cost(double NSC)
                        {new_symbols_cost = NSC ;}
                double get_new_symbols_cost() {return(new_symbols_cost) ;}
                void set_encoding_cost(double EC)
                        {encoding_cost = EC ;}
                double get_encoding_cost() {return(encoding_cost) ;}
                void set_compression_ratio(double CR)
                        {compression_ratio = CR ;}
                double get_compression_ratio() {return(compression_ratio) ;}
                void set_compression_difference(double CD)
                        {compression_difference = CD ;}
                double get_compression_difference()
                        {return(compression_difference) ;}
        protected:
                double new_symbols_cost ;
                double encoding_cost ;
                double compression_ratio ;
                double compression_difference ;
} ; // class cost_tree_object

class sequence: public cost_tree_object
{
        public:
                sequence(int org1=BASIC_PATTERN) : cost_tree_object()
                        {ID = sequence_ID_number++ ; sequence_depth = 1 ;
                        leaf_node_ID = NULL_VALUE ; abs_P = NULL_VALUE ;
                        total_cost = 0 ; origin = org1 ; keep = false ;
                        degree_of_matching = NULL_VALUE ;
                        new_this_cycle = true ;
                        driving_ID = NULL_VALUE ; target_ID = NULL_VALUE ;
                        derived_from_parsing = NULL_VALUE ;
                        single_reference = false ;
                        composite_alignment = false ;}
                sequence(sequence &seq)  : cost_tree_object(seq)
                        {ID = sequence_ID_number++ ;
                        sequence_depth = seq.get_sequence_depth() ;
                        leaf_node_ID = seq.get_leaf_node_ID() ;
                        abs_P = seq.get_abs_P() ; total_cost = seq.get_total_cost() ;
                        origin = seq.get_origin() ;
                        degree_of_matching = seq.get_degree_of_matching() ;
```

```
        keep = seq.get_keep() ;
        new_this_cycle = seq.is_new_this_cycle() ;
        driving_ID = seq.get_driving_ID() ;
        target_ID = seq.get_target_ID() ;
        derived_from_parsing = seq.get_derived_from_parsing() ;
        single_reference = seq.is_single_reference() ;
        composite_alignment = seq.composite_alignment ;}
base_object *clone() {sequence *result = new sequence(*this) ;
        return(result) ;}
~sequence() {this->finish_deletion() ;}

// Original sequence methods

bool contains_copy_of_ID_symbol(symbol *) ;
void add_unique_id_symbol() ;
bool has_unique_id_symbol() {symbol *s1 ;
        list_for(s1, symbol, this)
        {
                if (s1->get_type() == UNIQUE_ID_SYMBOL) return(true) ;
                if (s1->get_status() == CONTENTS) return(false) ;
        }
        return(false) ;}
bool check_for_single_reference() ;
bool last_symbol(symbol *s1) {symbol *s2 = (symbol *)
        get_last_child() ; if (s1 == s2) return(true) ;
        else return(false) ;}
void assign_frequencies_and_costs() ;
bool is_null() {symbol *s1 ; list_for(s1, symbol, this)
        if (s1->get_status() == CONTENTS) return(false) ;
        return(true) ;}
void add_reference(int context_number) ;
void set_all_symbols_status(int st) {symbol *s1 ; // Sets all symbols
        if (st != IDENTIFICATION && st != CONTENTS // to the same status.
                && st != BOUNDARY_MARKER)
                        abort_run("Invalid parameter for sequence::set_status().") ;
        list_for(s1, symbol, this)
                s1->set_status(st) ;}
int find_number_of_CLASS_ID_symbols() {int count = 0 ;
        symbol *s1 ; list_for(s1, symbol, this)
        {if (s1->get_type() != CONTEXT_SYMBOL) continue ;
        if (s1->get_status() != IDENTIFICATION) continue ;
        count++ ;} return(count) ;}
bool is_abstract_pattern() // Contains no DATA_SYMBOLs.
        {symbol *s1 ; list_for(s1, symbol, this)
                if (s1->get_type() == DATA_SYMBOL) return(false) ;
        return(true) ;}
bool add_context_symbol(int context_number) ;
void add_context_symbol_at_start(symbol *s1)
        {symbol *s2 = (symbol *)get_first_child() ;
        this->follow(s2, s1) ;}
int count_class_symbols()
        {symbol *symbol1 ; int count = 0 ;
                list_for(symbol1, symbol, this)
                        count++ ;
        return(count) ;}
sequence *shallow_copy() ; // Makes a copy of a 'this' sequence
        // without any information about rows (assumes a depth of 1).
bool is_copy_of(sequence *seq1) ;
symbol *get_first_cs()
        {symbol *symbol1 = (symbol *)get_first_child() ;
                if (symbol1->get_type() != LEFT_BRACKET) return(NIL) ;
                symbol1 = (symbol *)get_next_child() ;
                if (symbol1->get_type() != CONTEXT_SYMBOL)
                        return(NIL) ;
                else return(symbol1) ;}
symbol *get_next_cs()
        {symbol *symbol1 = (symbol *)get_next_child() ;
                if (symbol1->get_type() != CONTEXT_SYMBOL)
                        return(NIL) ;
                else return(symbol1) ;}
bool has_brackets()
        {list_element *pos1 = NIL ;
                symbol *symbol1 = (symbol *)
                        get_next_child_by_el_pos(&pos1) ;
                if (symbol1 == NIL) return(false) ;
                else if (symbol1->get_type() == LEFT_BRACKET
                        && symbol1->get_status() == BOUNDARY_MARKER)
                                return(true) ;
                else return(false) ;}
bool all_new_symbols_matched(sequence *cnp) ;
void add_ID_symbols(symbol *, bool) ;
sequence *check_patterns() ; // Checks 'this' against
        // pre-existing patterns to find one containing all
```

```
        // and only the same CONTENTS symbols.
void print_ID() {fprintf(output_file, "%s%d", "ID", ID) ;}
void wto(int print_code) ;
bool contents_symbols_match(sequence *pattern1) ;
void compute_costs() ;
void set_total_cost(double cost) {total_cost = cost ;}
double get_total_cost() {return(total_cost) ;}
void set_origin(int org1) {if (org1 != ABSTRACT_PATTERN &&
        org1 != BASIC_PATTERN && org1 != CODE_PATTERN
        && org1 != COMPOSITE_ALIGNMENT && org1 != SUB_ALIGNMENT)
                abort_run("Invalid parameter for sequence::set_origin()") ;
        origin = org1 ;}
int get_origin() {return(origin) ;}
void write_pattern(bool, bool) ;
int get_context_number() {char *s_name ;symbol *symbol1 =
        (symbol *)get_first_child() ;
        if (symbol1->get_type() != LEFT_BRACKET) return(NIL) ;
        symbol1 = (symbol *)get_next_child() ;
        s_name = symbol1->get_name() ;
        return(atoi(s_name)) ;}
bool is_single_reference()
        {return(single_reference) ;}
void set_single_reference(bool sr) {single_reference = sr ;}

// Alignment methods

void check_for_composite_structure() ;
bool new_hits_form_coherent_sequence() ;
bool is_legal() ;
void set_symbol_matches() {symbol *col1 ;
        list_for(col1, symbol, this)
                col1->set_symbol_matches() ;}
void create_patterns() ;
void process_old_pattern(int row) ;
int find_degree_of_matching(sequence *cnp) ;
bool has_all_data_symbols_matched() ;
void correct_column_values() ;
void compute_score_with_gaps() ;
void write_out_fully(char *, hit_node *, int, int,
        bool, sequence *) ;
void write_alignment(FILE *of, int wsl, int spp,
                char alignment_format)
        {if (alignment_format == 'H')
                write_alignment_horizontal(of, wsl, spp) ;
                else write_alignment_vertical(of) ;}
void write_alignment_vertical(FILE *) ;
void write_alignment_horizontal(FILE *, int, int) ;
void write_with_details(bool) ;
sequence *make_code(bool return_code) ;
int compute_character_length() ;
bool alignment_matches(sequence *al) ;
sequence *matches_earlier_alignment() ;
void set_sequence_depth(int al_depth)
        {sequence_depth = al_depth ;}
int get_sequence_depth() {return(sequence_depth) ;}
void set_leaf_node_ID(int ID)
        {leaf_node_ID = ID ;}
int get_leaf_node_ID() {return(leaf_node_ID) ;}
void set_abs_P(double a_P) {abs_P = a_P ;}
double get_abs_P() {return(abs_P) ;}
void show_al_structure(hit_node *h_node) ;
sequence *get_row_pattern(int row)
        {if (sequence_depth == 1 ||
                row > sequence_depth || row < 0)
                abort_run("Invalid value for row in get_al_el") ;
        symbol *al_col1 = (symbol *)this->get_first_child() ;
        alignment_element *al_el1 = al_col1->get_al_el(row) ;
        return(al_el1->get_original_pattern()) ;}
void set_degree_of_matching(int dom) {if (dom != NULL_VALUE
                && dom != FULL_A && dom != FULL_B && dom != FULL_C
                && dom != PARTIAL)
        abort_run("Invalid value for degree of matching in \
                sequence::set_degree_of_matching()") ;
        degree_of_matching = dom ;}
void set_keep(bool k) {keep = k ;}
bool get_keep() {return(keep) ;}
int get_degree_of_matching() {return(degree_of_matching) ;}
void set_new_this_cycle(bool ntc) {new_this_cycle = ntc ;}
bool is_new_this_cycle() {return(new_this_cycle) ;}
void set_driving_ID(int id_value)
        {driving_ID = id_value ;}
int get_driving_ID() {return(driving_ID) ;}
void set_target_ID(int id_value)
```

```
                        {target_ID = id_value ;}
                int get_target_ID() {return(target_ID) ;}
                void set_derived_from_parsing(int id_value)
                        {derived_from_parsing = id_value ;}
                int get_derived_from_parsing() {return(derived_from_parsing) ;}
                void set_composite_alignment(bool ca) {composite_alignment = ca ;}
                bool is_composite_alignment() {return(composite_alignment) ;}
        protected:
                // Alignment fields

                int sequence_depth ; // Records the number of rows in
                        // the sequence. An 'alignment' comprising one
                        // pattern has a depth of 1.
                int leaf_node_ID ; // The ID number of the leaf hit node
                        // (if any) which corresponds to this sequence.
                        // The ID is stored rather than the hit node itself
                        // so that the hit structure can be properly cleared
                        // at the end of each cycle and memory is not cluttered
                        // with hit nodes from previous cycles.
                double abs_P ; // The "absolute probability" of an
                        // alignment calculated from its EC (in
                        // compute_score_with_gaps).
                double total_cost ; // Total of the bit_cost values for all
                        // the symbols in the pattern.
                int origin ; // Used to mark the origin of a sequence as 'code',
                        // sub-alignment, basic pattern etc.
                bool keep ; // Records whether an alignment is to be retained
                        // on the next cycle.
                int degree_of_matching ; // FULL_A, FULL_B, FULL_C or PARTIAL.
                bool new_this_cycle ;
                int derived_from_parsing ; // If this is a code pattern, this field
                        // records the ID of the alignment from which it was derived.
                        // Otherwise it has a NULL_VALUE.
                int driving_ID ; // Records the ID of the driving pattern for
                        // this alignment.
                int target_ID ; // Records the ID of the target pattern for
                        // this alignment.
                bool single_reference ; // Marks alignments that contain a
                        // LEFT_BRACKET, a context symbol and a RIGHT_BRACKET
                        // and no other symbols.
                bool composite_alignment ;
} ; // class sequence

class hit_node: public cost_tree_object        // An element of the hit structure.
{
        public:
                hit_node() ;
                hit_node(hit_node &hn) ;
                ~hit_node() {this->finish_deletion() ;}
                base_object *clone() {hit_node *result = new hit_node(*this) ;
                        return(result) ;}

                bool all_contents_symbols_are_matched() ;
                void print_ID() {fprintf(output_file, "%c%d", '#', ID) ;}
                void set_hn_parent(hit_node *parent) {hn_parent = parent ;}
                hit_node *get_hn_parent() {return(hn_parent) ;}
                void set_driving_pattern(sequence *dr_sequence)
                        {driving_pattern = dr_sequence ;}
                sequence *get_driving_pattern() {return(driving_pattern) ;}
                void set_driving_symbol(symbol *dr_symbol)
                        {driving_symbol = dr_symbol ;}
                symbol *get_driving_symbol() {return(driving_symbol) ;}
                void set_driving_int_pos(int pos)
                        {driving_int_pos = pos ;}
                int get_driving_int_pos() {return(driving_int_pos) ;}
                void set_target_pattern(sequence *al)
                        {target_pattern = al ;}
                sequence *get_target_pattern() {return(target_pattern) ;}
                void set_target_symbol(symbol *symb)
                        {target_symbol = symb ;}
                symbol *get_target_symbol() {return(target_symbol) ;}
                void set_target_int_pos(int int_pos)
                        {target_int_pos = int_pos ;}
                int get_target_int_pos() {return(target_int_pos) ;}
                void set_leaf_entry(hit_node **node) {leaf_entry = node ;}
                hit_node **get_leaf_entry() {return(leaf_entry) ;}
                void wto(int)
                        {fprintf(output_file, "hit_node ") ;
                        this->print_ID(), fprintf(output_file, "\n\n") ;}
        protected:
                hit_node *hn_parent ;
                sequence *driving_pattern ;
                symbol *driving_symbol ;
```

```
                int driving_int_pos ;
                sequence *target_pattern ;
                symbol *target_symbol ;
                int target_int_pos ;
                hit_node **leaf_entry ; // This reverse pointer to an
                        // entry in leaf_array[] is needed so that, when
                        // a leaf node receives a child (so that it becomes a
                        // parent and is no longer a leaf node), the child
                        // may use the entry in leaf_array[] that was
                        // used by its parent (the old leaf node).
} ; // class hit_node

class combination: public base_object
{
        public:
                combination(int lfs) : base_object()
                        {C_ID = combination_ID_number++ ; len_fit_seq = lfs ;
                        fitting_sequence = new bool[len_fit_seq] ;
                        int i ; for (i = 0; i < len_fit_seq; i++)
                                fitting_sequence[i] = false ;
                        sub_alignment_list = new group ;
                        combination_score = 0 ;}
                combination(combination &comb1) ;
                ~combination(){sub_alignment_list->release_children() ;
                        delete sub_alignment_list ;
                        delete[] fitting_sequence ;}

                sequence *make_composite_alignment(sequence *) ;
                int number_of_alignments()
                        {return(sub_alignment_list->count_number_of_children()) ;}
                bool is_symbol() {return(false) ;}
                bool is_subset_of(combination *test_comb) ;
                bool can_accept(int pos_new_first, int pos_new_last) ;
                void add_sub_alignment(sequence *, int, int, sequence *) ;
                group *get_sub_alignment_list()
                        {return(sub_alignment_list) ;}
                double get_combination_score() {return(combination_score) ;}
                void set_combination_score(double c_score)
                        {combination_score = c_score ;}
                void print_combination() ;
                bool *get_fitting_sequence() {return(fitting_sequence) ;}
                void set_len_fit_seq(int length) {len_fit_seq = length ;}
                int get_len_fit_seq() {return(len_fit_seq) ;}
                void set_C_ID(int cid) {C_ID = cid ;}
                int get_C_ID() {return(C_ID) ;}
        protected:
                bool *fitting_sequence ;
                        // An array used to mark which symbols
                        // in the current symbol from New
                        // are 'covered' by one of the alignments in
                        // 'combination' (true) and which are not (false).
                int len_fit_seq ;
                group *sub_alignment_list ; // A list of basic alignments that
                        // fully or partially 'cover' the symbols in New.
                double combination_score ;
                int C_ID ;
} ; // class combination

class grammar: public base_object
{
        public:
                grammar() : base_object()
                        {G = E = score = 0; list_of_patterns = new group ;
                        ID = grammar_ID_number++ ; basis_for = NIL ;
                        derived_from_grammar = NULL_VALUE ;
                        derived_from_alignment = NIL ;}
                grammar(grammar &gr) ;
                ~grammar() {list_of_patterns->release_children() ;
                        delete list_of_patterns ; if (basis_for != NIL)
                        delete basis_for ;}

                void copy_details_and_patterns(grammar *) ;
                void delete_patterns() {list_of_patterns->delete_children() ;}
                bool contains(sequence *p1)
                        {return(list_of_patterns->contains(p1)) ;}
                void renumber_code_symbols(int) ;
                bool is_ID_symbol(symbol *s1) ;
                bool is_referenced(symbol *s1) ;
                void delete_code_patterns() {sequence *p1 ;
                        while (p1 = get_first_code_pattern())
                                list_of_patterns->delete_child(p1) ;}
                sequence *get_first_pattern() {sequence *p1 = (sequence *)
                        list_of_patterns->get_first_child() ; return(p1) ;}
```

```
                sequence *get_next_pattern() {sequence *p1 = (sequence *)
                        list_of_patterns->get_next_child() ; return(p1) ;}
                sequence *get_first_code_pattern() {sequence *p1;
                        list_for(p1, sequence, list_of_patterns)
                                if (p1->get_origin() == CODE_PATTERN) return(p1) ;
                        return(NIL) ;}
                sequence *get_first_non_code_pattern()
                        {sequence *p1 ; list_for(p1, sequence, list_of_patterns)
                                {if (p1->get_origin() != CODE_PATTERN) break ;}
                        return(p1) ;}
                sequence *get_next_non_code_pattern()
                        {sequence *p1 ;
                        while (p1 = (sequence *)list_of_patterns->get_next_child())
                                if (p1->get_origin() != CODE_PATTERN) break ;
                        return(p1) ;}
                friend sequence *single_reference(symbol *ref1, grammar *gr1) ;
                void extract_pattern(sequence *p1)
                        {list_of_patterns->extract_child(p1) ;}
                sequence *extract_first_pattern()
                        {sequence *p1 = (sequence *)
                                list_of_patterns->extract_first_child() ;
                        return(p1) ;}
                void merge_patterns() ;
                bool matches(grammar *gr1)
                        {group *patterns1 = gr1->get_list_of_patterns() ;
                        if (patterns1->group_matches(list_of_patterns))
                                return(true) ;
                        else return(false) ;}
                void tidy_up_code_symbols(int) ;
                void clean_up(int) ;
                bool add_pattern(sequence *) ;
                void release_patterns() {list_of_patterns->release_children() ;}
                void write_grammar(bool, bool) ;
                void compile_grammar(sequence *) ;
                void compute_G() ;
                void set_G(double GG) {G = GG ;}
                double get_G() {return(G) ;}
                void set_E(double EE) {E = EE ;}
                double get_E() {return(E) ;}
                void increment_E(double increment) {E += increment ;}
                void increment_G(double increment) {G += increment ;}
                void set_score(double score1) {score = score1 ;}
                void sum_G_and_E() {score = G + E ;}
                double get_score() {return(score) ;}
                int get_ID() {return(ID) ;}
                group *get_list_of_patterns() {return(list_of_patterns) ;}
                void initialise() {list_of_patterns->initialise() ;}
                void sort_by_ID() {list_of_patterns->sort_by_ID() ;}
                void set_derived_from_grammar(int id) {derived_from_grammar = id ;}
                int get_derived_from_grammar() {return(derived_from_grammar) ;}
                void set_derived_from_alignment(sequence *al) {derived_from_alignment = al ;}
                sequence *get_derived_from_alignment() {return(derived_from_alignment) ;}
                void add_derived_grammar(grammar *dg) {if (basis_for == NIL)
                        basis_for = new group ; basis_for->add_child(dg) ;}
                void write_trace(int indendation) ;

        protected:
                double G ; // Size of grammar in bits.
                double E ; // Size of New after it has been encoded in
                        // terms of the grammar.
                double score ; // The sum of G and E.
                int ID ;
                group *list_of_patterns, // The patterns in the grammar.
                        *basis_for ; // List of grammars that are derived
                                // from the given grammar.
                int derived_from_grammar ; // ID of grammar from which the
                        // present grammar was derived.
                sequence *derived_from_alignment ; // The most recent
                        // alignment from which the present grammar was derived.
} ; // class grammar
```

2.4 SP71_lib.cpp

```
// SP71: a program for unsupervised learning in the SP framework.

// Copyright (C) 2005 J G Wolff.

// This program is released to the public domain without any restrictions.
// You can redistribute it and/or modify it as you wish.
```

```
// In any publication that refers to any version of this program, please
// acknowledge J G Wolff as the originator of the program.

// A description of SP70, the precursor of this program, may be found
// in "Unsupervised learning in a framework of information compression
// by multiple alignment, unification and search", J G Wolff, March 2002.
// A copy may be obtained from http://uk.arxiv.org/abs/cs.AI/0302015.
// See also http://www.cognitionresearch.org.uk/papers/ul/ul.htm.

// Please retain this notice on all versions.

// Dr J G Wolff, www.CognitionResearch.org.uk.
// Email: jgw@cognitionresearch.org.uk.

// April 2005.

/*
** TITLE:                SP71_lib.cpp
**
** VERSION:         SP71, v 9.0
**
** ABSTRACT:        General purpose methods and functions for the SP system.
**
** ENVIRONMENT: The file "SP71_head.h" must be on the 'include'
**                                path at compile time.
**
** AUTHOR:                Gerry Wolff
**
** MODIFIED BY:
**
** SP71
**
** 1.0 JGW 18/6/03 -        Derived from SP70, v 9.2.
** 1.1 JGW 12/7/03 -        See SP71_comp.cpp.
** 1.2 JGW 20/10/03 -        ditto.
** 1.3 JGW 27/10/03 -        ditto.
** 1.4 JGW 3/11/03 -        ditto.
** 1.5 JGW 9/11/03 -        ditto.
** 1.6 JGW 10/11/03 -        ditto.
** 1.7 JGW 11/11/03 -        ditto.
** 1.8 JGW 5/12/03 -        ditto.
** 1.9 JGW 8/12/03 -        ditto.
** 1.10 JGW 9/12/03 -        ditto.
** 2.0 JGW 9/12/03 -        ditto.
** 2.1 JGW 12/12/03 -        ditto.
** 2.2 JGW 22/1/04 -        ditto.
** 2.3 JGW 22/1/04 -        ditto.
** 2.4 JGW 23/1/04 -        ditto.
** 2.5 JGW 28/1/04 -        ditto.
** 3.0 JGW 13/5/04 -        ditto.
** 4.1 JGW 2/6/04 -                ditto.
** 5.0 JGW 7/6/04 -                ditto.
** 5.1 JGW 11/6/04 -        ditto.
** 5.2 JGW 12/6/04 -        ditto.
** 5.3 JGW 15/6/04 -        ditto.
** 5.4 JGW 15/6/04 -        ditto.
** 5.5 JGW 21/6/04 -        ditto.
** 5.6 JGW 25/6/04 -        ditto.
** 6.0 JGW 23/7/04 -        ditto.
** 6.1 JGW 28/7/04 -        ditto.
** 6.2 JGW 30/7/04 -        ditto.
** 6.3 JGW 3/8/04 -                ditto.
** 7.0 JGW 5/8/04 -                ditto.
** 7.1 JGW 10/8/04 -        ditto.
** 7.2 JGW 6/12/04 -        ditto.
** 7.3 JGW 17/12/04 -        ditto.
** 7.4 JGW 20/12/04 -        ditto.
** 7.5 JGW 28/12/04 -        ditto.
** 7.6 JGW 5/1/05 -                ditto.
** 7.7 JGW 10/1/05 -        ditto.
** 7.8 JGW 12/1/05 -        ditto.
** 7.9 JGW 25/1/05 -        ditto.
** 7.10 JGW 31/1/05 -        ditto.
** 7.12 JGW 31/1/05 -        ditto.
** 7.13 JGW 21/2/05 -        ditto.
** 8.0 JGW 1/3/05 -                ditto.
** 8.1 JGW 17/3/05 -        ditto.
** 8.2 JGW 22/3/05 -        ditto.
** 9.0 JGW 30/3/05 -        ditto.
*/

#include "SP71_head.h"
```

```
#define MAKE_DIGIT_ABBREVIATIONS FALSE

/* PARAMETERS. */

/* Optional, ON or OFF. */

bool probabilities ; /* Calculations of probabilities at the end
        of each program run. */
bool show_al_structure ; // If true, the structure of each column of each
        // alignment is printed out, otherwise they are not printed.
bool show_hit_structure ; // If true, the hit structure is printed,
        // otherwise it is not printed.
bool show_unselected_alignments ; /* If ON, all the alignments which are
        formed are shown before selection (as well as showing the
        alignments which are selected. */
bool verbose ; // If ON, lots of messages are printed. Otherwise,
        // minimal messages are printed.
int use_abbreviations ; /* When LETTERS, alignments are printed
        using abbreviated names of symbols in alphabetic form (any symbol which
        starts with a digit is left as it is). When DIGITS, symbols are
        abbreviated as digits (which may include any trailing or leading '#'
        symbol which appears in the original symbol). When OFF, no abbreviations
        are used. The development of digit abbreviations has been halted. */
int show_all_parsing_alignments ;  /* If ON, all alignments produced by
        by compressing New against Old are shown. If OFF, only the best
        ones for any given pattern from New are shown. */
int id_c_symbol_constraint ; /* If this is ON, ID-symbols can only ever be matched
        with C-symbols and vice versa. If the parameter is OFF, the constraint is
        not applied. */

/* Other Parameters. */

char alignment_format ; /* The format of alignments ('V' for vertical
        and 'H' for horizontal) */

int hit_structure_rows ; /* The number of rows in the
        sort_array[] and leaf_array[]. */

int fail_score ; /* Any compression score which is equal to this or
        smaller counts as a fail. */

double cost_factor ; /* The multiplier to calculate the bit_cost of
        each data symbol from the minimum cost needed to discriminate it
        from other symbols. */

double average_symbol_type_cost ;
double average_CONTEXT_SYMBOL_type_cost ; // Includes UNIQUE_ID_SYMBOLs.

int keep_rows ; /* Controls the number of rows in the
        symbol_selection_array[][] which are used to determine which
        sequences are *not* purged at the end of each cycle. */

int driving_keep_rows ; /* The best driving_keep_rows are selected to
                                                be driving patterns for the next cycle.
                                                NOT CURRENTLY USED. */

int full_alignment_limit ; /* This limits the number of alternative
        alignments (for a given pattern from New) that are considered
        during the compiling of alternative grammars. */

int grammar_limit ; /* This limits the number of alternative grammars
        that may be developed at any one time (in
        compile_alternative_grammars()) */

int extraction_limit ; /* This limits the number of hit sequences for a given
        a given driving pattern that are used for learning. */

int not_used ; /* Not used. */

char figure_ID[20] ; /* Records the id for the data, for printing
        out in the latex_file. */

int combination_limit ; /* The number of combinations to be selected
        on each phase of combine_alignments(). */

int max_old_gap ; /* Sets the maximum permitted gap between hit symbols
        in any original old pattern (not an alignment formed during
        current processing). */

int max_driving_gap ; /* Sets the maximum permitted gap between hit symbols
        in any driving pattern. */

int max_target_gap ; /* Sets the maximum permitted gap between hit symbols
```

```
        in any target pattern. */

int max_unsupported_cycles ; /* For any one pattern from New, this parameter
        sets the maximum number of cycles which build alignments without
        any gain in maximum compression score compared with the best
        cycle for the current pattern from New. */

int max_alignments_in_one_cycle ; /* This can be used to limit the number
        of alignments formed in one cycle. It is applied as alignments are
        formed, in order of the approximate compression scores calculated for
        the hit structure. If this limit is applied, there is a possibility
        of cutting short the alignments before the best one has been made. */

int orientation ; /* Orientation of the output may be specified as
        LANDSCAPE or PORTRAIT. */

int write_al_chars_length ; // The maximum size of any alignment when written
        // out. Any excess is truncated with ellipses.

int write_section_chars_length ; // The maximum size of a section of
        // any alignment when written out.

double minimum_font_height ; /* The smallest font to be used in printing
        alignments. */

double maximum_font_height ; /* The largest font to be used in printing
        alignments. */

int n_grammars ; /* This is the number of grammars, over and obove the
        'naive' grammar, which provide a set of Old patterns after
        each New pattern has been processed. */

int create_patterns_diagnostic ; /* When ON, prints out diagnostic
        information for sequence::create_patterns() methods. Otherwise,
        no diagnostic information is printed. */

bool identification_symbols_marked ; // This is set by examining the symbols in the
        // corpus. If any of them have a '!' prefix, then IDENTIFICATION
        // symbols are set in the input by the use of this mark
        // and CONTENTS symbols are those without it.

/* End of Parameters. */

char *syntax_buffer ;
char *syn_buf_ptr ;

int context_number ;
int unique_id_number ;
int number_of_alignments_purged ;
int total_number_of_alignments ;
int number_of_alignments_retained_this_cycle ;
int new_hits ; /* Used in update_hit_structure() and in add_hit(). */
int driving_span ;
int target_span ;
int span ;
int number_of_parsing_alignments ;
int best_cycle_this_new_pattern ; // For a given pattern from New,
        // this is the cycle in which the best score for the
        // pattern was obtained.
int number_of_symbols_in_new, number_of_symbols_in_old ;
int number_of_patterns_in_new, original_number_of_patterns_in_old,
        final_number_of_patterns_in_old ;

group *brackets_list ; // Used in compare_patterns().
group *created_patterns ; // Used in sequence::created_new_patterns().
group *generalisation_list1 ; // Used in generalise_patterns().
group *generalisation_list2 ; // ditto.

const double log_10_2 = log10(2) ;

clock_t time1 ;
clock_t time2 ;
double old_symbol_bit_cost, adjusted_symbol_cost ;

sequence *best_alignment_this_new_pattern ; // The alignment giving
        // the best compression score in the current pattern from New.

/*** ARRAYS ETC USED IN WRITE_ALIGNMENT ***/

double font_heights[FONT_SET_SIZE] =
{7, 8, 9, 10, 11, 12, 14, 17.28, 20.74, 24.88} ;

struct key_entry key_array[KEY_ARRAY_SIZE] ; // Stores the key to the abbreviations of
```

```
// symbols in the input file.

int fe_key_array ;

const int max_alignment_depth = 50 ; // Determines the maximum
        // depth of an alignment, measured in rows. (Currently not
        // read in and not used).
const int max_alignment_length = 200 ; // Determines the maximum length of
        // an alignment, measured in symbols. (Currently not
        // read in but it is used).

const int write_alignment_char_rows = MAX_ALIGNMENT_DEPTH * 2 ; /* The maximum
        depth of an sequence that can be written (characters). */

char **write_al_chars ;
char **write_al_vacant_slots ;

struct row_entry row_array[MAX_ALIGNMENT_DEPTH] ; // Used to keep track of
        // symbols and positions for each row in
        // fill_write_al_chars.

char complete_symbol[MEDIUM_SCRATCH_ARRAY_SIZE],
        new_abbreviation[MEDIUM_SCRATCH_ARRAY_SIZE] ; // Used in
        // make_letter_abbreviations.

bool al1_rows[MEDIUM_SCRATCH_ARRAY_SIZE] ;

struct hit_entry
{
        hit_node *h_node ;
        symbol *newly_created_col ;
} hit_entries[MEDIUM_SCRATCH_ARRAY_SIZE] ; // Used in unify_best.

int fe_hit_entries ;

sequence *symbol_selection_array[MAX_NEW_ARRAY_ROWS][MCOLS] ; /* Each
        symbol corresponding to a symbol in New (in new_OAO_symbols[])
        receives zero or more pointers to sequences which contains that
        symbol. At the end of each cycle, The array is used for selection of
        target sequences for the following cycle. */

sequence *receptacle_pattern ;

hit_node **half_list ; // Used to store the worst 50% of the leaf nodes in the
        // leaf_array[].

int fe_half_list ;

struct hn_entry hit_node_array[HIT_NODE_ARRAY_SIZE] ;

int fe_hit_node_array ;

struct best_alignment_entry
{
        sequence *al1 ;
        group *contained_in ; /* List of the other best_alignments
        which contain 'alignment'. */
        double rel_P ;
        bool done ;
} best_alignments[MEDIUM_SCRATCH_ARRAY_SIZE] ; /* Stores a set of
        alignments, including best_alignment, which code the
        same symbols from New as best_alignment. It also stores
        the abs_P and rel_P values calculated for each of
        these alignments. Used in probabilities_of_inferences(). */

int fe_best_alignments ;

bool driving_brackets_array[MEDIUM_SCRATCH_ARRAY_SIZE] ;
bool target_brackets_array[MEDIUM_SCRATCH_ARRAY_SIZE] ;

struct names_entry
{
        char *old_name_number_chars ;
        int new_name_number ;
} *names_array ;

group *set_of_symbols ;

symbol *dr_array[MEDIUM_SCRATCH_ARRAY_SIZE], // Used in extract_patterns().
        *t_array[MEDIUM_SCRATCH_ARRAY_SIZE] ;

int fe_dr_array, fe_t_array ;
```

```
struct ref_struct
{
        sequence *target_pattern, *driving_pattern ;
        symbol *left_t_bracket, *t_class_symbol, *right_t_bracket,
                *left_dr_bracket, *dr_class_symbol, *right_dr_bracket ;
        list_element *t_el_pos, *dr_el_pos ;
        int left_t_bracket_int_pos, t_class_symbol_int_pos,
                right_t_bracket_int_pos, left_dr_bracket_int_pos,
                dr_class_symbol_int_pos, right_dr_bracket_int_pos ;
} reference_structure ;

/*** Mirror fields for hn_master ***/

int master_driving_int_pos ;
sequence *master_driving_pattern ;
symbol *master_driving_symbol ;
int master_target_int_pos ;
sequence *master_target_pattern ;
symbol *master_target_symbol ;

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks whether or not a character string is numeric.
**
** CALLING SEQUENCE:
**
**        bool isnumeric(char *s)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if the string is numeric, false otherwise.
**
**        s:                                The character string to be tested.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

bool isnumeric(char *s)
{
        char *c ;
        bool string_is_numeric = true ;

        for (c = s; *c != '\0'; c++)
        {
                if (isdigit(*c) == false)
                {
                        string_is_numeric = false ;
                        break ;
                }
        }

        return(string_is_numeric) ;

} // isnumeric

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Finds zero or more 'good' combinations of alignments in any given
**        cycle of the alignment-building process and adds them to
**        parsing_alignments.
**
**        The alignments that are combined are 'full' alignments in the sense
**        that all their Old CONTENTS symbols must be matched but the New
**        symbols are not necessarily all matched (if they were, then no
**        combination could be made as described next).
**
**        Alignments can be combined if, for any two alignments, none of the New
```

```
**        symbols between the first New symbol in alignment A and the last New
**        symbol in alignment A falls between the first New symbol in alignment
**        B and the last New symbol in alignment B - and vice versa. However,
**        there may be one or more unmatched New symbols between the last New
**        symbol of the left alignment and the first New symbol of the
**        right alignment.
**
**        Because the alignments in combine_al_array[] are in descending
**        order of compression difference, the simplifying assumption is
**        made that good combinations of alignments can be found by
**        adding alignments from successive entries in the array rather
**        than considering all possible combinations of alignments. Because
**        combinations are formed by adding alignments in this way, it
**        is assumed that combinations containing n + 1 alignments are
**        always better than combinations containing n alignments.
**
** CALLING SEQUENCE:
**
**        void combine_alignments(sequence *pattern1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        pattern1:                The pattern being parsed.
**
** IMPLICIT INPUTS:
**
**        alignments in parsing_alignments.
**
** IMPLICIT OUTPUTS:
**
**        Alignments added to parsing_alignments.
**
** SIDE EFFECTS:
**
**        NONE
*/

void combine_alignments(sequence *pattern1)
{
        struct combine_al_record
        {
                sequence *alignment ;
                int new_int_pos_first, new_int_pos_last ;
        } *combine_al_array ;

        int number_of_combinable_alignments ;
        group *current_set_of_combinations = new group ;

        sequence *al1 ;

        parsing_alignments->sort_by_compression_difference() ;

        // Compile information about each full (B) alignment that is needed
        // for the creation of combinations.

        number_of_combinable_alignments = 0 ;
        list_for(al1, sequence, parsing_alignments)
        {
                if (al1->get_degree_of_matching() != FULL_B) continue ;
                if (al1->get_row_pattern(0) != pattern1) continue ;
                number_of_combinable_alignments++ ;
        }

        int counter, new_int_pos, new_int_pos_first, new_int_pos_last ;
        symbol *symbol1, *symbol2 ;

        combine_al_array =
                new struct combine_al_record[number_of_combinable_alignments] ;

        counter = 0 ;
        list_for(al1, sequence, parsing_alignments)
        {
                if (al1->get_degree_of_matching() != FULL_B) continue ;
                if (al1->get_row_pattern(0) != pattern1) continue ;
                new_int_pos_first = NULL_VALUE ;
                combine_al_array[counter].alignment = al1 ;
                list_for(symbol1, symbol, al1)
                {
                        symbol2 = symbol1->get_row_symbol(0) ;
                        if (symbol2 == NIL) continue ;
                        new_int_pos = symbol1->get_row_orig_int_pos(0) ;
```

```
                if (new_int_pos_first == NULL_VALUE)
                        new_int_pos_first = new_int_pos ;
                new_int_pos_last = new_int_pos ;
        }
        combine_al_array[counter].new_int_pos_first = new_int_pos_first ;
        combine_al_array[counter].new_int_pos_last = new_int_pos_last ;
        counter++ ;
}

// Now combine alignments, in descending order of CD score and growing
// each combination as large as possible. Any alignment that does not
// fit into the current series is used to start a new series.

int i, j, length_pattern1 = pattern1->get_number_of_children() ;
combination *comb1, *comb2 ;
bool second_series_started ;
for (i = 0; i < number_of_combinable_alignments; i++)
{
        new_int_pos_first = combine_al_array[i].new_int_pos_first ;
        new_int_pos_last = combine_al_array[i].new_int_pos_last ;
        al1 = combine_al_array[i].alignment ;
        comb1 = new combination(length_pattern1) ;
        comb1->add_sub_alignment(al1,
                new_int_pos_first, new_int_pos_last, pattern1) ;
        second_series_started = false ;
        for (j = 0; j < number_of_combinable_alignments; j++)
        {
                if (j == i) continue ;
                new_int_pos_first = combine_al_array[j].new_int_pos_first ;
                new_int_pos_last = combine_al_array[j].new_int_pos_last ;
                al1 = combine_al_array[j].alignment ;
                if (comb1->can_accept(new_int_pos_first, new_int_pos_last))
                        comb1->add_sub_alignment(al1, new_int_pos_first,
                                                 new_int_pos_last, pattern1) ;
        }

        if (comb1->number_of_alignments() > 1)
        {
                // Check to make sure that the alignments in comb1 are not
                // simply a subset of any combinations in
                // current_set_of_combinations or set_of_combinations
                // (from previous cycles). If it is, delete comb1,
                // otherwise add it to current_set_of_combinations.

                list_for(comb2, combination, current_set_of_combinations)
                {
                        if (comb1->is_subset_of(comb2))
                        {
                                #if DIAGNOSTIC5

                                fprintf(output_file, "%s%d%s%d%s",
                                        "Combination C_ID",
                                        comb1->get_C_ID(),
                                        " is a subset of, or the same as, combination C_ID",
                                        comb2->get_C_ID(),
                                        " and is deleted.\n\n") ;

                                #endif

                                delete comb1 ;
                                goto L1 ;
                        }
                }

                list_for(comb2, combination, set_of_combinations)
                {
                        if (comb1->is_subset_of(comb2))
                        {
                                #if DIAGNOSTIC5

                                fprintf(output_file, "%s%d%s%d%s",
                                        "Combination C_ID",
                                        comb1->get_C_ID(),
                                        " is a subset of, or the same as, combination C_ID",
                                        comb2->get_C_ID(),
                                        " and is deleted.\n\n") ;

                                #endif

                                delete comb1 ;
                                goto L1 ;
                        }
                }
```

```
                current_set_of_combinations->add_child(comb1) ;
        }
        else
        {
                #if DIAGNOSTIC5

                fprintf(output_file, "%s%d%s",
                        "Combination C_ID",
                        comb1->get_C_ID(),
                        " contains only one alignment and is deleted.\n\n") ;

                #endif

                delete comb1 ;
        }
        L1: ;
}

// Find the best combinations up to combination_limit, print them
// out and make them into composite alignments. At the same time,
// add the combination to set_of_combinations.

fprintf(output_file, "Start of combining alignments.\n(") ;
print_pattern_cycle(true, pattern1) ;
fprintf(output_file, ")\n\n") ;

double best_score, temp_score ;
combination *best_comb ;
int total_number_of_combinations =
        current_set_of_combinations->count_number_of_children() ;

fprintf(output_file, "%s%d%s",
        "Total number of combinations = ",
        total_number_of_combinations,
        "\n\n") ;

sequence *composite_alignment ;
counter = 0 ;
while (counter < combination_limit)
{
        best_score = NULL_VALUE ;
        list_for(comb1, combination, current_set_of_combinations)
        {
                temp_score = comb1->get_combination_score() ;
                if (temp_score > best_score)
                {
                        best_comb = comb1 ;
                        best_score = temp_score ;
                }
        }

        if (best_score > NULL_VALUE)
        {
                current_set_of_combinations->extract_child(best_comb) ;
                best_comb->print_combination() ;
                composite_alignment =
                        best_comb->make_composite_alignment(pattern1) ;
                composite_alignment->find_degree_of_matching(pattern1) ;
                parsing_alignments->add_child(composite_alignment) ;
                set_of_combinations->add_child(best_comb) ;
        }
        else break ;
        counter++ ;
}

fprintf(output_file, "End of combining alignments.\n(") ;
print_pattern_cycle(true, pattern1) ;
fprintf(output_file, ")\n\n") ;

// Delete arrays and temporary objects.

current_set_of_combinations->delete_children() ;
delete current_set_of_combinations ;
delete[] combine_al_array ;

} // combine_alignments

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
```

```
**        This function deletes all the Old patterns and replaces them
**        with the patterns in the best n_grammars.
**
** CALLING SEQUENCE:
**
**        void purge_old_patterns(int n_grammars, sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        n_grammars:                The number of grammars to be preserved within the
**                                        set of Old patterns.
**        cnp:                        Current New pattern.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void purge_old_patterns(int n_grammars, sequence *cnp)
{
        fprintf(output_file, "Start of purge_old_patterns()\n\n") ;

        if (set_of_grammars->is_empty())
        {
                fprintf(output_file,
                        "There is no best grammar so no patterns to be purged.\n\n") ;
                return ;
        }

        int counter = 0 ;
        grammar *grammar1,
                *old_grammar = new grammar ; // This is a temporary vehicle
                // for the patterns in a new version of old_patterns. The
                // reason a grammar is used is that this allows access to
                // the methods clean_up(), tidy_up_code_symbols() and
                // merge_patterns().
        sequence *pattern1 ;
        bool pattern_added ;

        list_for(grammar1, grammar, set_of_grammars)
        {
                for (pattern1 = grammar1->get_first_pattern();
                        pattern1 != NIL; pattern1 =
                                grammar1->get_next_pattern())
                {
                        // pattern2 = new sequence(*pattern1) ;
                        pattern_added = old_grammar->add_pattern(pattern1) ;
                        // if (pattern_added == false)
                        //        delete pattern2 ;
                }
                if (++counter >= n_grammars) break ;
        }

        // Add patterns from naive grammar.

        for (pattern1 = naive_grammar->get_first_pattern();
                pattern1 != NIL; pattern1 =
                        naive_grammar->get_next_pattern())
        {
                pattern_added = old_grammar->add_pattern(pattern1) ;
        }

        fprintf(output_file, "%s%d%s%d%s",
                "New grammar created by combining of best ",
                n_grammars,
                " original grammars and naive grammar is GR",
                old_grammar->get_ID(),
                "\n\n") ;

        old_grammar->write_grammar(true, false) ;

        // Now clean up unnecessary code symbols, [merge patterns where
        // possible] and [renumber code symbols in a tidy manner].
```

```
        old_grammar->clean_up(NULL_VALUE) ;

        // old_grammar->merge_patterns() ;

        // old_grammar->tidy_up_code_symbols() ;

        // Now transfer the patterns in old_grammar to
        // the (empty) old_patterns. All the patterns that are
        // currently in old_patterns are removed and those that
        // are not in old_grammar are deleted.

        while (pattern1 = (sequence *)old_patterns->extract_first_child())
        {
                if (old_grammar->contains(pattern1) == false)
                        delete pattern1 ;
        }

        while (pattern1 = old_grammar->extract_first_pattern())
                old_patterns->add_child(pattern1) ;

        fprintf(output_file, "%s%d%s",
                "OLD PATTERNS DERIVED FROM THE BEST ",
                n_grammars,
                " GRAMMARS AT THE END OF ") ;
                print_pattern_cycle(false, cnp) ;
        fprintf(output_file, ":\n\n") ;

        if (old_patterns->is_empty())
                fprintf(output_file, "old_patterns is empty.\n\n") ;
        else
        {
                list_for(pattern1, sequence, old_patterns)
                {
                        pattern1->print_ID() ;
                        fprintf(output_file, " ") ;
                        pattern1->write_tree_object(PRINT_SEQUENCE_FREQUENCY) ;
                }
        }

        fprintf(output_file, "End of purge_old_patterns()\n\n") ;
} // purge_old_patterns

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**       Compares the list of IDENTIFICATION symbols in two patterns (excluding
**       bracket symbols) and returns values to show whether the list is
**       the same in the two patterns, or the first list is a subset
**       of the other, or vice versa, or neither list is a subset of the other.
**
** CALLING SEQUENCE:
**
**       int compare_classes(sequence *pattern1, int p1_classes,
**               sequence *pattern2, int p2_classes)
**
** FORMAL ARGUMENTS:
**
**       Return value:        A_DIFFERENT_FROM_B if the pattern1 list of
**                                  IDENTIFICATION symbols is not a subset of the
**                                  pattern2 list and vice versa, A_SUBSET_B if the
**                                  pattern1 list is a subset of the pattern2 list,
**                                  B_SUBSET_A if the pattern2 list is a subset of
**                                  the pattern1 list, and A_SAME_AS_B if the two
**                                  lists are the same (not necessarily in the same
**                                  order).
**       pattern1:               One of the two patterns.
**       p1_classes:               The number of non-bracket IDENTIFICATION symbols
**                                  in pattern1.
**       pattern2:               The second of the two patterns.
**       p2_classes:               The number of non-bracket IDENTIFICATION symbols
**                                  in pattern2.
**
** IMPLICIT INPUTS:
**
**       NONE
**
** IMPLICIT OUTPUTS:
**
**       NONE
**
```

```
** SIDE EFFECTS:
**
**        NONE
*/

int compare_classes(sequence *pattern1, int p1_classes,
        sequence *pattern2, int p2_classes)
{
        symbol *symbol1, *symbol2 ;
        int symbol_type, fe_class_array = 0, i ;
        struct class_entry
        {
                symbol *class1, *class2 ;
        } class_array[MEDIUM_SCRATCH_ARRAY_SIZE] ;

        if (pattern1 == pattern2)
                abort_run("Anomalous input in compare_classes()") ;

        list_for(symbol1, symbol, pattern1)
        {
                symbol_type = symbol1->get_type() ;
                if (symbol_type == LEFT_BRACKET) continue ;
                if (symbol_type == CONTENTS) break ;

                for(i = 0; i < fe_class_array; i++)
                        if (symbol1 == class_array[i].class1)
                                goto L1 ;

                list_for(symbol2, symbol, pattern2)
                {
                        symbol_type = symbol2->get_type() ;
                        if (symbol_type == LEFT_BRACKET) continue ;
                        if (symbol_type == CONTENTS) break ;

                        for(i = 0; i < fe_class_array; i++)
                                if (symbol2 == class_array[i].class2)
                                        goto L2 ;

                        if (symbol1->name_matches(symbol2))
                        {
                                class_array[fe_class_array].class1 = symbol1 ;
                                class_array[fe_class_array].class2 = symbol2 ;
                                plus_one(&fe_class_array, MEDIUM_SCRATCH_ARRAY_SIZE,
                                        "Overflow of class_array[] in compare_classes().") ;
                                break ;
                        }

                        L2: ;
                }

                L1: ;
        }

        if (p1_classes == fe_class_array)
        {
                if (p2_classes == fe_class_array)
                        return(A_SAME_AS_B) ;
                else if (p2_classes > fe_class_array)
                        return(A_SUBSET_B) ;
        }
        else if (p2_classes == fe_class_array)
        {
                if (p1_classes > fe_class_array)
                        return(B_SUBSET_A) ;
        }
        else return(A_DIFFERENT_FROM_B) ;

} // compare_classes

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks whether two patterns belong in the same class.
**
** CALLING SEQUENCE:
**
**        symbol *same_class(sequence *pattern1, sequence *pattern2)
**
** FORMAL ARGUMENTS:
**
**        Return value:        NIL if they are not in the same class, otherwise
```

```
**                                      the class symbol of the shared class is returned.
**
**        pattern1:               One of the two patterns to be assessed.
**        pattern2:               The other pattern.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

symbol *same_class(sequence *pattern1, sequence *pattern2)
{
        symbol *symbol1, *symbol2 ;

        list_for(symbol1, symbol, pattern1)
        {
                if (symbol1->get_type() == LEFT_BRACKET) continue ;
                if (symbol1->get_status() == CONTENTS) break ;
                list_for(symbol2, symbol, pattern2)
                {
                        if (symbol2->get_type() == LEFT_BRACKET) continue ;
                        if (symbol2->get_status() == CONTENTS) break ;

                        if (symbol1->name_matches(symbol2)) return(symbol1) ;
                }
        }
        return(NIL) ;
} // same_class

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Makes a new left or right bracket symbol.
**
** CALLING SEQUENCE:
**
**        symbol *make_bracket_symbol(int type, int status, al_depth)
**
** FORMAL ARGUMENTS:
**
**        Return value:        A new bracket symbol.
**
**        type:                        LEFT_BRACKET or RIGHT_BRACKET.
**        status:                        BOUNDARY_MARKER or CONTENTS.
**        al_depth:               The number of rows.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

symbol *make_bracket_symbol(int type, int status, int al_depth)
{
        symbol *symbol1 = new symbol(NIL, al_depth, NULL_VALUE) ;
        symbol1->set_type(type) ;
        if (type == LEFT_BRACKET) symbol1->set_name("<") ;
        else if (type == RIGHT_BRACKET) symbol1->set_name(">") ;
        else abort_run("Invalid type for make_bracket_symbol().") ;
        symbol1->set_status(status) ;
        symbol1->assign_frequency_and_cost() ;
        symbol1->set_is_a_hit(false) ;

        return(symbol1) ;
} // make_bracket_symbol
```

```
/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Increments and index by 1, checks that it does not exceed a limit
**        and aborts the program, printing out a message if the limit is
**        exceeded.
**
** CALLING SEQUENCE:
**
**        void increment_index(int *index, int limit, char *message)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        index:                        A pointer to the index to be incremented.
**        limit:                        The limit to be checked.
**        message:               The message to be printed out.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void increment_index(int *index, int limit, char *message)
{
        (*index)++ ;
        if ((*index) >= limit)
                abort_run(message) ;

} // increment_index

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Prints the number of the pattern from New and, within that pattern,
**        the cycle.
**
** CALLING SEQUENCE:
**
**        void print_pattern_cycle(bool print_cycle, sequence *pattern1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        print_cycle:        If true, print the current value of cycle,
**                                    otherwise don't.
**        pattern1:                The pattern currently being parsed.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void print_pattern_cycle(bool print_cycle, sequence *pattern1)
{
        if (print_cycle == false)
                fprintf(output_file, "%s%d%s%d",
                        "PATTERN ID",
```

```
                        pattern1->get_ID(),
                        ", PHASE ",
                        phase) ;
        else
        {
                fprintf(output_file, "%s%d",
                        "PATTERN ID",
                        pattern1->get_ID()) ;
                if (phase == 2)
                        fprintf(output_file, "%s%d%s%d%s%d",
                                " (ID",
                                current_new_pattern->get_ID(),
                                "), CYCLE ",
                                cycle,
                                ", PHASE ",
                                phase) ;
                else fprintf(output_file, "%s%d%s%d",
                                ", CYCLE ",
                                cycle,
                                ", PHASE ",
                                phase) ;
        }
        fflush(output_file) ;
} // print_pattern_cycle

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Makes a new column for insertion in new_alignment
**        (in unify_best).
**
** CALLING SEQUENCE:
**
**        symbol *make_column(int new_alignment_depth,
**                        sequence *driving_pattern, symbol *driving_symbol,
**                        sequence *target_pattern, symbol *target_symbol,
**                        int target_pattern_row_in_new_alignment, int hn_ID,
**                        bool hit_this_cycle)
**
** FORMAL ARGUMENTS:
**
**        Return value:                       A pointer to the newly-created column. Value
**                                                    is NIL if no column is created.
**
**        new_alignment_depth:        The depth of the new alignment.
**        driving_pattern:                The driving pattern.
**        driving_symbol:                 The driving symbol containing information to be
**                                                    copied into the new column.
**        driving_int_pos:                The int position of the driving_symbol in
**                                                    the driving pattern.
**        target_pattern:                 The target pattern.
**        target_symbol:                        The target symbol containing information to be
**                                                    copied into the new column.
**        target_int_pos:                 The int position of the target_symbol in
**                                                    the target pattern.
**        target_pattern_row_in_new_alignment: The row of the new column into which the
**                                                    first row of the target symbol should be copied.
**        hn_ID:                                     The ID of the hit node (corresponding to this
**                                                    new column. NULL_VALUE if there is no hit node.
**        hit_this_cycle:                 true if the column records a hit on this cycle,
**                                                    false otherwise.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

symbol *make_column(int new_alignment_depth,
        sequence *driving_pattern, symbol *driving_symbol,
        sequence *target_pattern, symbol *target_symbol,
        int target_pattern_row_in_new_alignment, int hn_ID,
        bool hit_this_cycle)
```

```
{
        symbol *new_al_col1, *symbol1, *driving_new_symbol ;
        alignment_element *al_el1, *al_el2 ;
        int driving_pattern_depth, target_pattern_depth, i ;

        if (driving_symbol != NIL) symbol1 = driving_symbol ;
        else if (target_symbol != NIL) symbol1 = target_symbol ;
        else abort_run("driving_symbol and target_symbol are both NIL in \
                make_column") ;

        new_al_col1 = new symbol(symbol1->get_name(),
                new_alignment_depth, hn_ID) ;

        if (hit_this_cycle) new_al_col1->set_is_a_hit(true) ;

        new_al_col1->set_bit_cost(symbol1->get_bit_cost()) ;
        new_al_col1->set_type(symbol1->get_type()) ;

        if (driving_symbol != NIL)
        {
                driving_pattern_depth = driving_pattern->get_sequence_depth() ;
                if (driving_pattern_depth == 1) // Driving pattern is New
                {
                        al_el1 = new_al_col1->get_al_el(0) ;
                        al_el1->set_el_obj(driving_symbol) ;
                        al_el1->set_original_pattern(driving_pattern) ;
                }
                else for (i = 0; i < driving_pattern_depth; i++)
                {
                        al_el1 = new_al_col1->get_al_el(i) ;
                        al_el2 = driving_symbol->get_al_el(i) ;
                        *al_el1 = *al_el2 ;
                }
                if (driving_symbol->is_a_hit())
                        new_al_col1->set_is_a_hit(true) ;
        }

        if (target_symbol != NIL)
        {
                target_pattern_depth = target_pattern->get_sequence_depth() ;
                if (target_pattern_depth == 1) // Target pattern is one of
                        // original patterns in Old
                {
                        al_el1 = new_al_col1->
                                get_al_el(target_pattern_row_in_new_alignment) ;
                        driving_new_symbol = (symbol *)al_el1->get_el_obj() ;
                        if (driving_new_symbol != NIL &&
                                driving_new_symbol != target_symbol)
                                abort_run("Anomaly in make_column \
                                (target_pattern_depth == 1)") ;
                        al_el1->set_el_obj(target_symbol) ;
                        al_el1->set_original_pattern(target_pattern) ;
                }
        }

        // Now put the correct values in for
        // same_column_above and same_column_below.

        new_al_col1->set_symbol_matches() ;

        return(new_al_col1) ;
} // make_column

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Prints the current time.
**
** CALLING SEQUENCE:
**
**        void time_now()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        From the computer clock.
**
** IMPLICIT OUTPUTS:
```

```
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void time_now()
{
        struct tm *time_now ;
        time_t secs_now ;
        char str[80] ;

        tzset() ;
        time(&secs_now) ;
        time_now = localtime(&secs_now) ;
        strftime(str, 80,
                "Time: %H:%M, %A, %B %d, 20%y",
                time_now);
        fprintf(output_file, "%s\n\n", str);
} // time_now

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Prints the year.
**
** CALLING SEQUENCE:
**
**        void year()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        From the computer clock.
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void year()
{
        struct tm *time_now ;
        time_t secs_now ;
        char str[20] ;

        tzset() ;
        time(&secs_now) ;
        time_now = localtime(&secs_now) ;
        strftime(str, 20,
                "20%y",
                time_now);
        fprintf(output_file, "%s", str);
} // time_now

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Prepares the ground for unify_best by copying the nodes of
**        a hit sequence into hit_entries[]. This means that it is easy
**        to step through the nodes in left-to-right order - which is
**        more intuitive than always using them in reverse order.
**
** CALLING SEQUENCE:
**
**        void prepare_hit_sequence(hit_node *leaf_node)
**
** FORMAL ARGUMENTS:
```

```
**
**        Return value:        void
**
**        leaf_node:        The leaf node of the hit sequence in the hit structure.
**
** IMPLICIT INPUTS:
**
**        The hit structure
**
** IMPLICIT OUTPUTS:
**
**        hit_entries[]
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void prepare_hit_sequence(hit_node *leaf_node)
{
        hit_node *h_node, *left_h_node ;
        int node_position ;

        // First count the number of nodes in the hit sequence
        // so that a check can be made that it does not exceed the
        // size of hit_entries[] and to find the correct place
        // in hit_entries[] to begin loading the nodes.

        node_position = -1 ;
        for (h_node = leaf_node; h_node != hit_root;
                h_node = h_node->get_hn_parent())
                        node_position++ ;

        fe_hit_entries = node_position + 1 ;
        if (fe_hit_entries >= MEDIUM_SCRATCH_ARRAY_SIZE)
                abort_run("Chain of hit_nodes is too long in unify_best") ;

        for (left_h_node = leaf_node; left_h_node != hit_root;
                left_h_node = left_h_node->get_hn_parent())
                        hit_entries[node_position--].h_node = left_h_node ;
} // prepare_hit_sequence

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks for mismatches in a hit sequence (in unify_best).
**
** CALLING SEQUENCE:
**
**        bool mismatch_found(hit_node *leaf_node, sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if there are mismatches, false otherwise.
**
**        leaf_node:                The last node of the hit structure sequence being
**                                        processed.
**        cnp:                        The New pattern being processed.
**
** IMPLICIT INPUTS:
**
**        hit_entries[] and fe_hit_entries
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

bool mismatch_found(hit_node *leaf_node, sequence *cnp)
{
        hit_node *h_node, *left_h_node, *right_h_node ;
        sequence *driving_pattern, *target_pattern ;
        symbol *last_driving_symbol, *last_target_symbol ;

        // Unless the driving pattern is the same as New, a 'mis-match'
```

```
        // occurs if, at any point in the alignment between one hit (for the
        // the current cycle) and the next hit (for the current cycle), there
        // are one or more non-hit symbols from the driving pattern AND
        // one or more non-hit symbols from the target pattern. If there
        // is any mis-match in an alignment, it is not valid and should
        // not be formed.

        h_node = leaf_node ;
        driving_pattern = h_node->get_driving_pattern() ;
        if (driving_pattern == cnp) return(false) ;

        // The driving_pattern is not the same as New.

        // Check for mis-matches *after* the last hit (for the current cycle).

        target_pattern = h_node->get_target_pattern() ;
        last_driving_symbol = (symbol *)
                driving_pattern->get_last_child() ;
        last_target_symbol = (symbol *)
                target_pattern->get_last_child() ;
        if (last_driving_symbol != h_node->get_driving_symbol()
                && last_target_symbol != h_node->get_target_symbol())
                return(true) ;

        // Check for mis-matches between the first and last hits.

        left_h_node = NIL ;
        for (int i = 0; i < fe_hit_entries; i++)
        {
                right_h_node = hit_entries[i].h_node ;
                if (left_h_node == NIL)
                {
                        left_h_node = right_h_node ;
                        continue ;
                }
                if (right_h_node->get_driving_int_pos() -
                        left_h_node->get_driving_int_pos() > 1
                        && right_h_node->get_target_int_pos() -
                        left_h_node->get_target_int_pos() > 1)
                        return(true) ;
                left_h_node = right_h_node ;
        }

        // Now check for mis-matches before the first hit.

        h_node = hit_entries[0].h_node ;
        if (h_node->get_driving_int_pos() > 0
                && h_node->get_target_int_pos() > 0)
                return(true) ;

        // No mismatch has been detected.

        return(false) ;
} // mismatch_found

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks a newly-constructed alignment to make sure that symbols from
**        cnp are in their correct order.
**
** CALLING SEQUENCE:
**
**        bool check_ordering_of_new(sequence *new_alignment, sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if the symbols from cnp
**                                    are in their correct order, false otherwise.
**
**        new_alignment:               The alignment to be checked.
**        cnp:                        The New pattern being processed.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
```

```
** SIDE EFFECTS:
**
**        NONE
**
*/

bool check_ordering_of_new(sequence *new_alignment, sequence *cnp)
{
        int new_symbol_int_pos1 = NULL_VALUE, new_symbol_int_pos2 ;
        symbol *symbol1 ;

        for (symbol1 = (symbol *)new_alignment->get_first_child();
                symbol1 != NIL ; symbol1 = (symbol *)
                        new_alignment->get_next_child())
        {
                new_symbol_int_pos2 = symbol1->get_row_orig_int_pos(0) ;
                if (new_symbol_int_pos2 != NULL_VALUE)
                {
                        if (new_symbol_int_pos1 != NULL_VALUE)
                        {
                                if (new_symbol_int_pos2 <= new_symbol_int_pos1)
                                        return(false) ;
                        }
                        new_symbol_int_pos1 = new_symbol_int_pos2 ;
                }
        }

        return(true) ;
} // check_ordering_of_new

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks that gaps between hit symbols in a newly-formed alignment
**        conform to relevant parameters.
**
** CALLING SEQUENCE:
**
**        bool al_gaps_OK(sequence *al1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if the gaps are within limits, false otherwise.
**
**        al1:                        The alignment to be tested.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

bool al_gaps_OK(sequence *al1)
{
        symbol *al_col1 ;
        int previous_new_hit_int_pos = NULL_VALUE, this_new_hit_int_pos ;

        // Test gaps between hit symbols in New.

        al1->initialise() ;
        while (al_col1 = (symbol *)al1->get_next_child())
        {
                this_new_hit_int_pos = al_col1->get_row_orig_int_pos(0) ;

                if (this_new_hit_int_pos == NULL_VALUE)
                {
                        previous_new_hit_int_pos = this_new_hit_int_pos ;
                        continue ;
                }
                if (previous_new_hit_int_pos == NULL_VALUE) continue ;

                // if ((this_new_hit_int_pos - previous_new_hit_int_pos - 1) >
                //        max_new_gap)
```

```
                //        return(false) ;

                previous_new_hit_int_pos = this_new_hit_int_pos ;
        }
        return(true) ;
} // al_gaps_OK

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Forms an alignment from a hit sequence. This function is called
**        from unify_best.
**
** CALLING SEQUENCE:
**
**        sequence *make_one_alignment(hit_node *leaf_node, sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:        The alignment which is formed. NIL if, for some reason,
**                                    the hit sequence proves to be invalid.
**
**        leaf_node:                The leaf node of the hit sequence from which the new
**                                    alignment is to be formed.
**        cnp:                        The New pattern being processed.
**
** IMPLICIT INPUTS:
**
**        hit_entries[], fe_hit_entries
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

sequence *make_one_alignment(hit_node *leaf_node, sequence *cnp)
{
        sequence *driving_pattern, *target_pattern, *temp_patt, *new_alignment ;
        symbol *newly_created_col, *driving_symbol, *target_symbol,
                *hit_target_symbol, *newly_created_hit_col, *symbol1 ;
        int driving_pattern_depth,
                new_alignment_depth, hit_node_index,
                target_pattern_row_in_new_alignment,
                driving_int_pos, target_int_pos, row1 ;
        hit_node *h_node ;
        alignment_element *al_el1, *al_el2 ;

        // 1 CALCULATE THE DEPTH OF THE NEW ALIGNMENT AS THE SUM OF
        // OF THE DEPTHS OF THE DRIVING PATTERN AND THE TARGET PATTERN.
        // IT IS ASSUMED THAT THE TARGET PATTERN IS ALWAYS AN OLD PATTERN,
        // WITH A DEPTH OF 1.

        int start_of_new_target_pattern = NULL_VALUE ;
        driving_pattern = leaf_node->get_driving_pattern() ;
        target_pattern = leaf_node->get_target_pattern() ;
        driving_pattern_depth = driving_pattern->get_sequence_depth() ;

        new_alignment_depth = driving_pattern_depth + 1 ;

        if (new_alignment_depth >= MAX_ALIGNMENT_DEPTH)
                abort_run("Too many rows in alignment in unify_best") ;

        // Determine the row position of the target pattern in the
        // new alignment.

        target_pattern_row_in_new_alignment = driving_pattern_depth ;

        // 2 CREATE AN OBJECT FOR A NEW ALIGNMENT.

        new_alignment = new sequence ;
        new_alignment->set_driving_ID(driving_pattern->get_ID()) ;
        new_alignment->set_target_ID(target_pattern->get_ID()) ;

        // 3 IF THE DRIVING PATTERN IS NOT THE SAME AS NEW, COPY IN
        // ALL THE COLUMNS FOR THE DRIVING PATTERN BUT
        // USING COLUMNS WITH A LENGTH OF AL_CONSTR_DEPTH. IN hit_entries[]
```

```
// KEEP A RECORD OF WHICH OF THE NEWLY-CREATED COLUMNS REPRESENTS
// A HIT.

// IF THE DRIVING PATTERN IS THE SAME AS NEW,
// DO THE SAME BUT ONLY FOR THOSE SYMBOLS IN THE DRIVING_PATTERN
// THAT FORM HITS WITH THE TARGET PATTERN.

hit_node_index = 0 ;
if (driving_pattern == cnp)
{
        // The driving pattern is the same as New. Only make
        // columns for those symbols in the driving pattern which
        // form hits with the target pattern.

        for (hit_node_index = 0; hit_node_index < fe_hit_entries;
                hit_node_index++)
        {
                h_node = hit_entries[hit_node_index].h_node ;
                driving_symbol = h_node->get_driving_symbol() ;
                driving_pattern = h_node->get_driving_pattern() ;
                driving_int_pos = driving_pattern->
                        get_int_pos_by_child(driving_symbol) ;
                target_symbol = h_node->get_target_symbol() ;
                target_pattern = h_node->get_target_pattern() ;
                target_int_pos = target_pattern->
                        get_int_pos_by_child(target_symbol) ;
                newly_created_col = make_column(new_alignment_depth,
                        driving_pattern, driving_symbol,
                        target_pattern, target_symbol,
                        target_pattern_row_in_new_alignment,
                        h_node->get_ID(), true) ;

                new_alignment->add_child(newly_created_col) ;

                hit_entries[hit_node_index].newly_created_col =
                        newly_created_col ;
        }
}
else
{
        // The driving pattern is not the same as New.

        driving_pattern->initialise() ;
        while (driving_symbol = (symbol *)
                driving_pattern->get_next_child())
        {
                driving_int_pos = driving_pattern->
                        get_int_pos_by_child(driving_symbol) ;

                // Note: the last field below (for whether or not
                // this column contains a hit) is set to false
                // as a default. The value is set properly a few
                // lines down.

                newly_created_col = make_column(new_alignment_depth,
                        driving_pattern, driving_symbol,
                        target_pattern, NIL,
                        target_pattern_row_in_new_alignment,
                        NULL_VALUE, false) ;

                new_alignment->add_child(newly_created_col) ;

                // Now check to see whether this driving symbol is
                // part of a hit. If it is, make a note in hit_entries
                // to show the correspondence between this hit
                // and newly_created_col.

                if (hit_node_index >= fe_hit_entries) continue ;
                h_node = hit_entries[hit_node_index].h_node ;
                if (driving_symbol == h_node->get_driving_symbol())
                {
                        hit_entries[hit_node_index].newly_created_col =
                                newly_created_col ;
                        newly_created_col->
                                set_h_node_ID(h_node->get_ID()) ;
                        newly_created_col->
                                set_is_a_hit(true) ;
                        hit_node_index++ ;
                }
        }
}

// 4 STEP THROUGH THE HIT SEQUENCE FILLING IN THE DETAILS OF
```

```
// THE TARGET SYMBOL IN EACH COLUMN REPRESENTING A HIT.

symbol *last_column_added ; // This is needed to determine the
        // the correct positioning of trailing unmatched target symbols,
        // if any.
for (hit_node_index = 0; hit_node_index < fe_hit_entries;
hit_node_index++)
{
        h_node = hit_entries[hit_node_index].h_node ;
        newly_created_col =
                hit_entries[hit_node_index].newly_created_col ;

        target_symbol = h_node->get_target_symbol() ;

        target_pattern = h_node->get_target_pattern() ;
        target_int_pos = h_node->get_target_int_pos() ;
        al_el2 = newly_created_col->
                get_al_el(target_pattern_row_in_new_alignment) ;
        al_el2->set_el_obj(target_symbol) ;
        al_el2->set_original_pattern(target_pattern) ;
        last_column_added = newly_created_col ;
}

// 5 STEP THROUGH THE TARGET PATTERN AND THE TABLE OF HIT_ENTRIES[],
// INSERTING NEW COLUMNS IN new_alignment CORRESPONDING TO SYMBOLS
// IN THE TARGET PATTERN WHICH HAVE NOT FORMED HITS
// WITH THE DRIVING PATTERN.

target_symbol = (symbol *)target_pattern->get_first_child() ;
for (hit_node_index = 0; hit_node_index < fe_hit_entries;
        hit_node_index++)
{
        h_node = hit_entries[hit_node_index].h_node ;
        newly_created_hit_col =
                hit_entries[hit_node_index].newly_created_col ;
        hit_target_symbol = h_node->get_target_symbol() ;
        while (target_symbol != hit_target_symbol)
        {
                newly_created_col = make_column(new_alignment_depth,
                        driving_pattern, NIL,
                        target_pattern, target_symbol,
                        target_pattern_row_in_new_alignment,
                        NULL_VALUE, false) ;
                new_alignment->precede(newly_created_col,
                        newly_created_hit_col) ;
                target_symbol = (symbol *)
                        target_pattern->get_next_child() ;
        }

        // Step past the target symbol which corresponds to the
        // current hit_target_symbol and get the next target
        // symbol.

        target_symbol = (symbol *)target_pattern->get_next_child() ;
}

// 6 DO TAIL OF THE TARGET PATTERN (IF ANY).

while (target_symbol != NIL)
{
        newly_created_col = make_column(new_alignment_depth,
                driving_pattern, NIL,
                target_pattern, target_symbol,
                target_pattern_row_in_new_alignment, NULL_VALUE,
                false) ;
        new_alignment->follow(last_column_added, newly_created_col) ;
        last_column_added = newly_created_col ;
        target_symbol = (symbol *)target_pattern->get_next_child() ;
}

// Check for anomalies in the ordering of symbols in
// cnp within the alignment.

if (check_ordering_of_new(new_alignment, cnp) == false)
{
        delete new_alignment ;
        new_alignment = NIL ;
        if (verbose)
        {
                fprintf(output_file,
                        "Anomalous ordering of symbols from New in hit sequence ") ;
                leaf_node->print_ID() ;
                fprintf(output_file, ". No new alignment is formed.\n\n") ;
```

```
                }
                return(NIL) ;
        }

        // Check that gaps between hits conform with relevant parameters.

        if (al_gaps_OK(new_alignment) == false)
        {
                delete new_alignment ;
                new_alignment = NIL ;
                if (verbose)
                {
                        fprintf(output_file, "Gaps too large in hit sequence ") ;
                        leaf_node->print_ID() ;
                        fprintf(output_file, ". Alignment is discarded.\n\n") ;
                }
                return(NIL) ;
        }

        new_alignment->set_sequence_depth(new_alignment_depth) ;
        new_alignment->set_leaf_node_ID(leaf_node->get_ID()) ;

        // 7 MAKE SURE THAT EVERY CELL OF EACH ROW OF THE ALIGNMENT CONTAINS
        // A REFERENCE TO THE PATTERN FOR THAT ROW. NB This information
        // is needed in compute_score_with_gaps (later), hence it needs to be
        // done before compute_score_with_gaps is called.

        for (row1 = 0; row1 < new_alignment_depth; row1++)
        {
                // First find the first reference to the pattern for
                // this row.

                new_alignment->initialise() ;
                while (symbol1 = (symbol *)
                        new_alignment->get_next_child())
                {
                        temp_patt = symbol1->get_row_pattern(row1) ;
                        if (temp_patt != NIL) break ;
                }

                // Now put the value of temp_patt in every cell for this row.

                new_alignment->initialise() ;
                while (symbol1 = (symbol *)
                        new_alignment->get_next_child())
                {
                        al_el1 = symbol1->get_al_el(row1) ;
                        al_el1->set_original_pattern(temp_patt) ;
                }
        }

        // 8 MARK POSITIONS OF SYMBOLS IN INTEGERS AND SHOW PARENT
        // OF EACH SYMBOL

        new_alignment->mark_parent_and_int_positions_non_recursive() ;

        // 9 MARK STATUS OF SYMBOLS

        list_element *pos1 = NIL ;
        symbol *first_symbol = (symbol *)new_alignment->get_first_child(),
                *last_symbol = (symbol *)new_alignment->get_last_child(),
                *symbol2 ;
        list_for_el_pos(symbol1, symbol, new_alignment, pos1)
        {
                if (symbol1->is_a_hit())
                        symbol1->set_status(CONTENTS) ;
                else
                {
                        symbol2 = symbol1->find_unmatched_symbol() ;
                        if (symbol2 == NIL)
                                abort_run("Anomaly in make_one_alignment().") ;
                        symbol1->set_status(symbol2->get_status()) ;
                }
        }

        new_alignment->set_symbol_matches() ;

        return(new_alignment) ;
} // make_one_alignment

/*********************************************************************/

/*
```

```
** FUNCTIONAL DESCRIPTION:
**
**        Using a hit sequence recorded in hit_entries[], write the
**        corresponding alignment horizontally.
**
** CALLING SEQUENCE:
**
**        void write_hit_sequence_horizontal()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        hit_entries[]
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void write_hit_sequence_horizontal()
{
        hit_node *leaf_node = hit_entries[fe_hit_entries - 1].h_node,
                *h_node ;
        int hit_index, char_row, char_col ;
        char character_array[3][LARGE_SCRATCH_ARRAY_SIZE] ;
        sequence *driving_pattern = leaf_node->get_driving_pattern(),
                *target_pattern = leaf_node->get_target_pattern() ;
        symbol *d_symbol, *hit_d_symbol, *t_symbol, *hit_t_symbol ;

        // Clear character_array[][].

        for (char_row = 0; char_row < 3; char_row++)
                for (char_col = 0; char_col < LARGE_SCRATCH_ARRAY_SIZE;
                        char_col++)
                                character_array[char_row][char_col] = ' ' ;

        fprintf(output_file, "%s%d%s",
                "HIT SEQUENCE #",
                leaf_node->get_ID(),
                ":\n\n") ;

        // Step through hit_entries[] filling in the names of symbols
        // from the driving_pattern and the target_pattern.

        int d_char_pos = 0, t_char_pos = 0, name_index, hit_char_pos,
                temp_pos, c_col_for_link, symbol_length ;
        char *symbol_name, c ;
        driving_pattern->initialise() ;
        target_pattern->initialise() ;
        for (hit_index = 0; hit_index < fe_hit_entries; hit_index++)
        {
                h_node = hit_entries[hit_index].h_node ;
                hit_d_symbol = h_node->get_driving_symbol() ;
                hit_t_symbol = h_node->get_target_symbol() ;

                // Fill in the names of the non-hit symbols before
                // hit_d_symbol and hit_t_symbol.

                while (d_symbol = (symbol *)driving_pattern->get_next_child())
                {
                        if (d_symbol == hit_d_symbol) break ;

                        symbol_name = d_symbol->get_name() ;
                        name_index = 0 ;
                        c = *symbol_name ;
                        while (c != '\0')
                        {
                                character_array[0][d_char_pos++] = c ;
                                c = symbol_name[++name_index] ;
                        }

                        // We need one space between symbol names.
                        plus_one(&d_char_pos, LARGE_SCRATCH_ARRAY_SIZE,
                                "Overflow in write_hit_sequence_horizontal()") ;
                }
```

```
        while (t_symbol = (symbol *)target_pattern->get_next_child())
        {
                if (t_symbol == hit_t_symbol) break ;

                symbol_name = t_symbol->get_name() ;
                name_index = 0 ;
                c = *symbol_name ;
                while (c != '\0')
                {
                        character_array[2][t_char_pos++] = c ;
                        c = symbol_name[++name_index] ;
                }
                // We need one space between symbol names.
                plus_one(&t_char_pos, LARGE_SCRATCH_ARRAY_SIZE,
                        "Overflow in write_hit_sequence_horizontal()") ;
        }

        // Now fill in the hit symbols and the '|' character between them.

        temp_pos = // Maximum of d_char_pos and t_char_pos.
                (d_char_pos > t_char_pos) ? d_char_pos : t_char_pos ;

        symbol_name = hit_d_symbol->get_name() ;
        name_index = 0 ;
        c = *symbol_name ;
        hit_char_pos = temp_pos ;
        while (c != '\0')
        {
                if (hit_char_pos > LARGE_SCRATCH_ARRAY_SIZE)
                        abort_run("Overflow in write_hit_sequence_horizontal()") ;
                character_array[0][hit_char_pos++] = c ;
                c = symbol_name[++name_index] ;
        }

        d_char_pos = hit_char_pos + 1 ; // Add 1 for space character.

        symbol_name = hit_t_symbol->get_name() ;
        name_index = 0 ;
        c = *symbol_name ;
        hit_char_pos = temp_pos ;
        while (c != '\0')
        {
                character_array[2][hit_char_pos++] = c ;
                c = symbol_name[++name_index] ;
        }

        t_char_pos = hit_char_pos + 1 ; // Add 1 for space character.

        // The '|' should be half way along the length of the hit
        // symbol name.

        symbol_length = strlen(symbol_name) ;
        c_col_for_link = temp_pos + (symbol_length / 2) ;

        character_array[1][c_col_for_link] = '|' ;
}

// Now do trailing non-hit symbols, if any.

while (d_symbol = (symbol *)driving_pattern->get_next_child())
{
        symbol_name = d_symbol->get_name() ;
        name_index = 0 ;
        c = *symbol_name ;
        while (c != '\0')
        {
                character_array[0][d_char_pos++] = c ;
                c = symbol_name[++name_index] ;
        }
        // We need one space between symbol names.
        plus_one(&d_char_pos, LARGE_SCRATCH_ARRAY_SIZE,
                "Overflow in write_hit_sequence_horizontal()") ;
}

while (t_symbol = (symbol *)target_pattern->get_next_child())
{
        symbol_name = t_symbol->get_name() ;
        name_index = 0 ;
        c = *symbol_name ;
        while (c != '\0')
        {
                character_array[2][t_char_pos++] = c ;
                c = symbol_name[++name_index] ;
```

```
                }
                // We need one space between symbol names.
                plus_one(&t_char_pos, LARGE_SCRATCH_ARRAY_SIZE,
                        "Overflow in write_hit_sequence_horizontal()") ;
        }

        temp_pos = // Maximum of d_char_pos and t_char_pos.
                (d_char_pos > t_char_pos) ? d_char_pos : t_char_pos ;

        // Now write out character_array[][].

        for (char_row = 0; char_row < 3; char_row++)
        {
                if (char_row == 0)
                        fprintf(output_file, "%d%c",
                                0,
                                ' ') ;
                else if (char_row == 1) fprintf(output_file, "  ") ;
                else fprintf(output_file, "%d%c",
                                1,
                                ' ') ;

                for (char_col = 0; char_col < temp_pos; char_col++)
                {
                        fprintf(output_file, "%c",
                                character_array[char_row][char_col]) ;
                }
                if (char_row == 0)
                        fprintf(output_file, "%d%c",
                                0,
                                '\n') ;
                else if (char_row == 1) fprintf(output_file, "\n") ;
                else fprintf(output_file, "%d%c",
                                1,
                                '\n') ;
        }

        fprintf(output_file, "\n") ;

} // write_hit_sequence_horizontal

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Using a hit sequence recorded in hit_entries[], write the
**        corresponding alignment vertically.
**
** CALLING SEQUENCE:
**
**        void write_hit_sequence_vertical()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        hit_entries[]
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void write_hit_sequence_vertical()
{
        hit_node *leaf_node = hit_entries[fe_hit_entries - 1].h_node,
                *h_node ;
        int hit_index, char_row, char_col, d_char_pos = 0,
                t_char_pos = 0, name_index, hit_char_pos, symbol_length ;
        char character_array[LARGE_SCRATCH_ARRAY_SIZE]
                [MEDIUM_SCRATCH_ARRAY_SIZE], *symbol_name, c ;
        sequence *driving_pattern = leaf_node->get_driving_pattern(),
                *target_pattern = leaf_node->get_target_pattern() ;
        symbol *d_symbol, *hit_d_symbol, *t_symbol, *hit_t_symbol ;

        // Clear character_array[][].
```

```
for (char_row = 0; char_row < LARGE_SCRATCH_ARRAY_SIZE; char_row++)
        for (char_col = 0; char_col < MEDIUM_SCRATCH_ARRAY_SIZE;
                char_col++)
                        character_array[char_row][char_col] = ' ' ;

fprintf(output_file, "%s%d%s",
        "HIT SEQUENCE #",
        leaf_node->get_ID(),
        ":\n\n") ;

// Find width of column 1 - the length of the longest symbol
// name in the driving_pattern.

int col_1_width = 0  ;
list_for(d_symbol, symbol, driving_pattern)
{
        symbol_name = d_symbol->get_name() ;
        symbol_length = strlen(symbol_name) ;
        if (symbol_length > col_1_width)
                col_1_width = symbol_length ;
}

int start_col_2 = col_1_width + 3 ; // Allow enough space
        // for at least ' - ' between columns.
int end_broken_line = start_col_2 - 1 ;

// Check that the character_array[][] is wide enough.

int col_2_width = 0  ;
list_for(t_symbol, symbol, target_pattern)
{
        symbol_name = t_symbol->get_name() ;
        symbol_length = strlen(symbol_name) ;
        if (symbol_length > col_2_width)
                col_2_width = symbol_length ;
}

int overall_width = start_col_2 + col_2_width, hit_row ;
if (overall_width > MEDIUM_SCRATCH_ARRAY_SIZE)
        abort_run("Array overflow in write_hit_sequence_vertical().") ;

// Step through hit_entries[] filling in the names of symbols
// from the driving_pattern and the target_pattern.

driving_pattern->initialise() ;
target_pattern->initialise() ;
int d_row = 0, t_row = 0 ;
for (hit_index = 0; hit_index < fe_hit_entries; hit_index++)
{
        h_node = hit_entries[hit_index].h_node ;
        hit_d_symbol = h_node->get_driving_symbol() ;
        hit_t_symbol = h_node->get_target_symbol() ;

        // Fill in the names of the non-hit symbols before
        // hit_d_symbol and hit_t_symbol.

        while (d_symbol = (symbol *)driving_pattern->get_next_child())
        {
                if (d_symbol == hit_d_symbol) break ;

                d_char_pos = 0 ;
                symbol_name = d_symbol->get_name() ;
                name_index = 0 ;
                c = *symbol_name ;
                while (c != '\0')
                {
                        character_array[d_row][d_char_pos++] = c ;
                        c = symbol_name[++name_index] ;
                }

                d_row++ ;
        }

        while (t_symbol = (symbol *)target_pattern->get_next_child())
        {
                if (t_symbol == hit_t_symbol) break ;

                t_char_pos = start_col_2 ;
                symbol_name = t_symbol->get_name() ;
                name_index = 0 ;
                c = *symbol_name ;
                while (c != '\0')
```

```
                {
                        character_array[t_row][t_char_pos++] = c ;
                        c = symbol_name[++name_index] ;
                }

                t_row++ ;
        }

        hit_row = (d_row > t_row) ? d_row : t_row ;

        // Now fill in the hit symbols.

        hit_char_pos = 0 ;
        symbol_name = hit_d_symbol->get_name() ;
        name_index = 0 ;
        c = *symbol_name ;
        while (c != '\0')
        {
                character_array[hit_row][hit_char_pos++] = c ;
                c = symbol_name[++name_index] ;
        }

        // Now put in a broken line up to the beginning of
        // the second column.

        hit_char_pos++ ;
        while (hit_char_pos < end_broken_line)
                character_array[hit_row][hit_char_pos++] = '-' ;

        hit_char_pos = start_col_2 ;
        symbol_name = hit_t_symbol->get_name() ;
        name_index = 0 ;
        c = *symbol_name ;
        while (c != '\0')
        {
                character_array[hit_row][hit_char_pos++] = c ;
                c = symbol_name[++name_index] ;
        }
}

// Now do trailing non-hit symbols, if any.

d_row = hit_row + 1 ;
while (d_symbol = (symbol *)driving_pattern->get_next_child())
{
        d_char_pos = 0 ;
        symbol_name = d_symbol->get_name() ;
        name_index = 0 ;
        c = *symbol_name ;
        while (c != '\0')
        {
                character_array[d_row][d_char_pos++] = c ;
                c = symbol_name[++name_index] ;
        }
        d_row++ ;
}

t_row = hit_row + 1 ;
while (t_symbol = (symbol *)target_pattern->get_next_child())
{
        t_char_pos = start_col_2 ;
        symbol_name = t_symbol->get_name() ;
        name_index = 0 ;
        c = *symbol_name ;
        while (c != '\0')
        {
                character_array[t_row][t_char_pos++] = c ;
                c = symbol_name[++name_index] ;
        }
        t_row++ ;
}

// Now write out character_array[][].

// First write column numbers at the top.

fprintf(output_file, "%c", '0') ;
for (int i = 1; i < start_col_2; i++)
        fprintf(output_file, "%c", ' ') ;
fprintf(output_file, "%c%s",
        '1',
        "\n\n") ;
```

```
        int row_limit = (d_row > t_row) ? d_row : t_row ;

        for (char_row = 0; char_row < row_limit; char_row++)
        {
                for (char_col = 0; char_col < overall_width; char_col++)
                {
                        fprintf(output_file, "%c",
                                character_array[char_row][char_col]) ;
                }
                fprintf(output_file, "\n") ;
        }

        // Write column numbers at the bottom.

        fprintf(output_file, "\n0") ;
        for (i = 1; i < start_col_2; i++)
                fprintf(output_file, "%c", ' ') ;
        fprintf(output_file, "%c%s",
                '1',
                "\n\n") ;

} // write_hit_sequence_vertical

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Displays a hit sequence in horizontal or vertical format.
**
** CALLING SEQUENCE:
**
**        void write_hit_sequence()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void write_hit_sequence()
{
        if (alignment_format == 'H')
                write_hit_sequence_horizontal() ;
        else write_hit_sequence_vertical() ;
} // write_hit_sequence

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        From the best hit sequence in the hit_structure, this function
**        forms a new alignment. It is assumed that every CONTENTS symbol
**        in the target pattern forms a hit with a symbol in the driving
**        pattern.
**
** CALLING SEQUENCE:
**
**        sequence *unify_best(int leaf_index, sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:                The new sequence, NIL if none is formed.
**
**        leaf_index:                        The position of the last node of
**                                                the current hit sequence in the leaf_array[].
**        cnp:                                 The New pattern being processed.
**
** IMPLICIT INPUTS:
**
**        The hit_structure
```

```
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

sequence *unify_best(int leaf_index, sequence *cnp)
{
        hit_node *leaf_node = leaf_array[leaf_index] ;
        sequence *matching_sequence, *new_alignment = NIL,
                *driving_pattern = leaf_node->get_driving_pattern() ;

        if (leaf_node == NIL)
                abort_run("NIL value for leaf_node in unify_best") ;

        if (verbose)
        {
                fprintf(output_file, "STARTING unify_best (") ;
                print_pattern_cycle(true, cnp) ;
                fprintf(output_file, ") LEAF NODE ID = ") ;
                leaf_node->print_ID() ;
                fprintf(output_file, "\n\n") ;
                fflush(output_file) ;
        }

        // 1 LOAD NODES OF THE HIT SEQUENCE INTO hit_entries[] SO THAT
        // THEY CAN BE ACCESSED EASILY IN LEFT-TO-RIGHT ORDER (WHICH
        // IS MORE INTUITIVE).

        prepare_hit_sequence(leaf_node) ;

        // 3 FORM AN ALIGNMENT

        new_alignment = make_one_alignment(leaf_node, cnp) ;
        if (new_alignment == NIL) goto L1 ;

        // Check to see that the new alignment is legal in terms of the
        // embedding of brackets. If it is not, delete it.

        if (new_alignment->is_legal() == false)
        {
                fprintf(output_file, "%s%d%s",
                        "The following alignment (ID",
                        new_alignment->get_ID(),
                        ") is not legal and is deleted.\n\n") ;
                new_alignment->write_alignment(output_file,
                        write_section_chars_length, NULL_VALUE, alignment_format) ;
                delete new_alignment ;
                new_alignment = NIL ;
                goto L1 ;
        }

        // Check to see whether new alignment is composite and mark it
        // as such if it is.

        new_alignment->check_for_composite_structure() ;

        // 4 CLASSIFY ALIGNMENT BY DEGREE OF MATCHING
        // AND RECOMPUTE SCORES OF THE NEW ALIGNMENT

        new_alignment->find_degree_of_matching(cnp) ;
        new_alignment->make_code(false) ;

        leaf_node->set_new_symbols_cost(new_alignment->get_new_symbols_cost()) ;
        leaf_node->set_encoding_cost(new_alignment->get_encoding_cost()) ;
        leaf_node->set_compression_ratio(new_alignment->get_compression_ratio()) ;
        leaf_node->set_compression_difference(new_alignment->
                get_compression_difference()) ;

        // 5 CHECK FOR LOW SCORING ALIGNMENT

        if (new_alignment->get_compression_difference() <= fail_score)
        {
                if (verbose)
                {
                        fprintf(output_file, "Alignment ") ;
                        new_alignment->print_ID() ;
                        fprintf(output_file, " for node ") ;
                        leaf_node->print_ID() ;
```

```
                        fprintf(output_file, " has a fail score and is deleted.\n\n") ;
                        fflush(output_file) ;
                }

                leaf_node->set_compression_difference(fail_score) ;

                delete new_alignment ;
                new_alignment = NIL ;
                goto L1 ;
        }

        if (show_al_structure)
                new_alignment->show_al_structure(leaf_node) ;

        // 6 CHECK WHETHER NEW ALIGNMENT IS A DUPLICATE OF ANY
        // PREVIOUSLY-FORMED ALIGNMENT

        // Match the new sequence against sequences formed earlier in
        // this cycle and previous cycles. If there is a match, delete
        // new sequence and return NIL.

        if (matching_sequence =
                new_alignment->matches_earlier_alignment())
        {
                if (verbose)
                {
                        fprintf(output_file, "Alignment ") ;
                        new_alignment->print_ID() ;
                        fprintf(output_file, " for node ") ;
                        leaf_node->print_ID() ;
                        fprintf(output_file, " matches sequence ") ;
                        matching_sequence->print_ID(),
                        fprintf(output_file, " and is discarded.\n\n") ;
                        fflush(output_file) ;
                }

                leaf_node->set_compression_difference(fail_score) ;
                delete new_alignment ;
                new_alignment = NIL ;
                goto L1 ;
        }

        // 8 DELIVER NEW ALIGNMENT

        alignments_array[leaf_index] = new_alignment ;

        L1: ;

        if (verbose)
        {
                fprintf(output_file, "FINISHED unify_best (") ;
                print_pattern_cycle(true, cnp) ;
                fprintf(output_file, ") LEAF NODE ID = ") ;
                leaf_node->print_ID() ;
                fprintf(output_file, "\n\n") ;
                fflush(output_file) ;
        }

        return(new_alignment) ;

} /* unify_best */

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Removes from alignments_array[] any sequences and corresponding codes
**        which are not marked as KEEP. At the same time, clear the
**        symbol_selection_array[][] ready for the next cycle.
**
** CALLING SEQUENCE:
**
**        void purge_parsing_alignments(sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        cnp:                        Current New pattern.
**
** IMPLICIT INPUTS:
**
```

```
**        alignments_array[]
**
** IMPLICIT OUTPUTS:
**
**        Alignments and codes removed from alignments_array[]
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void purge_parsing_alignments(sequence *cnp)
{
        sequence *sequence2 ;
        int row1, col1, leaf_index ;
        hit_node *h_node ;

        fprintf(output_file, "STARTING purge_parsing_alignments (") ;
        print_pattern_cycle(true, cnp) ;
        fprintf(output_file, ")\n\n") ;
        fflush(output_file) ;

        // Clear symbol_selection_array[][] to avoid dangling pointers.

        for (row1 = 0; row1 < MAX_NEW_ARRAY_ROWS; row1++)
        {
                for (col1 = 0; col1 < MCOLS; col1++)
                        symbol_selection_array[row1][col1] = NIL ;
        }

        /* Delete the sequences that are not marked as KEEP
        and retain the ones that are. */

        number_of_alignments_purged = 0 ;
        row1 = 0 ;
        while (h_node = get_leaf_nodes_in_order(&row1, &sequence2, &leaf_index))
        {
                if (h_node == NIL) break ;
                if (sequence2 == NIL) continue ;

                /* Check to see if sequence2 is marked as KEEP. */

                if (sequence2->get_keep() == false)
                {
                        if (verbose)
                        {
                                fprintf(output_file, "Alignment ") ;
                                sequence2->print_ID() ;
                                fprintf(output_file, "%s%d%s",
                                        " (hit node #",
                                        sequence2->get_leaf_node_ID(),
                                        "), is purged\n\n") ;
                        }

                        delete sequence2 ;
                        alignments_array[leaf_index] = NIL ;
                        number_of_alignments_purged++ ;
                        continue ;
                }
        }

        fprintf(output_file, "%s%d%s",
                "Number of alignments purged = ",
                number_of_alignments_purged,
                "\n\n") ;

        fprintf(output_file, "FINISHED purge_parsing_alignments (") ;
        print_pattern_cycle(true, cnp) ;
        fprintf(output_file, ")\n\n") ;
        fflush(output_file) ;

} /* purge_parsing_alignments */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Selects a sub-set of the alignments formed in the current cycle.
**
**        This function uses the symbol_selection_array[][] to ensure that
**        as many sections of New as possible are represented in the
```

```
**        selection.
**
** CALLING SEQUENCE:
**
**        void make_selection(sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:                void
**
**        cnp:                                The New pattern being processed.
**
** IMPLICIT INPUTS:
**
**        old_patterns
**
** IMPLICIT OUTPUTS:
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void make_selection(sequence *cnp)
{
        symbol *B_symbol1 ;
        alignment_element *al_el1 ;
        int row1, row2, col, type_of_alignment,
                max_entries_in_symbol, col_n, symbol_counter, leaf_index,
                length_of_new = cnp->get_number_of_children() ;
        sequence *sequence1 ;
        hit_node *h_node ;

        row1 = 0 ;
        while (h_node = get_leaf_nodes_in_order(&row1, &sequence1, &leaf_index))
        {
                if (sequence1 == NIL) continue ;

                type_of_alignment = sequence1->get_degree_of_matching() ;

                if (phase == 2 && type_of_alignment == PARTIAL) continue ;

                /* Look for symbols from New in the sequence which form
                hits with symbols in Old. Make entries in the corresponding
                symbols of symbol_selection_array[][]. */

                list_for(B_symbol1, symbol, sequence1)
                {
                        al_el1 = B_symbol1->get_al_el(0) ;
                        if (al_el1->get_el_obj() == NIL) continue ;
                        if (al_el1->get_same_column_below() == NULL_VALUE)
                                continue ;

                        /* al_el1 is in a symbol/column which contains at
                        least one hit. Find the position of the
                        symbol in New linked to this al_el. */

                        col_n = al_el1->get_orig_patt_int_pos() ;

                        /* Check whether there are any empty slots for
                        this symbol position in symbol_selection_array[][].
                        If there are, insert a pointer to sequence1
                        and quit. If not, proceed to next section. */

                        for (row2 = 0; row2 < keep_rows; row2++)
                        {
                                if (symbol_selection_array[row2][col_n] == NIL)
                                {
                                        symbol_selection_array[row2][col_n] =
                                                sequence1 ;
                                        break ;
                                }
                        }
                }
        }

        /* Now traverse the symbol_selection_array[][] compiling a list of the
        alignments to be retained and used as driving patterns on the
        next cycle. */

        // Driving alignments:
```

```
        for (row1 = 0; row1 < keep_rows; row1++)
        {
                for (col = 0; col < length_of_new; col++)
                {
                        sequence1 = symbol_selection_array[row1][col] ;
                        if (sequence1 == NIL) continue ;
                        sequence1->set_keep(true) ;
                }
        }

        /* Find the maximum number of entries in any symbol of the
        symbol_selection_array[][] and print it out. */

        max_entries_in_symbol = 0 ;
        for (col = 0; col < length_of_new; col++)
        {
                symbol_counter = 0 ;
                for (row1 = 0; row1 < keep_rows; row1++)
                {
                        if (symbol_selection_array[row1][col] == NIL) break ;
                        symbol_counter++ ;
                }
                if (symbol_counter > max_entries_in_symbol)
                        max_entries_in_symbol = symbol_counter ;
        }

        fprintf(output_file, "FOR ") ;
        print_pattern_cycle(1, cnp) ;
        fprintf(output_file, "%s%s%d%s",
                ",\nTHE MAXIMUM NUMBER OF ENTRIES IN A COLUMN OF ",
                "symbol_selection_array[][] IS ",
                max_entries_in_symbol,
                "\n\n") ;

        // Remove from alignments_array[] any alignment which is not
        // marked as KEEP. At the same time, clear the
        // symbol_selection_array[][] ready for the next cycle.

        purge_parsing_alignments(cnp) ;

        // Transfer alignments to parsing_alignments.

        int hit_node_row = 0 ;
        sequence *al1 ;

        while (h_node = get_leaf_nodes_in_order(&hit_node_row, &al1,
                &leaf_index))
        {
                if (al1 == NIL) continue ;
                parsing_alignments->add_child(al1) ;
        }

        parsing_alignments->sort_by_compression_difference() ;

        if (show_all_parsing_alignments)
        {
                // Print out selected alignments.

                fprintf(output_file,
                        "SELECTED SET OF ALIGNMENTS FORMED IN ") ;
                print_pattern_cycle(1, cnp) ;
                fprintf(output_file, "\n\n") ;

                list_for(al1, sequence, parsing_alignments)
                {
                        if (al1->get_keep() == false) continue ;
                        al1->write_out_fully("SELECTED ALIGNMENT", h_node,
                                write_section_chars_length, NULL_VALUE, true,
                                cnp) ;
                }
        }

} /* make_selection */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Creates Shannon-Fano-Elias codes according to the method in
**        Cover and Thomas, pp 101-103 but modified to remove rounding
**        of values and the addition of 1. The output is put in sfe_array[].
**
```

```
**        The S-F-E method gives a code with this size:
**
**                l(x) = ceil(log_2(1 / p(x)) + 1.
**
**        This function does not construct an explicit code. It simply
**        calculates:
**
**                l(x) = log_2(1 / p(x)).
**
**        If a symbol has a frequency of 0, its bit cost is set to
**        average_symbol_type_cost. This is a purely arbitrary value
**        failing anything better.
**
** CALLING SEQUENCE:
**
**        void modified_sizes_of_sfe_codes(group *object_list)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        object_list:        The list of objects for which sizes of codes are to
**                                be calculated.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        Codewords and lengths for codewords in sfe_array[].
**
** SIDE EFFECTS:
**
**        NONE
*/

void modified_sizes_of_sfe_codes(group *object_list)
{
        double total_frequency = 0 ;
        base_object *obj1 ;
        int i, fe_sfe_array, size_of_array =
                object_list->get_number_of_children() ;
        struct sfe_entry
        {
                base_object *obj ;
                double prob_x ;
                double code_size ;
        } *sfe_array ;

        sfe_array = new sfe_entry[size_of_array] ;

        /* Clear sfe_array[]. */

        for (i = 0; i < size_of_array; i++)
        {

                sfe_array[i].obj = NIL ;
                sfe_array[i].prob_x = 0 ;
                sfe_array[i].code_size = 0 ;
        }

        /* Enter objects in sfe_array[] and find total frequency for objects. */

        i = 0 ;
        object_list->initialise() ;
        while (obj1 = object_list->get_next_child())
        {
                sfe_array[i].obj = obj1 ;
                plus_one(&i, LARGE_SCRATCH_ARRAY_SIZE,
                        "sfe_array[] is too small") ;
                total_frequency += obj1->get_frequency() ;
        }

        fe_sfe_array = i ;

        /* Calculate the probability of each object and enter values
        in sfe_array[]. */

        int obj_frequency ;
        for (i = 0; i < fe_sfe_array; i++)
        {
                obj_frequency = sfe_array[i].obj->get_frequency() ;
```

```
                if (obj_frequency == 0) sfe_array[i].prob_x = 0 ;
                else sfe_array[i].prob_x = obj_frequency / total_frequency ;
        }

        /* Calculate values for code_size. Each one is the size of the
        code for the corresponding object. */

        double prob_x ;
        for (i = 0; i < fe_sfe_array; i++)
        {
                prob_x = sfe_array[i].prob_x ;
                if (prob_x == 0) sfe_array[i].code_size =
                        average_symbol_type_cost ;
                else sfe_array[i].code_size = log_2(1 / prob_x) ;
        }

        /* Apply the sizes to the objects in object_list. */

        for (i = 0; i < fe_sfe_array; i++)
                sfe_array[i].obj->set_encoding_cost(sfe_array[i].code_size) ;

        delete[] sfe_array ;

} /* modified_sizes_of_sfe_codes */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Tests whether the symbols up to and including 'end_symbol' in
**        'pattern' render that pattern unique within Old.
**
** CALLING SEQUENCE:
**
**        bool is_unique(sequence *test_pattern, symbol *end_symbol,
**                sequence **matching_pattern)
**
** FORMAL ARGUMENTS:
**
**        Return value:                true if the combination of symbols up to and
**                                             including 'end_symbol' render the
**                                             test_pattern unique within old_patterns,
**                                             false otherwise.
**
**        test_pattern:                A pattern within old_patterns.
**        end_symbol:                 A symbol in test_pattern which marks the end
**                                             of the set of symbols to be tested.
**        matching_pattern:        A pointer to a variable which is NIL if
**                                             test_pattern is unique or takes the value
**                                             of a pattern where the leading symbols match
**                                             the test_pattern.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

bool is_unique(sequence *test_pattern, symbol *end_symbol,
                                sequence **matching_pattern)
{
        list_element *el_pos1 = NIL, *el_pos2, *el_pos3 ;
        sequence *pattern ;
        symbol *symbol1, *symbol2 ;

        while (pattern = (sequence *)
                new_patterns->get_next_child_by_el_pos(&el_pos1))
        {
                if (pattern == test_pattern) continue ;
                el_pos2 = el_pos3 = NIL ;
                list_for_el_pos(symbol1, symbol, test_pattern, el_pos2)
                {
                        symbol2 = (symbol *)
                                pattern->get_next_child_by_el_pos(&el_pos3) ;
                        if (symbol2 == NIL)
```

```
                        {
                                *matching_pattern = pattern ;
                                return(false) ;
                        }
                        if (!symbol1->name_matches(symbol2)) break ;
                        if (symbol1 == end_symbol)
                        {
                                *matching_pattern = pattern ;
                                return(false) ;
                        }
                }
        }

        el_pos1 = NIL ;

        while (pattern = (sequence *)
                old_patterns->get_next_child_by_el_pos(&el_pos1))
        {
                if (pattern == test_pattern) continue ;
                el_pos2 = el_pos3 = NIL ;
                while (symbol1 = (symbol *)
                        test_pattern->get_next_child_by_el_pos(&el_pos2))
                {
                        symbol2 = (symbol *)
                                pattern->get_next_child_by_el_pos(&el_pos3) ;
                        if (symbol2 == NIL)
                        {
                                *matching_pattern = pattern ;
                                return(false) ;
                        }
                        if (!symbol1->name_matches(symbol2)) break ;
                        if (symbol1 == end_symbol)
                        {
                                *matching_pattern = pattern ;
                                return(false) ;
                        }
                }
        }
        *matching_pattern = NIL ;
        return(true) ;
} /* is_unique */

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        A function that prints out, in sorted order,
**        the compression scores of all the hit sequences in the
**        hit structure.
**
** CALLING SEQUENCE:
**
**        void print_hit_scores()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void print_hit_scores()
{
        int row, leaf_index ;
        sequence *al1 ;
        hit_node *h_node ;

        fprintf(output_file, "COMPRESSION SCORES OF ALIGNMENTS:\n\n") ;

        row = 0 ;
        while (h_node = get_leaf_nodes_in_order(&row, &al1, &leaf_index))
```

```
                fprintf(output_file, "%s%1.2f%s%1.2f%s",
                "CR = ",
                h_node->get_compression_ratio(),
                ", CD = ",
                h_node->get_compression_difference(),
                "\n") ;
        fprintf(output_file, "\n") ;
        fflush(output_file) ;
} /* print_hit_scores */

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Sets up a new (empty) structure for storing hit nodes.
**
** CALLING SEQUENCE:
**
**        void set_up_hit_structure()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void set_up_hit_structure()
{
        int i ;

        hit_root = new hit_node ; // It is assumed that any pre-existing
                // tree of hit nodes was deleted towards the end of recognise().

        for (i = 0; i < hit_structure_rows; i++)
        {
                leaf_array[i] = NIL ;
                alignments_array[i] = NIL ;
                sort_array[i] = i ;
        }
        fe_sort = 0 ;
} /* set_up_hit_structure */

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Called from write_hit_structure().
**
** CALLING SEQUENCE:
**
**        void shs(hit_node *h_node, int indentation)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        h_node:                        The top node of the structure.
**        indendation:        The amount of indentation.
**
** IMPLICIT INPUTS:
**
**        The hit structure
**
** IMPLICIT OUTPUTS:
**
**        Output
**
** SIDE EFFECTS:
**
**        NONE
```

```
*/

void shs(hit_node *h_node, int indentation)
{
        hit_node *child ;
        int i ;

        for (i = 0; i < indentation; i++) fprintf(output_file, "  ") ;

        if (h_node != hit_root)
        {
                h_node->print_ID() ;
                fprintf(output_file, ": ") ;
                (h_node->get_driving_pattern())->print_ID() ;
                fprintf(output_file, "%s%s%s%d%s",
                        ", ",
                        (h_node->get_driving_symbol())->get_name(),
                        ", d_pos ",
                        h_node->get_driving_int_pos(),
                        "; ") ;
                (h_node->get_target_pattern())->print_ID() ;
                fprintf(output_file, "%s%s%s%d%s%1.2f%s%1.2f%s",
                        ", ",
                        (h_node->get_target_symbol())->get_name(),
                        ", t_pos ",
                        h_node->get_target_int_pos(),
                        " (CR = ",
                        h_node->get_compression_ratio(),
                        ", CD = ",
                        h_node->get_compression_difference(),
                        ")") ;
        }
        else fprintf(output_file, "hit_root") ;
        fprintf(output_file, "\n") ;

        h_node->initialise() ;
        while (child = (hit_node *)h_node->get_next_child())
                shs(child, indentation + 1) ;
} /* shs */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Prints out hit structure (for diagnosis).
**
** CALLING SEQUENCE:
**
**        void write_hit_structure(hit_node *h_node, int indentation) ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        h_node:                        The top node of the structure.
**        indendation:        The amount of indentation.
**
** IMPLICIT INPUTS:
**
**        The hit structure
**
** IMPLICIT OUTPUTS:
**
**        Output
**
** SIDE EFFECTS:
**
**        NONE
*/

void write_hit_structure(hit_node *h_node, int indentation)
{
        write_message("HIT STRUCTURE:") ;
        shs(h_node, indentation) ;
        fprintf(output_file, "\n") ;
        fflush(output_file) ;
} /* write_hit_structure */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
```

```
**
**        Returns a new unique id number from unique_id_number.
**
** CALLING SEQUENCE:
**
**        int get_new_unique_id_number()
**
** FORMAL ARGUMENTS:
**
**        Return value:        The new unique id number.
**
** IMPLICIT INPUTS:
**
**        context_number
**
** IMPLICIT OUTPUTS:
**
**        A new value for unique_id_number.
**
** SIDE EFFECTS:
**
**        NONE
**
*/

int get_new_unique_id_number()
{
        double unique_id_number_length = ceil(log10(unique_id_number)) ;
        if (unique_id_number_length > STRING_LENGTH - 2)
                abort_run("String too long in get_new_unique_id_number()") ;
        return(unique_id_number++) ;
} // get_new_unique_id_number

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Creates a symbol from context_number and adds it to
**        generalisation_list1.
**
** CALLING SEQUENCE:
**
**        void add_symbol_to_generalisation_list1(int context_number)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        Symbol added to generalisation_list1.
**
** SIDE EFFECTS:
**
**        NONE
*/

void add_symbol_to_generalisation_list1(int context_number)
{
        char symbol_name[STRING_LENGTH] ;
        sprintf(symbol_name, "%d", context_number) ;

        symbol *cs = new symbol("", 1, NULL_VALUE) ;
        cs->set_name(symbol_name) ;
        bool symbol_added =
                generalisation_list1->add_symbol_without_copies_or_duplicates(cs) ;
        if (symbol_added == false)
                delete cs ;
} // add_symbol_to_generalisation_list1

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Returns a new context number from code_number. This function also
**        creates a symbol with a name which is the string version of the
**        context number and adds this symbol to generalisation_list1.
```

```
**
** CALLING SEQUENCE:
**
**        int get_new_context_number()
**
** FORMAL ARGUMENTS:
**
**        Return value:        The new context number.
**
** IMPLICIT INPUTS:
**
**        context_number
**
** IMPLICIT OUTPUTS:
**
**        A new value for context_number.
**
** SIDE EFFECTS:
**
**        NONE
**
*/

int get_new_context_number()
{
        double context_number_length = ceil(log10(context_number)) ;
        if (context_number_length > STRING_LENGTH - 2)
                abort_run("String too long in get_new_context_number()") ;

        return(context_number++) ;
} // get_new_context_number

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Makes a new discrimination symbol using the integer value
**        stored in an internal static variable.
**
** CALLING SEQUENCE:
**
**        symbol *make_unique_id_symbol()
**
** FORMAL ARGUMENTS:
**
**        Return value:                        An object which is a new code symbol.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

symbol *make_unique_id_symbol()
{
        symbol *cs ;
        char symbol_name[STRING_LENGTH] ;

        // Make new unique id symbol.

        cs = new symbol("", 1, NULL_VALUE) ;

        int unique_id_num = get_new_unique_id_number() ;
        sprintf(symbol_name, "%s%d",
                "#",
                unique_id_num) ;
        cs->set_type(UNIQUE_ID_SYMBOL) ;
        cs->set_status(IDENTIFICATION) ;
    cs->set_name(symbol_name) ;

        // Assign a value for the bit_cost. For the time
        // being, all code symbols will have a bit_cost of CODE_BIT_COST bits.

        cs->assign_frequency_and_cost() ;
```

```
        return(cs) ;
} /* make_unique_id_symbol */

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        The frequencies of symbol types are used to compute the
**        information cost of each symbol type using a simplified version
**        of the Shannon-Fano-Elias method. Symbols with the type DATA_SYMBOL
**        have their information costs increased by cost_factor. This is
**        so that they can be treated as relatively large 'chunks' of
**        information, so that it becomes feasible to encode relatively
**        small patterns.
**
**        Calculates average_symbol_type_cost.
**
**        In addition, the function assigns the frequency of its type
**        to each symbol in the corpus and the associated bit_cost.
**
** CALLING SEQUENCE:
**
**        void calculate_and_assign_frequencies_and_costs(group *symbol_types,
**                bool ref_is_corpus)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        symbol_types:        A set of symbol types, each one with
**                                  an associated frequency.
**        ref_is_corpus:        if true, the reference patterns are the whole corpus,
**                                  (normally just the New patterns), otherwise Old.
**
** IMPLICIT INPUTS:
**
**        The corpus
**
** IMPLICIT OUTPUTS:
**
**        bit_cost value on each symbol in the corpus.
**
** SIDE EFFECTS:
**
**        NONE
*/

void calculate_and_assign_frequencies_and_costs(group *symbol_types,
        bool ref_is_corpus)
{
        symbol *symbol1 ;
        double total_symbol_type_cost = 0, temp_cost,
                total_CONTEXT_SYMBOL_type_cost = 0 ;
        int alphabet_size = symbol_types->get_number_of_children(),
                number_of_CONTEXT_types = 0 ;

        // Calculate sizes of symbols in symbol_types using a modified
        // version of the SFE method.

        modified_sizes_of_sfe_codes(symbol_types) ;

        // For all symbols that are DATA_SYMBOLs, inflate the bit_cost
        // by the cost_factor.

        list_for(symbol1, symbol, symbol_types)
        {
                if (symbol1->get_type() == DATA_SYMBOL)
                {
                        temp_cost = symbol1->get_bit_cost() ;
                        temp_cost *= cost_factor ;
                        symbol1->set_bit_cost(temp_cost) ;
                }
        }

        /* Print results. */

        if (ref_is_corpus)
                fprintf(output_file, "%s%s",
                        "FREQUENCIES OF SYMBOL TYPES IN CORPUS (NORMALLY JUST NEW PATTERNS)\n",
                        "AND INFORMATION COSTS (IN BITS)\n\n") ;
        else
        {
```

```
                fprintf(output_file, "%s%s",
                        "FREQUENCIES OF SYMBOL TYPES IN OLD PATTERNS\n",
                        "AND INFORMATION COSTS (IN BITS)\n(") ;
                print_pattern_cycle(false, current_new_pattern) ;
                fprintf(output_file, ")\n\n") ;
        }
        write_lines(output_file, "", START) ;
        list_for(symbol1, symbol, symbol_types)
        {
                fprintf(output_file, "%s%s%d%s%1.2f%s",
                        symbol1->get_name(),
                        ", Fr = ", symbol1->get_frequency(),
                        ", bit_cost = ", symbol1->get_bit_cost(),
                        "\n") ;
                total_symbol_type_cost += symbol1->get_bit_cost() ;

                if (symbol1->get_type() != CONTEXT_SYMBOL
                        && symbol1->get_type() != UNIQUE_ID_SYMBOL)
                                continue ;

                total_CONTEXT_SYMBOL_type_cost += symbol1->get_bit_cost() ;
                number_of_CONTEXT_types++ ;
        }

        fprintf(output_file, "\n") ;

        average_symbol_type_cost =
                total_symbol_type_cost / alphabet_size ;
        if (total_CONTEXT_SYMBOL_type_cost <= 0
                || number_of_CONTEXT_types <= 0)
                        average_CONTEXT_SYMBOL_type_cost = CODE_BIT_COST ;
        else average_CONTEXT_SYMBOL_type_cost =
                total_CONTEXT_SYMBOL_type_cost / number_of_CONTEXT_types ;

        fprintf(output_file, "%s%1.2f%s",
                "Average of bit_costs for symbol types = ",
                average_symbol_type_cost, "\n\n") ;

        // Now assign the values for frequency and bit_cost
        // to the symbol instances in the corpus.

        assign_symbol_frequencies_and_costs(symbol_types) ;

} /* calculate_and_assign_frequencies_and_costs */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Gets a new empty entry in leaf_array[]. If the number of entries
**        is exhausted, it returns false, otherwise true.
**
** CALLING SEQUENCE:
**
**        int get_new_leaf_entry() ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        The leaf_index value for the next available empty
**                                    cell in leaf_array. If none is available, NULL_VALUE
**                                    is returned.
**
** IMPLICIT INPUTS:
**
**        fe_sort
**
** IMPLICIT OUTPUTS:
**
**        new value for fe_sort
**
** SIDE EFFECTS:
**
**        NONE
*/

int get_new_leaf_entry()
{
        int leaf_index ;

        if (fe_sort >= hit_structure_rows) return(NULL_VALUE) ;
        leaf_index = sort_array[fe_sort++] ;
        return(leaf_index) ;
```

```
} /* get_new_leaf_entry */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Compiles an unordered list of the leaf nodes with the smallest
**        values for CD.
**
** CALLING SEQUENCE:
**
**        void find_worst_leaf_nodes(sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        cnp:                        Current New pattern.
**
** IMPLICIT INPUTS:
**
**        The hit structure.
**
** IMPLICIT OUTPUTS:
**
**        An unordered list of the worst leaf nodes in half_list[].
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void find_worst_leaf_nodes(sequence *cnp)
{
        int half_fe_sort = fe_sort / 2, row1, row2, best_worst_hl_index ;
        double best_worst_CD, temp_CD, time_started = clock(),
                time_finished, time_difference_secs, time_per_item_secs ;
        hit_node *h_node1 ;

        fe_half_list = 0 ;

        if (half_fe_sort < 1) abort_run("Too few entries in half_list[]") ;
        if (half_fe_sort > half_hit_structure_rows)
                abort_run("Anomalous value for half_fe_sort in find_mid") ;

        // Start by filling up half_list with the first half of the
        // nodes in leaf_array[]. At the same time, find the
        // best leaf node in the half list.

#if PRINT_CD_VALUES
        fprintf(output_file, "CD values for leaf nodes:\n\n") ;
#endif

        best_worst_CD = fail_score ;
        for (row1 = 0; row1 < half_fe_sort; row1++)
        {
                h_node1 = leaf_array[sort_array[row1]] ;
                if (h_node1 == NIL)
                abort_run("NIL value for h_node1 \
                in find_worst_leaf_nodes") ;
                half_list[fe_half_list] = h_node1 ;
                temp_CD = h_node1->get_compression_difference() ;

#if PRINT_CD_VALUES
                fprintf(output_file, "%1.2f", temp_CD) ;
                print_counter++ ;
                if (print_counter < 10) fprintf(output_file, ", ") ;
                else
                {
                        fprintf(output_file, ",\n") ;
                        print_counter = 0 ;
                }
#endif

                if (temp_CD > best_worst_CD)
                {
                        best_worst_CD = temp_CD ;
                        best_worst_hl_index = fe_half_list ;
                }
                fe_half_list++ ;
        }
```

```
        // Now continue with the second half of sort array[] comparing CD
        // values in the second half with CD values in half_list[]. If
        // a CD value is worse than the best value in half_list, it replaces
        // that value.

        for (row1 = row1; row1 < fe_sort; row1++)
        {
                h_node1 = leaf_array[sort_array[row1]] ;
                if (h_node1 == NIL)
                abort_run("NIL value for h_node1 \
                in find_worst_leaf_nodes") ;
                temp_CD = h_node1->get_compression_difference() ;

#if PRINT_CD_VALUES
                fprintf(output_file, "%1.2f", temp_CD) ;
                print_counter++ ;
                if (row1 + 1 >= fe_sort) fprintf(output_file, ".\n\n") ;
                else if (print_counter < 10) fprintf(output_file, ", ") ;
                else
                {
                        fprintf(output_file, ",\n") ;
                        print_counter = 0 ;
                }
#endif

                if (temp_CD >= best_worst_CD) continue ;

                // temp_CD is worse than best_worst_CD so the corresponding
                // hit node replaces the hit node for the
                // best_worst_CD in half_list[]. Then it is necesssary to
                // find a new value for best_worst_CD and its corresponding
                // value for best_worst_hl_index.

                half_list[best_worst_hl_index] = h_node1 ;

                best_worst_CD = fail_score ;
                for (row2 = 0; row2 < half_fe_sort; row2++)
                {
                        temp_CD = half_list[row2]->
                                get_compression_difference() ;
                        if (temp_CD > best_worst_CD)
                        {
                                best_worst_CD = temp_CD ;
                                best_worst_hl_index = row2 ;
                        }
                }
        }

#if PRINT_CD_VALUES

        fprintf(output_file, "%s%1.2f%s",
                "Value of best_worst_CD = ",
                best_worst_CD,
                "\n\n") ;

        fprintf(output_file, "Values in half_list[]:\n\n") ;

        print_counter = 0 ;
        for (row1 = 0; row1 < half_fe_sort; row1++)
        {
                h_node1 = half_list[row1] ;
                temp_CD = h_node1->get_compression_difference() ;
                fprintf(output_file, "%1.2f", temp_CD) ;
                print_counter++ ;
                if (row1 + 1 >= half_fe_sort) fprintf(output_file, ".\n\n") ;
                else if (print_counter < 10) fprintf(output_file, ", ") ;
                else
                {
                        fprintf(output_file, ",\n") ;
                        print_counter = 0 ;
                }
        }
#endif

        time_finished = clock() ;
        time_difference_secs = (time_finished - time_started) / CLOCKS_PER_SEC ;
        time_per_item_secs = time_difference_secs / (double)fe_sort ;

        fprintf(output_file,
                "Finding worst leaf nodes started and finished.\n(") ;

        print_pattern_cycle(true, cnp) ;
```

```
        fprintf(output_file, "%s%d%s%g%s%g%s",
                ")\n\nNumber of items = ",
                fe_sort,
                ",\nSifting time = ",
                time_difference_secs,
                " seconds,\nSifting time per item = ",
                time_per_item_secs,
                " seconds\n\n") ;
} // find_worst_leaf_nodes

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Purges the hit structure. There is an assumption that the
**        leaf nodes have been sorted.
**
** CALLING SEQUENCE:
**
**        void purge_hit_structure(sequence *cnp) ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        cnp:                        Current New pattern
**
** IMPLICIT INPUTS:
**
**        The hit structure
**
** IMPLICIT OUTPUTS:
**
**        The purged hit structure
**
** SIDE EFFECTS:
**
**        NONE
*/

void purge_hit_structure(sequence *cnp)
{
        hit_node *h_node, *parent_h_node, **leaf_entry ;
        int row, fe_sort_at_start = fe_sort ;
        double start_end_fe_sort ;

        fprintf(output_file, "PURGING OF HIT_STRUCTURE (") ;
        print_pattern_cycle(true, cnp) ;
        fprintf(output_file, "%s%d%s",
                ")\n\nValue of fe_sort at start = ",
                fe_sort,
                "\n\n") ;

        find_worst_leaf_nodes(cnp) ; // Compiles an unordered list of the
        // worst leaf nodes. Then these nodes can be purged.

        /* half_list[] contains the bottom 50% of the leaf nodes. Delete
        each of these hit nodes, pruning each branch
        up to the first node that has more than one child. */

        for (row = 0; row < fe_half_list; row++)
        {
                h_node = half_list[row] ;
                if (h_node == NIL)
                abort_run("NIL value for h_node in \
                purge_hit_structure") ;

                leaf_entry = h_node->get_leaf_entry() ; // This is the cell
                        // in leaf_array[] which contains a pointer to
                        // a pointer to a leaf node.
                (*leaf_entry) = NIL ;
                while (h_node != hit_root)
                {
                        parent_h_node = h_node->get_hn_parent() ;
                        parent_h_node->extract_child(h_node) ;
                        delete h_node ;

                        /* Test whether the parent_h_node has any other
                        children. If so, stop extracting nodes
                        up this branch of the tree. */
```

```
                        if (parent_h_node->get_first_child() != NIL) break ;
                        h_node = parent_h_node ;
                }
        }

        // Now compute new values for fe_sort so that the NIL values in
        // leaf_array[] are not referenced by any cells in fe_sort[] between
        // 0 and fe_sort.

        fe_sort = 0 ;
        for (row = 0; row < hit_structure_rows; row++)
                if (leaf_array[row] != NIL) sort_array[fe_sort++] = row ;

                int i = fe_sort ;
                for (row = 0; row < hit_structure_rows; row++)
                        if (leaf_array[row] == NIL) sort_array[i++] = row ;

                start_end_fe_sort = (double)fe_sort / (double)fe_sort_at_start ;
                fprintf(output_file, "%s%d%s%1.2f%s",
                        "Value of fe_sort at end = ",
                        fe_sort,
                        "\n\nCurrent value of fe_sort is ",
                        start_end_fe_sort,
                        " of value before purging\n\n") ;

                fprintf(output_file, "PURGE IS COMPLETED (") ;
                print_pattern_cycle(true, cnp) ;
                fprintf(output_file, ")\n\n") ;

} /* purge_hit_structure */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Estimates compression scores for hit sequences added to the
**        hit structure.
**
**        The adjustment for gaps between hits used in earlier versions
**        has been dropped.
**
** CALLING SEQUENCE:
**
**        void estimate_compression_scores(double *NSC,
**                double *EC, hit_node *parent_h_node)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        NSC:                        A pointer to a double variable in the calling function.
**        EC:                                A pointer to a double variable in the calling function.
**        parent_h_node:        The parent hit node.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void estimate_compression_scores(double *NSC,
        double *EC, hit_node *parent_h_node)
{
        // The adjustment for gaps between hits used in earlier versions
        // has been dropped.

        if (cycle == 1)
        {
                /* Driving pattern is one of the New patterns. */

                if (parent_h_node == hit_root)
                {
                        *NSC = master_driving_symbol->get_bit_cost() +
                                master_target_pattern->get_new_symbols_cost() ;
                        *EC = master_target_pattern->get_encoding_cost() ;
```

```
                }
                else
                {
                        *NSC = parent_h_node->get_new_symbols_cost() +
                                master_driving_symbol->get_bit_cost() ;
                        *EC = parent_h_node->get_encoding_cost() ;
                }
        }
        else
        {
                // Driving pattern is an alignment from a previous cycle.

                if (parent_h_node == hit_root)
                {
                        *NSC = master_driving_pattern->get_new_symbols_cost() +
                                master_target_pattern->get_new_symbols_cost() ;
                        *EC = master_driving_pattern->get_encoding_cost() +
                                master_target_pattern->get_encoding_cost() ;
                }
                else
                {
                        *NSC = parent_h_node->get_new_symbols_cost() ;
                        *EC = parent_h_node->get_encoding_cost() ;
                }

                // Make an adjustment to EC if the current hit involves at
                // least one IDENTIFICATION symbol.

                if (master_driving_symbol->get_status() == IDENTIFICATION
                        || master_target_symbol->get_status() == IDENTIFICATION)
                                *EC -= master_target_symbol->get_bit_cost() ;
        }
} // estimate_compression_scores

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        This function makes a hit_node. It is called from add_hit().
**
** CALLING SEQUENCE:
**
**        void make_hit_node(hit_node *parent_h_node, sequence *pattern1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        parent_h_node:        The parent of the hit_node to be created.
**        pattern1:                The pattern being parsed.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        A new hit_node attached to a chain of 'pending' hit nodes.
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void make_hit_node(hit_node *parent_h_node, sequence *pattern1)
{
        hit_node *copy_h_node = new hit_node(*hn_master) ;
        double NSC, EC ;

        /* Estimate the compression score for the hit sequence
        up to and including the new hit. */

        estimate_compression_scores(&NSC, &EC, parent_h_node) ;

        if (EC < 0) EC = 0 ;

        copy_h_node->set_new_symbols_cost(NSC) ;
        copy_h_node->set_encoding_cost(EC) ;
        copy_h_node->set_compression_ratio(EC / NSC) ;
        copy_h_node->set_compression_difference(NSC - EC) ;
```

```
        /* Attach the new copy to the hit_node_array[]. */

        hit_node_array[fe_hit_node_array].new_hit_node = copy_h_node ;
        hit_node_array[fe_hit_node_array].hn_parent = parent_h_node ;
        plus_one(&fe_hit_node_array, HIT_NODE_ARRAY_SIZE,
                "hit_node_array[] is too small.") ;
        copy_h_node->set_hn_parent(parent_h_node) ;

        new_hits++ ;

} /* make_hit_node */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Adds one or more records of a hit to the hit structure.
**        The function traverses the tree of hit_nodes recursively.
**        New nodes representing the hit are created wherever they are needed.
**        They are not added to the hit structure immediately but are
**        attached to a chain of hit nodes and then (in
**        update_hit_structure()) added        to the hit structure if there
**        is enough space to accommodate them. If there is not enough space,
**        the hit structure is purged and add_hit() is called again.
**
** CALLING SEQUENCE:
**
**        bool add_hit(hit_node *parent_h_node, sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if a hit is added, false otherwise.
**
**        parent_h_node:        The top node of the (sub) tree.
**        cnp:                        The New pattern being processed.
**
** IMPLICIT INPUTS:
**
**        Global structures, including hn_master.
**
** IMPLICIT OUTPUTS:
**
**        Global structures
**
** SIDE EFFECTS:
**
**        NONE
*/

bool add_hit(hit_node *parent_h_node, sequence *cnp)
{
        hit_node *child ;
        bool hit_added = false ;

        parent_h_node->initialise() ;
        while (child = (hit_node *)parent_h_node->get_next_child())
        {
                // compression_ratio == fail_score marks the child as
                // 'dead'. Nothing must be added below this node.

                if (child->get_compression_difference() <= fail_score)
                        continue ;

                if (master_driving_pattern !=
                        child->get_driving_pattern()) continue ;

                if (master_driving_int_pos <=
                        child->get_driving_int_pos())
                        continue ;

                if (master_target_pattern !=
                        child->get_target_pattern()) continue ;

                if (master_target_int_pos <=
                        child->get_target_int_pos())
                        continue ;

                hit_added = add_hit(child, cnp) ;
        }

        if (hit_added) return(true) ;
```

```
        // There is no child at this level which match all the criteria
        // for going deeper in the tree of hit nodes. So the hit can be
        // installed as a new leaf node child of parent_h_node,
        // provided the gap between driving symbols in the new node and
        // its parent are less than max_driving_gap and likewise for
        // the target symbols.

        if (parent_h_node != hit_root)
        {
                if ((master_driving_int_pos - parent_h_node->
                        get_driving_int_pos() - 1) > max_driving_gap)
                        return(false) ;
                if ((master_target_int_pos - parent_h_node->
                        get_target_int_pos() - 1) > max_target_gap)
                        return(false) ;
        }

        make_hit_node(parent_h_node, cnp) ;

        return(true) ;
} /* add_hit */

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Updates the data structures which record hits.
**
** CALLING SEQUENCE:
**
**        void update_hit_structure(sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:                void
**
**        cnp:                                The New pattern being processed.
**
** IMPLICIT INPUTS:
**
**        The hit structure
**
** IMPLICIT OUTPUTS:
**
**        The hit structure
**
** SIDE EFFECTS:
**
**        NONE
*/

void update_hit_structure(sequence *cnp)
{
        hit_node *parent_h_node, *node,
                **leaf_entry, *new_h_node ;
        int i, leaf_index ;

        new_hits = 0 ;
        fe_hit_node_array = 0 ;

        add_hit(hit_root, cnp) ;

        /* Check whether there is enough space in leaf_array[]. If not,
        purge the hit structure, clear the chain of hit nodes
        and try again. */

        if (fe_sort + new_hits >= hit_structure_rows)
        {
                purge_hit_structure(cnp) ;

                /* Now delete the new_hit_nodes in the hit_node_array[]. */

                for (i = 0; i < fe_hit_node_array; i++)
                {
                        delete hit_node_array[i].new_hit_node ;
                        hit_node_array[i].hn_parent = NIL ;
                }

                new_hits = 0 ;
                fe_hit_node_array = 0 ;

                add_hit(hit_root, cnp) ;
```

```
                if (fe_sort + new_hits >= hit_structure_rows)
                        abort_run("Too little space in leaf_array[]") ;
        }

        /* Now put the copies of hn_master in their proper places in
        the hit_structure. */

        for (i = 0; i < fe_hit_node_array; i++)
        {
                new_h_node = hit_node_array[i].new_hit_node ;
                parent_h_node = hit_node_array[i].hn_parent ;

                /* Check whether the parent_h_node is a leaf node. If it is,
                use the parent_h_node's entry in leaf_array[]. If not,
                check whether the new_h_node is NIL. If it is, take a
                new entry in leaf_array[]. If the parent_h_node is
                the hit_root then, although at first it is a leaf node,
                its entry in leaf_array[] is not available so a new
                entry must be taken. */

                if (parent_h_node == hit_root)
                {
                        leaf_index = get_new_leaf_entry() ;
                        if (leaf_index == NULL_VALUE)
                                abort_run("No leaf_array[] entries left") ;
                        leaf_array[leaf_index] = new_h_node ;
                        new_h_node->set_leaf_entry(&leaf_array[leaf_index]) ;
                }
                else
                {
                        parent_h_node->initialise() ;
                        node = (hit_node *)parent_h_node->get_next_child() ;
                        if (node == NIL)
                        {
                                // parent_h_node is itself a leaf node.

                                leaf_entry = parent_h_node->get_leaf_entry() ;
                                (*leaf_entry) = new_h_node ;
                                new_h_node->set_leaf_entry(leaf_entry) ;
                                parent_h_node->set_leaf_entry(NIL) ;
                        }
                        else
                        {
                                // parent_h_node is not a leaf node but
                                // the new_hit_node will be a new leaf node.

                                leaf_index = get_new_leaf_entry() ;
                                if (leaf_index == NULL_VALUE)
                                abort_run("No leaf_array[] entries \
                                left") ;
                                leaf_array[leaf_index] = new_h_node ;
                                new_h_node->
                                        set_leaf_entry(&leaf_array[leaf_index]);
                        }
                }
                parent_h_node->add_child(new_h_node) ;
        }

        /* Now clear the hit_node_array[] to avoid problems when
        hit nodes are deleted. */

        for (i = 0; i < HIT_NODE_ARRAY_SIZE; i++)
        {
                hit_node_array[i].new_hit_node = NIL ;
                hit_node_array[i].hn_parent = NIL ;
        }
} /* update_hit_structure */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Tests whether any one symbol from Old is being matched
**        with itself. If it is, this is an illegal hit which is ignored.
**        If two matching columns contain the same symbol from New, this
**        is accepted because there is still a bona fide match between symbols
**        in Old. When the new alignment is formed, the two appearances
**        of one symbol in New can be merged.
**
**        If a New pattern is matched with itself, any one instance of a
**        symbol cannot be matched with itself.
```

```
**
** CALLING SEQUENCE:
**
**        bool anomalous_hit(symbol *driving_symbol, int driving_pattern_depth,
**                symbol *target_symbol, int target_pattern_depth)
**
** FORMAL ARGUMENTS:
**
**        Return value:                        true if the hit anomalous. false otherwise.
**
**        driving_symbol:                        The driving symbol.
**        driving_pattern_depth:        The depth of the driving pattern.
**        target_symbol:                        The target symbol.
**        target_pattern_depth:        The depth of the target pattern.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

bool anomalous_hit(symbol *driving_symbol, int driving_pattern_depth,
                                        symbol *target_symbol, int target_pattern_depth)
{
        symbol *d_symbol, *t_symbol ;
        int driving_row, target_row ;

        if (driving_symbol == target_symbol) return(true) ;

        if (driving_pattern_depth == 1 && target_pattern_depth == 1)
                // Since the two symbols have already been compared
                // and given a negative result, we know that
                // the two symbols are not the same.
                return(false) ;

        if (driving_pattern_depth == 1)
        {
                // We know that the target_pattern_depth must be > 1.

                for (target_row = 0; target_row < target_pattern_depth;
                target_row++)
                {
                        t_symbol = target_symbol->get_row_symbol(target_row) ;
                        if (t_symbol == NIL) continue ;
                        if (driving_symbol == t_symbol) return(true) ;
                }
                return(false) ;
        }

        if (target_pattern_depth == 1)
        {
                // We know that the driving_pattern_depth must be > 1.

                for (driving_row = 0; driving_row < driving_pattern_depth;
                driving_row++)
                {
                        d_symbol = driving_symbol->get_row_symbol(driving_row) ;
                        if (d_symbol == NIL) continue ;
                        if (target_symbol == d_symbol) return(true) ;
                }
                return(false) ;
        }

        // driving_pattern_depth and target_pattern_depth must both be > 1.

        for (driving_row = 0; driving_row < driving_pattern_depth;
        driving_row++)
        {
                d_symbol = driving_symbol->get_row_symbol(driving_row) ;
                if (d_symbol == NIL) continue ;
                for (target_row = 0; target_row < target_pattern_depth;
                target_row++)
                {
                        t_symbol = target_symbol->get_row_symbol(target_row) ;
                        if (t_symbol == NIL) continue ;
                        if (d_symbol == t_symbol) return(true) ;
```

```
                }
        }
        return(false) ;
} /* anomalous_hit */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Compares two symbols and makes various tests to ensure that any
**        hit is a valid hit.
**
** CALLING SEQUENCE:
**
**        bool match_symbols(sequence *driving_pattern, symbol *driving_symbol,
**                list_element *dr_el_pos, sequence *target_pattern,
**                symbol *target_symbol, list_element *ts_el_pos)
**
** FORMAL ARGUMENTS:
**
**        Return value:               true if all tests are passed, false otherwise.
**
**        driving_pattern:        The driving pattern.
**        driving_symbol:         The driving symbol.
**        dr_el_pos:                      The el_pos of the driving symbol.
**        target_pattern:         The target pattern.
**        target_symbol:                The target symbol.
**        ts_el_pos:                      The el_pos of the target symbol.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        Additions to the hit structure
**
** SIDE EFFECTS:
**
**        NONE
**
*/

bool match_symbols(sequence *driving_pattern, symbol *driving_symbol,
        list_element *dr_el_pos, sequence *target_pattern,
        symbol *target_symbol, list_element *ts_el_pos)
{
        if (driving_symbol->name_matches(target_symbol) == false)
                return(false) ;

        int driving_pattern_depth = driving_pattern->get_sequence_depth(),
                target_pattern_depth = target_pattern->get_sequence_depth() ;

        if (anomalous_hit(driving_symbol, driving_pattern_depth, target_symbol,
                target_pattern_depth))
                return(false) ;

        return(true) ;

} // match_symbols

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Compares a driving pattern/alignment with a target pattern or
**        alignment. Brackets are matched only in a 'legal' manner.
**
** CALLING SEQUENCE:
**
**        void compare_patterns(sequence *driving_pattern,
**                sequence *target_pattern, sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:                       void.
**
**        driving_pattern:                The pattern from Old which 'drives' the
**                                                search for matching patterns.
**        target_pattern:                       The target pattern.
**        cnp:                                  The New pattern being processed.
```

```
**
** IMPLICIT INPUTS:
**
**        old_patterns
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void compare_patterns(sequence *driving_pattern,
        sequence *target_pattern, sequence *cnp)
{
        symbol *driving_symbol, *target_symbol ;
        int driving_symbol_int_pos, target_symbol_int_pos, dr_status, t_status ;
        list_element *dr_el_pos = NIL, *t_el_pos ;

        list_for_el_pos(driving_symbol, symbol, driving_pattern, dr_el_pos)
        {
                hn_master->set_driving_symbol(driving_symbol) ;
                master_driving_symbol = driving_symbol ;
                driving_symbol_int_pos = dr_el_pos->get_position() ;
                hn_master->set_driving_int_pos(driving_symbol_int_pos) ;
                master_driving_int_pos = driving_symbol_int_pos ;

                t_el_pos = NIL ;
                while (target_symbol = (symbol *)target_pattern->
                        get_next_child_by_el_pos(&t_el_pos))
                {
                        // When the driving pattern is the same as the
                        // target pattern, the test which follows
                        // prevents the comparison of symbol A with symbol B
                        // and then, later, the comparison of symbol B
                        // with symbol A.

                        target_symbol_int_pos = t_el_pos->get_position() ;

                        if (driving_pattern == target_pattern)
                                if (target_symbol_int_pos >= driving_symbol_int_pos)
                                        break ;

                        // Constraint on the matching of IDENTIFICATION (or BOUNDARY_MARKER)
                        // symbols against CONTENTS symbols and vice versa.

                        if (id_c_symbol_constraint)
                        {
                                dr_status = driving_symbol->get_status() ;
                                t_status = target_symbol->get_status() ;
                                if (dr_status == IDENTIFICATION
                                        || dr_status == BOUNDARY_MARKER)
                                {
                                        if (t_status != CONTENTS) continue ;
                                }
                                else if (dr_status == CONTENTS)
                                {
                                        if (t_status != IDENTIFICATION
                                                && t_status != BOUNDARY_MARKER)
                                                        continue ;
                                }
                        }

                        if (match_symbols(driving_pattern, driving_symbol,
                                dr_el_pos, target_pattern, target_symbol, t_el_pos))
                        {
                                hn_master->set_target_symbol(target_symbol) ;
                                master_target_symbol = target_symbol ;
                                hn_master->set_target_int_pos(target_symbol_int_pos) ;
                                master_target_int_pos = target_symbol_int_pos ;

                                update_hit_structure(cnp) ;
                        }
                }

                if (phase == 1 && cycle == 1)
                {
                        if (driving_pattern == current_new_pattern
                                && target_pattern == receptacle_pattern)
                        {
                                symbol *copy_driving_symbol = new symbol(*driving_symbol) ;
```

```
                                copy_driving_symbol->set_status(CONTENTS) ;
                                receptacle_pattern->add_child(copy_driving_symbol) ;
                        }
                }
        }
} // compare_patterns

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Finds a match for a 'reference' that has been found by find_reference()
**        and creates enties in the hit structure.
**
** CALLING SEQUENCE:
**
**        bool match_reference()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

bool match_reference()
{
        sequence *driving_pattern = reference_structure.driving_pattern ;
        symbol *t_class_symbol = reference_structure.t_class_symbol,
                *symbol1, *symbol2, *symbol3 ;
        list_element *el_pos1 = reference_structure.dr_el_pos, *el_pos2,
                *el_pos3 ;
        int depth_of_bracketting,
                driving_pattern_depth = driving_pattern->get_sequence_depth(),
                target_pattern_depth = 1, dr_status, t_status ;

        while (symbol1 = (symbol *)
                driving_pattern->get_next_child_by_el_pos(&el_pos1))
        {
                if (symbol1->get_type() != LEFT_BRACKET) continue ;
                el_pos2 = el_pos1 ;

                // Look for matching CONTEXT_SYMBOL.

                while (symbol2 = (symbol *)
                        driving_pattern->get_next_child_by_el_pos(&el_pos2))
                {
                        // The CONTEXT_SYMBOL must follow the LEFT_BRACKET without
                        // any other symbol types in between.

                        if (symbol2->get_type() != CONTEXT_SYMBOL) goto L1 ;

                        // Constraint on the matching of IDENTIFICATION symbols against
                        // CONTENTS symbols and vice versa.

                        if (id_c_symbol_constraint)
                        {
                                dr_status = symbol2->get_status() ;
                                t_status = t_class_symbol->get_status() ;
                                if (dr_status == IDENTIFICATION
                                        || dr_status == BOUNDARY_MARKER)
                                {
                                        if (t_status != CONTENTS) continue ;
                                }
                                else if (dr_status == CONTENTS)
                                {
                                        if (t_status != IDENTIFICATION
                                                && t_status != BOUNDARY_MARKER)
                                                        continue ;
                                }
                        }
```

```
                        if (symbol2->name_matches(t_class_symbol)) break ;
                }

                if (symbol2 == NIL) continue ;

                // Make sure that symbol2 does not contain t_class_symbol.

                if (anomalous_hit(symbol2, driving_pattern_depth,
                        t_class_symbol, 1))
                                return(false) ;

                // Look for matching RIGHT_BRACKET, taking account of the
                // (relative) depth of backetting.

                el_pos3 = el_pos2 ;
                depth_of_bracketting = 0 ;
                while (symbol3 = (symbol *)
                        driving_pattern->get_next_child_by_el_pos(&el_pos3))
                {
                        if (symbol3->get_type() == LEFT_BRACKET)
                                depth_of_bracketting++ ;
                        if (symbol3->get_type() == RIGHT_BRACKET)
                        {
                                if (depth_of_bracketting == 0)
                                        // We have a RIGHT_BRACKET at the right level.
                                        break ;
                                depth_of_bracketting-- ; // Carry on searching.
                        }
                }

                // We have a valid match for all three symbols.

                reference_structure.left_dr_bracket = symbol1 ;
                reference_structure.left_dr_bracket_int_pos =
                        el_pos1->get_position() ;
                reference_structure.dr_class_symbol = symbol2 ;
                reference_structure.dr_class_symbol_int_pos =
                        el_pos2->get_position() ;
                reference_structure.right_dr_bracket = symbol3 ;
                reference_structure.right_dr_bracket_int_pos =
                        el_pos3->get_position() ;
                reference_structure.dr_el_pos = el_pos3 ;
                return(true) ;

                L1: ;
        }
        return(false) ;
} // match_reference

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Called from compare_patterns_with_brackets(). This function finds
**        the next 'reference' in the target_pattern, meaning a left bracket
**        followed by a class symbol followed by a right bracket. The position
**        that searching starts is recorded in the global structure
**        reference_structure and that value is updated ready for the next
**        search if a reference is found. That structure also records the
**        values of the symbols that have been found.
**
** CALLING SEQUENCE:
**
**        bool find_reference()
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if a 'reference' is found, false otherwise.
**
** IMPLICIT INPUTS:
**
**        reference_structure
**
** IMPLICIT OUTPUTS:
**
**        reference_structure
**
** SIDE EFFECTS:
**
**        NONE
*/
```

```
bool find_reference()
{
        sequence *target_pattern = reference_structure.target_pattern ;
        list_element *el_pos1 = reference_structure.t_el_pos,
                *el_pos2, *el_pos3 ;
        symbol *symbol1, *symbol2, *symbol3 ;

        while (symbol1 = (symbol *)
                target_pattern->get_next_child_by_el_pos(&el_pos1))
        {
                if (symbol1->get_type() != LEFT_BRACKET) continue ;
                el_pos2 = el_pos1 ;
                symbol2 = (symbol *)
                        target_pattern->get_next_child_by_el_pos(&el_pos2) ;
                if (symbol2->get_type() != CONTEXT_SYMBOL) continue ;
                el_pos3 = el_pos2 ;
                symbol3 = (symbol *)
                        target_pattern->get_next_child_by_el_pos(&el_pos3) ;
                if (symbol3->get_type() != RIGHT_BRACKET) continue ;

                reference_structure.left_t_bracket = symbol1 ;
                reference_structure.left_t_bracket_int_pos =
                        el_pos1->get_position() ;
                reference_structure.t_class_symbol = symbol2 ;
                reference_structure.t_class_symbol_int_pos =
                        el_pos2->get_position() ;
                reference_structure.right_t_bracket = symbol3 ;
                reference_structure.right_t_bracket_int_pos =
                        el_pos3->get_position() ;
                reference_structure.t_el_pos = el_pos3 ;
                return(true) ;
        }

        reference_structure.t_el_pos = NIL ;
        return(false) ;
} // find_reference

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Matches one pattern against another when both patterns contain brackets.
**
** CALLING SEQUENCE:
**
**        void compare_patterns_with_brackets(sequence *driving_pattern,
**                sequence *target_pattern, sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:                void
**
**        driving_pattern:        The driving pattern.
**        target_pattern:                The target pattern.
**        cnp:                                The New pattern being processed.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void compare_patterns_with_brackets(sequence *driving_pattern,
        sequence *target_pattern, sequence *cnp)
{
        if (target_pattern->is_abstract_pattern() == false)
                return ;

        reference_structure.target_pattern = target_pattern ;
        reference_structure.driving_pattern = driving_pattern ;
        reference_structure.left_t_bracket = NIL ;
        reference_structure.t_class_symbol = NIL ;
        reference_structure.right_t_bracket = NIL ;
        reference_structure.left_dr_bracket = NIL ;
        reference_structure.dr_class_symbol = NIL ;
```

```
        reference_structure.right_dr_bracket = NIL ;
        reference_structure.t_el_pos = NIL ;
        reference_structure.dr_el_pos = NIL ;

        while (find_reference())
        {
                if (match_reference())
                {
                        master_driving_int_pos =
                                reference_structure.left_dr_bracket_int_pos ;
                        master_driving_pattern =
                                reference_structure.driving_pattern ;
                        master_driving_symbol =
                                reference_structure.left_dr_bracket ;
                        master_target_int_pos =
                                reference_structure.left_t_bracket_int_pos ;
                        master_target_pattern =
                                reference_structure.target_pattern ;
                        master_target_symbol =
                                reference_structure.left_t_bracket ;
                        hn_master->set_driving_int_pos(master_driving_int_pos) ;
                        hn_master->set_driving_pattern(master_driving_pattern) ;
                        hn_master->set_driving_symbol(master_driving_symbol) ;
                        hn_master->set_target_int_pos(master_target_int_pos) ;
                        hn_master->set_target_pattern(master_target_pattern) ;
                        hn_master->set_target_symbol(master_target_symbol) ;
                        update_hit_structure(cnp) ;

                        master_driving_int_pos =
                                reference_structure.dr_class_symbol_int_pos ;
                        master_driving_symbol =
                                reference_structure.dr_class_symbol ;
                        master_target_int_pos =
                                reference_structure.t_class_symbol_int_pos ;
                        master_target_symbol =
                                reference_structure.t_class_symbol ;
                        hn_master->set_driving_int_pos(master_driving_int_pos) ;
                        hn_master->set_driving_symbol(master_driving_symbol) ;
                        hn_master->set_target_int_pos(master_target_int_pos) ;
                        hn_master->set_target_symbol(master_target_symbol) ;
                        update_hit_structure(cnp) ;

                        master_driving_int_pos =
                                reference_structure.right_dr_bracket_int_pos ;
                        master_driving_symbol =
                                reference_structure.right_dr_bracket ;
                        master_target_int_pos =
                                reference_structure.right_t_bracket_int_pos ;
                        master_target_symbol =
                                reference_structure.right_t_bracket ;
                        hn_master->set_driving_int_pos(master_driving_int_pos) ;
                        hn_master->set_driving_symbol(master_driving_symbol) ;
                        hn_master->set_target_int_pos(master_target_int_pos) ;
                        hn_master->set_target_symbol(master_target_symbol) ;
                        update_hit_structure(cnp) ;
                }
        }

} // compare_patterns_with_brackets

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Matches a driving pattern with target patterns.
**
** CALLING SEQUENCE:
**
**        void match_driving_pattern(sequence *driving_pattern,
**                sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:               void
**
**        driving_pattern:        The driving pattern.
**        cnp:                            The New pattern being processed.
**
** IMPLICIT INPUTS:
**
**        NONE
**
```

```
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void match_driving_pattern(sequence *driving_pattern, sequence *cnp)
{
        list_element *el_pos1 ;
        sequence *target_pattern ;

        fprintf(output_file, "NEXT DRIVING PATTERN (") ;
        print_pattern_cycle(true, cnp) ;
        fprintf(output_file, ")\n\n") ;

        driving_pattern->write_pattern(true, true) ;

        if (driving_pattern->get_number_of_children() >= MCOLS)
                abort_run("MCOLS is too small.") ;

        hn_master->set_driving_pattern(driving_pattern) ;
        master_driving_pattern = driving_pattern ;

        // Take target patterns from old_patterns.

        el_pos1 = NIL ;
        list_for_el_pos(target_pattern, sequence, old_patterns, el_pos1)
        {
                if (target_pattern->get_number_of_children() >= MCOLS)
                        abort_run("MCOLS is too small.") ;

                hn_master->set_target_pattern(target_pattern) ;
                master_target_pattern = target_pattern ;

                if (cycle == 1)
                        compare_patterns(driving_pattern, target_pattern, cnp) ;
                else compare_patterns_with_brackets(driving_pattern,
                        target_pattern, cnp) ;
        }

        // In this version, it appears that there is no need to compare
        // the driving pattern with any of the parsing_alignments. Therefore,
        // this portion of code has been removed (to odds_and_ends.cpp).

} // match_driving_pattern

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Finds hit sequences between driving patterns and patterns in
**        Old and in alignments already formed.
**
** CALLING SEQUENCE:
**
**        bool find_best_matches(sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if one or more hit sequences are
**                                    formed. false otherwise.
**
**        cnp:                        The New pattern being processed.
**
** IMPLICIT INPUTS:
**
**        old_patterns
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

bool find_best_matches(sequence *cnp)
{
```

```
        sequence *driving_pattern ;
        list_element *el_pos1 ;

        if (cycle == 1)
        {
                driving_pattern = cnp ;
                match_driving_pattern(driving_pattern, cnp) ;
        }
        else
        {
                el_pos1 = NIL ;
                list_for_el_pos(driving_pattern, sequence,
                        parsing_alignments, el_pos1)
                {
                        if (driving_pattern->is_new_this_cycle() == false) continue ;
                                // We only want to use alignments that were newly created
                                // on the previous cycle.

                        if (driving_pattern->get_degree_of_matching() != PARTIAL
                                        // We need to allow FULL_A alignments to be tried because
                                        // some of them may be composite.
                                && driving_pattern->new_hits_form_coherent_sequence())
                                        match_driving_pattern(driving_pattern, cnp) ;
                }
        }

        if (hit_root->get_first_element() == NIL)
        {
                fprintf(output_file, "FOR ") ;
                print_pattern_cycle(true, cnp) ;
                fprintf(output_file,
                        ", NO HIT SEQUENCES HAVE BEEN FOUND.\n\n") ;
                return(false) ;
        }
        else if (show_hit_structure)
                write_hit_structure(hit_root, 0) ;

        return(true) ;

} /* find_best_matches */

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Writes the best alignment to latex_file in a form ready
**        for printing using latex.
**
** CALLING sequence:
**
**        void output_alignment(sequence *sequence1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        sequence1:        The sequence to be output.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void output_alignment(sequence *sequence1)
{
        int al_length_in_characters = 0, al_lines, i ;
        double target_font_height_in_points,
                actual_font_height_in_points, actual_font_width_in_points,
                actual_font_width_in_mm,
                baselineskip_in_points, base_line_skip_in_mm,
                printing_height, printing_width,
                al_actual_depth_in_mm,
                font_heights[FONT_SET_SIZE] =
```

```
{7, 8, 9, 10, 11, 12, 14, 17.28, 20.74, 24.88} ;

// double actual_font_height_in_mm ;

/* Note re fontsizes: there are 72.27 'points' to an inch which
is 2.8452756 points to 1 mm (assuming 1 inch = 25.4 mm). This
is defined in sp.h as PRINTING_POINTS_PER_MM.

Here are values to go in \fontsize{17.28pt}{20.74pt} in Latex files:

        \fontsize{07.00pt}{08.40pt}
        \fontsize{08.00pt}{09.60pt}
        \fontsize{09.00pt}{10.80pt}
        \fontsize{10.00pt}{12.00pt}
        \fontsize{12.00pt}{14.40pt}
        \fontsize{14.00pt}{16.80pt}
        \fontsize{17.28pt}{20.74pt}
        \fontsize{20.74pt}{24.89pt}
        \fontsize{24.88pt}{29.86pt}

        The second figure is 1.2 times the first figure.

        A 'standard' (12 pt?) character is 2.125 mm wide and 7.6 mm high.

        An A4 sheet of paper has dimensions as follows:

        Width: 210 mm. With margins of 25 mm, this gives a
        printing space of 160 mm (6.25 inch).

        Height: 297 mm. With margins of 25 mm, this gives a
        printing space of 247 mm (9.6 inch).

        For the two orientations, and assuming the letter size above,
        these figures mean:

        Portrait: 75.29 letters / line, 32.5 lines.

        Landscape: 116.24 letters / line, 21 lines.

        Font families: \fontfamily{family}, families include:
        cmr for Computer Modern Roman, cmss for Computer Modern Sans Serif,
        cmtt for Computer Modern Typewriter.

        Standard fontsizes in LaTeX are 7, 8, 9, 10, 11, 12, 14, 17.28, 20.74,
        and 24.88. The baselineskip_in_points (distance from the
        bottom of one line to the bottom of the next) is normally 1.2 times
        the point size. */

if (orientation == LANDSCAPE)
{
        printing_height = PRINTING_SPACE_SHORT_SIDE ;
        printing_width = PRINTING_SPACE_LONG_SIDE ;
}
else
{
        printing_height = PRINTING_SPACE_LONG_SIDE ;
        printing_width = PRINTING_SPACE_SHORT_SIDE ;
}

/* Find the length of the sequence in characters. 1 is
added to the length of each symbol except the last to represent
the space between symbols. */

al_length_in_characters = sequence1->compute_character_length() ;
al_lines = (sequence1->get_sequence_depth() * 2) - 1 ; // The number
// of lines in the printed alignment is twice the number
// of rows (to allow for an extra line between each row)
// less 1 because the extra line is not needed for the
// last row.

// Calculate a 'target' font size. This may then be adjusted in
// the light of available font sizes and maximum_font_height
// and minimum_font_height.

target_font_height_in_points = (printing_width /
        al_length_in_characters)
        * PRINTING_POINTS_PER_MM
        * CHARACTER_HEIGHT_WIDTH_RATIO ;

// Find available font size which is >= minimum_font_height
// and <= maximum_font_height and is nearest to the target_font size.
// Start by looking for the available font size which is nearest
// to the target font size.
```

```
double difference, smallest_difference = HIGH_VALUE ;
int index_best ;

for (i = 0; i < FONT_SET_SIZE; i++)
{
        if (target_font_height_in_points < font_heights[i])
                difference = font_heights[i] -
                target_font_height_in_points ;
        else difference = target_font_height_in_points -
                font_heights[i] ;
        if (difference < smallest_difference)
        {
                smallest_difference = difference ;
                index_best = i ;
        }
}

actual_font_height_in_points = font_heights[index_best] ;

if (actual_font_height_in_points < minimum_font_height)
        actual_font_height_in_points = minimum_font_height ;

if (actual_font_height_in_points > maximum_font_height)
        actual_font_height_in_points = maximum_font_height ;

actual_font_width_in_points = actual_font_height_in_points /
        CHARACTER_HEIGHT_WIDTH_RATIO ;

// Calculated the distance between the bottom of one line and
// the bottom of the next line below.

baselineskip_in_points = actual_font_height_in_points * 1.2 ;

// Convert measurements in points to measurements in mm.

// actual_font_height_in_mm = actual_font_height_in_points /
// PRINTING_POINTS_PER_MM ;
actual_font_width_in_mm = actual_font_width_in_points /
        PRINTING_POINTS_PER_MM ;
base_line_skip_in_mm = baselineskip_in_points / PRINTING_POINTS_PER_MM ;

fprintf(output_file, "%s%1.2f%s",
        "actual_font_height_in_points = ",
        actual_font_height_in_points,
        " pt\n\n") ;

// Calculate how many character columns of an alignment can
// be fitted across the printing_width without, at this stage,
// allowing for the row number printed at the beginning and end
// of each row. This adjustment will be made in write_alignment.
// Most sections will be shorter than the calculated number of
// columns because alignments will only be split at blank columns.

int write_section_chars_length_latex = (int)
        ceil(printing_width / actual_font_width_in_mm) ;

// Calculate the number of sections of the alignment can
// be fitted into the printing_height.

al_actual_depth_in_mm = base_line_skip_in_mm * (al_lines + 2) ;
// al_lines is increased by 2 to allow for the 2 blank
// lines between one section and the next or between
// the last section and the "Figure xxx" label. No attempt
// has been made to allow for this bottom line because
// there are likely to be few occasions when there is
// not a spare line to accommodate it.

int sections_per_page = (int)floor(printing_height /
        al_actual_depth_in_mm) ;

fprintf(latex_file, "%s%s%s%s%s%s",
        "\\documentclass{article}\n",
        "\\textheight 254mm\n",
        "\\textwidth 177.8mm\n",
        "\\topmargin -20mm\n",
        "\\oddsidemargin 0mm\n",
        "\\evensidemargin 0mm\n") ;


if (orientation == LANDSCAPE)
        fprintf(latex_file, "\\usepackage{lscape}\n") ;
```

```
        fprintf(latex_file, "\\begin{document}\n") ;

        if (orientation == LANDSCAPE)
                fprintf(latex_file, "\\begin{landscape}\n") ;

        fprintf(latex_file, "%s%1.2f%s%1.2f%s",
                "\\fontsize{",
                actual_font_height_in_points,
                "pt}{",
                baselineskip_in_points,
                "pt}\n") ;

        fprintf(latex_file, "%s%s",
                "\\pagestyle{empty}\n",
                "\\begin{verbatim}\n") ;

        sequence1->write_alignment_horizontal(latex_file,
                write_section_chars_length_latex,
                sections_per_page) ;

        fprintf(latex_file, "%s%s%s",
                "\nFigure ",
                figure_ID,
                "\n") ;

        fprintf(latex_file, "\\end{verbatim}\n") ;

        if (orientation == LANDSCAPE)
                fprintf(latex_file, "\\end{landscape}\n") ;

        fprintf(latex_file, "\\end{document}\n") ;

        fflush(latex_file) ;
} /* output_alignment */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Compares two double variables.
**
** CALLING SEQUENCE:
**
**        bool is_better_than(double A, double B)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if A is better than B, false otherwise
**
**        A, B:                        The two variables to be compared.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

bool is_better_than(double A, double B)
{
        if (A < B) return(true) ;
        else return(false) ;
} // is_better_than

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Calculates the logarithm to the base 2 according to the formula
**        given in Knuth's "The Art of Computer Programming", vol 2,
**        page 23.
**
** CALLING SEQUENCE:
**
```

```
**        double log_2(double x)
**
** FORMAL ARGUMENTS:
**
**        Return value:        The required log to the base 2 of x.
**
**        x:                               The number whose logarithm is to be calculated.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

double log_2(double x)
{
       return(log10(x) / log_10_2) ;
} // log_2

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Looks up a symbol name in key_array[] to find the corresponding
**        abbreviation. If no abbreviation can be found, the original
**        string is returned. Any symbol beginning with '#' is
**        returned directly.
**
** CALLING SEQUENCE:
**
**        char *find_symbol_abbreviation(char *symbol_name_1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        A character string for the abbreviation.
**
**        symbol_name_1:        The name whose abbreviation is to be found.
**
** IMPLICIT INPUTS:
**
**        key_array[]
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

char *find_symbol_abbreviation(char *symbol_name_1)
{
        int key_index ;

        for (key_index = 0; key_index < fe_key_array; key_index++)
        {
                if (strcmp(symbol_name_1,
                        key_array[key_index].complete_symbol) == 0)
                                return(key_array[key_index].abbreviated_symbol) ;
        }

        // If this point is reached, then no match was found for symbol_name_2.
        // In this case, the function returns symbol_name_1.

        return(symbol_name_1) ;
} // find_symbol_abbreviation

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
```

```
**        In write_alignment, fills in the non-hit symbols before a hit
**        symbol.
**
** CALLING SEQUENCE:
**
**        bool fill_in_non_hit_symbols(int s_row1, symbol *hit_symbol,
**                int *fe_col)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if the write_al_chars_length has been breached.
**                                     false otherwise.
**
**        s_row1:         The relevant row in row_array[].
**        hit_symbol: The hit symbol which terminates the sequence of
**                                symbols. If hit_symbol == NIL then all the trailing
**                                symbols for each row are added to the alignment.
**        fe_col:         The first empty column in the given row of
**                                write_al_chars[][] when the last non-hit symbol
**                                before the hit_symbol has been filled in.
**
** IMPLICIT INPUTS:
**
**        arrays for write_alignment.
**
** IMPLICIT OUTPUTS:
**
**        arrays for write_alignment.
**
** SIDE EFFECTS:
**
**        NONE
**
*/

bool fill_in_non_hit_symbols(int s_row1, symbol *hit_symbol, int *fe_col)
{
        symbol *symbol1 ;
        bool char_column_not_OK ;
        sequence *pattern1 = row_array[s_row1].row_pattern ;
        int end_col, c_col1, c_col2, slot_width, c_row1 = s_row1 * 2 ;
        char *c, *symbol_name ;

        pattern1->set_current_el_pos(row_array[s_row1].patt_last_el_pos) ;
        c_col1 = row_array[s_row1].fe_col_in_write_al_chars ;
        while (symbol1 = (symbol *)pattern1->get_next_child())
        {
                if (symbol1 == hit_symbol) break ;

                if (use_abbreviations == LETTERS || use_abbreviations == DIGITS)
                        symbol_name = find_symbol_abbreviation(symbol1->
                        get_name()) ;
                else symbol_name = symbol1->get_name() ;

                // Find the next available space in this row of
                // write_al_chars[][] to take the symbol name plus
                // space for one space character.

                slot_width = strlen(symbol_name) + 1 ; // slot_width
                // is the enough space for the symbol name
                // plus 1 character for the space following the symbol.

                char_column_not_OK = true ;
                while (char_column_not_OK)
                {
                        char_column_not_OK = false ;
                        end_col = c_col1 + slot_width ;
                        for (c_col2 = c_col1; c_col2 < end_col; c_col2++)
                        {
                                if (write_al_vacant_slots[s_row1][c_col2] ==
                                        'x')
                                {
                                        char_column_not_OK = true ;
                                        c_col1 = c_col2 + 1;
                                        break ;
                                }
                        }
                }

                // Check that we are still within the bounds
                // of write_al_chars[][].

                if (c_col1 + slot_width >= write_al_chars_length)
```

```
                {
                        // The writing out of this row of the alignment
                        // will be truncated at this point, with "..." to
                        // show that there is more to be written.

                        *fe_col = c_col1 ;
                        return(true) ;
                }
                else
                {
                        // Fill in the symbol name.

                        for (c = symbol_name; *c != '\0'; c++)
                                write_al_chars[c_row1][c_col1++] = *c ;
                }

                // Add 1 to c_col to allow for the space character following
                // the symbol.

                c_col1++ ;
        }

        // Test to see whether the end of the pattern has been reached.

        if (symbol1 == NIL) row_array[s_row1].pattern_is_finished = true ;

        // Update the value of patt_last_el_pos for this row. At this
        // point, there is no need to update the value of
        // fe_col_in_write_al_chars because this will be updated
        // in write_alignment after the hit symbol has been written.

        row_array[s_row1].patt_last_el_pos = pattern1->get_current_el_pos() ;

        *fe_col = c_col1 ;
        return(false) ;

} // fill_in_non_hit_symbols

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Partitions sort_array[]. (Used by sort_leaf_array()).
**
** CALLING SEQUENCE:
**
**        void partition(int low, int high, int *mid)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        low:                The 'low' position in sort_array[].
**        high:                The 'high' position in sort_array[].
**        mid:                A pointer to the 'mid' position in sort_array[]
**                                (in the calling function).
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void partition(int low, int high, int *mid)
{
        int left_ptr, right_ptr, temp_num ;
        double pivot_score ;
        hit_node *h_node ;

        left_ptr = low ;
        right_ptr = high ;
        while (left_ptr < right_ptr)
        {
        /* Search for the rightmost element greater than
        h_node->get_compression_difference() for the 'low' position
```

```
        in leaf_array[] (which is the 'pivot'), starting from
        the right. */

                h_node = leaf_array[sort_array[low]] ;
                pivot_score = h_node->get_compression_difference() ;
                h_node = leaf_array[sort_array[right_ptr]] ;
                while (h_node->get_compression_difference() < pivot_score)
                        h_node = leaf_array[sort_array[--right_ptr]] ;

                /* Seach for the leftmost element greater than the
                pivot score, starting from the left. */

                h_node = leaf_array[sort_array[left_ptr]] ;
                while (left_ptr < right_ptr
                        && (h_node->get_compression_difference() >=
                        pivot_score))
                        h_node = leaf_array[sort_array[++left_ptr]] ;

                /* If the pointers have not crossed, switch the values. */

                if (left_ptr < right_ptr)
                {
                        temp_num = sort_array[left_ptr] ;
                        sort_array[left_ptr] = sort_array[right_ptr] ;
                        sort_array[right_ptr] = temp_num ;
                }
        }
        *mid = right_ptr ;
        temp_num = sort_array[*mid] ;
        sort_array[*mid] = sort_array[low] ;
        sort_array[low] = temp_num ;
} /* partition */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Sorts rows of leaf_array[] by compression_ratio in leaf_node.
**        This version uses the Quicksort method.
**
** CALLING SEQUENCE:
**
**        void sort_leaf_array(int low, int high) ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        low:                        The 'low' position in sort_array[].
**        high:                        The 'high' position in sort_array[].
**
** IMPLICIT INPUTS:
**
**        sort_array[] and leaf_array[]
**
** IMPLICIT OUTPUTS:
**
**        sort_array[] and leaf_array[]
**
** SIDE EFFECTS:
**
**        NONE
*/

void sort_leaf_array(int low, int high)
{
        int mid ;

        if (low < high)
        {
                partition(low, high, &mid) ; /* Partition sort_array[] */
                sort_leaf_array(low, mid - 1) ; /* Sort the lower portion */
                sort_leaf_array(mid + 1, high) ; /* Sort the upper portion */
        }
} /* sort_leaf_array */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Makes new alignments.
```

```
**
** CALLING SEQUENCE:
**
**        bool make_alignments(sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if one or more new sequences
**                                   are formed, false otherwise.
**
**        cnp:                        The New pattern being processed.
**
** IMPLICIT INPUTS:
**
**        Global data structures
**
** IMPLICIT OUTPUTS:
**
**        New unifications
**
** SIDE EFFECTS:
**
**        NONE
*/

bool make_alignments(sequence *cnp)
{
        sequence *al1, *al2 ;
        hit_node *node1 ;
        int hit_node_row, leaf_index ;
        bool new_alignment_formed = false ;

        fprintf(output_file, "STARTING make_alignments (") ;
        print_pattern_cycle(true, cnp) ;
        fprintf(output_file, ")\n\n") ;
        fflush(output_file) ;

        number_of_parsing_alignments = 0 ;

        sort_leaf_array(0, fe_sort - 1) ; // Sort using approximate,
                // cheap-to-compute scores.

        hit_node_row = 0 ;
        while (node1 = get_leaf_nodes_in_order(&hit_node_row, &al1,
                &leaf_index))
        {
                if (node1->get_compression_difference() <= fail_score)
                        continue ;

                /* Make a new sequence and, if it is not NIL and
                does not match a sequence formed on this cycle and all
                earlier cycles, add it to alignments_array[]. */

                // Scores are re-computed for each new alignment
                // that is formed.

                al2 = unify_best(leaf_index, cnp) ;

                if (al2 == NIL) continue ;

                if (show_unselected_alignments)
                {
                        fprintf(output_file, "NEW ALIGNMENT (") ;
                        print_pattern_cycle(true, cnp) ;
                        fprintf(output_file, ")\n\n") ;
                        al2->print_ID() ;
                        fprintf(output_file, " : ") ;
                        (node1->get_driving_pattern())->print_ID() ;
                        fprintf(output_file, " : ") ;
                        (node1->get_target_pattern())->print_ID() ;
                        fprintf(output_file, " : ") ;
                        node1->print_ID() ;
                        fprintf(output_file,
                                "%s%1.2f%s%1.2f%s%1.2f%s%1.2f%s%1.2f%s",
                                " NSC = ", al2->get_new_symbols_cost(),
                                ", EC = ", al2->get_encoding_cost(),
                                ", CR = ", al2->get_compression_ratio(),
                                ", CD = ", al2->get_compression_difference(),
                                ",\nAbsolute P = ", al2->get_abs_P(),
                                "\n\n") ;

                        al2->write_alignment(output_file,
                                write_section_chars_length, NULL_VALUE,
```

```
                                alignment_format) ;

                        fflush(output_file) ;
                }

                new_alignment_formed = true ;

                number_of_parsing_alignments++ ;

                if (number_of_parsing_alignments > max_alignments_in_one_cycle)
                {
                        fprintf(output_file, "%s%s%d%s",
                                "Formation of alignments cut short by ",
                                "MAX_ALIGNMENTS_IN_ONE_CYCLE (",
                                max_alignments_in_one_cycle,
                                ")\n\n") ;
                        break ;
                }
        }

        // Resort using re-computed scores.

        sort_leaf_array(0, fe_sort - 1) ;

        fprintf(output_file, "FINISHED make_alignments (") ;
        print_pattern_cycle(true, cnp) ;
        fprintf(output_file, ")\n\n") ;
        fflush(output_file) ;

        return(new_alignment_formed) ;

} /* make_alignments */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**
**
** CALLING SEQUENCE:
**
**        bool max_unsupported_cycles_reached(sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if, in this pattern from New, there have been
**                                   more than max_unsupported_cycles since the
**                                   best CD value for the pattern was reached.
**                                   false otherwise.
**
**        cnp:                        The current New pattern.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

bool max_unsupported_cycles_reached(sequence *cnp)
{
        if (cycle - best_cycle_this_new_pattern >= max_unsupported_cycles)
        {
                // We have reached the set limit on the number of
                // cycles which may be performed without any gain
                // in compression score.

                fprintf(output_file, "FOR ") ;
                print_pattern_cycle(true, cnp) ;
                fprintf(output_file, "%s%d%s",
                        ", THE MAXIMUM NUMBER OF UNSUPPORTED CYCLES (",
                        max_unsupported_cycles,
                        ") HAS BEEN REACHED\n\n") ;

                return(true) ;
```

```
        }
        else return(false) ;
} // max_unsupported_cycles_reached

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Compresses a sequence from New by repeated applications of
**        find_best_matches.
**
** CALLING SEQUENCE:
**
**        bool recognise(sequence *pattern1)
**
** FORMAL ARGUMENTS:
**
**        Return value:                        true if unifiable matchings have been found,
**                                                    false otherwise.
**
**        pattern1:                                A new pattern or a derived pattern to be
**                                                    recognised.
**
** IMPLICIT INPUTS:
**
**        old_patterns
**
** IMPLICIT OUTPUTS:
**
**        old_patterns
**
** SIDE EFFECTS:
**
**        NONE
*/

bool recognise(sequence *pattern1)
{
        cycle++ ;
        bool matches_have_been_found ;

        fprintf(output_file, "START OF ") ;
        print_pattern_cycle(true, pattern1) ;
        fprintf(output_file, ":\n\n") ;

        set_up_hit_structure() ;

        matches_have_been_found = find_best_matches(pattern1) ;

        // Now that the values of 'new_this_cycle' in parsing_alignments
        // from the previous cycle have served their purpose in controlling
        // which alignments are used as 'driving' alignments in this cycle,
        // they may all be set to false.

        sequence *pattern2 ;
        list_for(pattern2, sequence, parsing_alignments)
                pattern2->set_new_this_cycle(false) ;

        if (phase == 1 && cycle == 1)
        {
                receptacle_pattern->add_ID_symbols(NIL, true) ; // The receptacle
                        // pattern is a 'top level' pattern so it needs a
                        // unique ID symbol.
                fprintf(output_file, "Receptacle pattern: ") ;
                receptacle_pattern->write_with_details(true) ;
        }

        if (matches_have_been_found == false)
        {
                delete hit_root ;
                return(false) ;
        }

        if (show_unselected_alignments)
        {
                fprintf(output_file,
                        "SET OF ALIGNMENTS FORMED, BEFORE SELECTION, IN ") ;
                print_pattern_cycle(true, pattern1) ;
                fprintf(output_file, "\n\n") ;
                fflush(output_file) ;
        }
```

```
if (make_alignments(pattern1) == false)
{
        fprintf(output_file,"FOR ") ;
        print_pattern_cycle(true, pattern1) ;
        fprintf(output_file,
                "\nNO NEW ALIGNMENTS HAVE BEEN FOUND.\n\n") ;

        fprintf(output_file, "END OF ") ;
        print_pattern_cycle(true, pattern1) ;
        fprintf(output_file, "\n\n") ;

        delete hit_root ; // This deletes the whole tree of hit
                // nodes, recursively.

        return(false) ;
}

make_selection(pattern1) ;

// Determine which of FULL_A, FULL_B or PARTIAL alignments
// have been found.

sequence *al1 ;
bool partial_al_found = false, full_A_al_found = false,
        full_B_al_found = false ;
int type_of_alignment ;

list_for(al1, sequence, parsing_alignments)
{
        type_of_alignment = al1->get_degree_of_matching() ;
        if (type_of_alignment == FULL_A) full_A_al_found = true ;
        if (type_of_alignment == FULL_B) full_B_al_found = true ;
        if (type_of_alignment == PARTIAL) partial_al_found = true ;
}

parsing_alignments->sort_by_compression_difference() ;

if (full_B_al_found)
        combine_alignments(pattern1) ;

parsing_alignments->sort_by_ID() ;

// Print ID numbers of alignments.

fprintf(output_file, "%s%s%s",
        "In the following lists, alignments that are new on this cycle\n",
        "are marked with an asterisk ('*').\n\n",
        "SELECTED ALIGNMENTS FROM ") ;
print_pattern_cycle(true, pattern1) ;
fprintf(output_file, "\n\n") ;

parsing_alignments->write_IDs(NULL_VALUE, pattern1) ;

fflush(output_file) ;

if (full_A_al_found)
{
        fprintf(output_file, "SELECTED FULL ALIGNMENTS (A) FROM ") ;
        print_pattern_cycle(true, pattern1) ;
        fprintf(output_file, "\n\n") ;

        parsing_alignments->write_IDs(FULL_A, pattern1) ;
}
else
{
        fprintf(output_file, "NO FULL_A ALIGNMENTS FROM ") ;
        print_pattern_cycle(true, pattern1) ;
        fprintf(output_file, "\n\n") ;
}

if (full_B_al_found)
{
        fprintf(output_file, "SELECTED FULL ALIGNMENTS (B) FROM ") ;
        print_pattern_cycle(true, pattern1) ;
        fprintf(output_file, "\n\n") ;

        parsing_alignments->write_IDs(FULL_B, pattern1) ;
}
else
{
        fprintf(output_file, "NO FULL_B ALIGNMENTS FROM ") ;
        print_pattern_cycle(true, pattern1) ;
        fprintf(output_file, "\n\n") ;
```

```
        }

        if (partial_al_found)
        {
                fprintf(output_file, "SELECTED PARTIAL ALIGNMENTS FROM ") ;
                print_pattern_cycle(true, pattern1) ;
                fprintf(output_file, "\n\n") ;
                parsing_alignments->write_IDs(PARTIAL, pattern1) ;
        }
        else
        {
                fprintf(output_file, "NO PARTIAL ALIGNMENTS FROM ") ;
                print_pattern_cycle(true, pattern1) ;
                fprintf(output_file, "\n\n") ;
        }

        delete hit_root ; // This deletes the whole tree of hit
                // nodes, recursively.

        fprintf(output_file, "END OF ") ;
        print_pattern_cycle(true, pattern1) ;
        fprintf(output_file, "\n\n") ;

        fflush(output_file) ;

        if (max_unsupported_cycles_reached(pattern1)
                || (full_A_al_found == false && full_B_al_found == false))
                        return(false) ;

        return(true) ;
} /* recognise */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Called from find_containing_sequences.
**
** CALLING SEQUENCE:
**
**        bool contained_in(sequence *sequence1, sequence *test_sequence)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if sequence1 is contained
**                                   in test_sequence,
**                                   false otherwise.
**
**        sequence1:               The given sequence.
**        test_sequence:        The sequence being tested to see whether it
**                                   contains the given sequence or not.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

#if CONTAINED_IN

bool contained_in(sequence *sequence1, sequence *test_sequence)
{
        symbol *B_symbol1, *al_col2 ;
        sequence *pattern_of_B_symbol1, *pattern_of_al_col2 ;
        symbol *first_symbol ;
        int col1, col2, number_of_children, test_number_of_children,
                sequence_depth = 1, test_sequence_depth = 1,
                ra_index, fe_rows_array = 0, i, offset ;
        struct rows_entry
        {
                symbol *B_symbol1, *al_col2 ;
                sequence *pattern_of_B_symbol1, *pattern_of_al_col2 ;
                bool match_found ;
        } rows_array[MEDIUM_SCRATCH_ARRAY_SIZE] ;
```

```
bool match_found, original_pattern_match_found ;

if (sequence1 == NIL || test_sequence == NIL)
        abort_run("NIL parameter(s) for contained_in") ;
if (typeof(sequence1) != typeof(test_sequence)) return(false) ;
if (typeof(sequence) == SYMBOL || typeof(test_sequence) == SYMBOL)
        abort_run("SYMBOL parameter(s) for contained_in") ;

/* Compare the lengths of the sequences. */

number_of_children = sequence1->number_of_children ;
test_number_of_children = test_sequence->number_of_children ;

if (number_of_children > test_number_of_children) return(false) ;

/* Find sequence depth. */

sequence1->initialise() ;
while (B_symbol1 = (symbol *)sequence1->get_next_child())
        sequence_depth++ ;

/* Find test_sequence_depth and compare with sequence_depth. */

test_sequence->initialise() ;
while (B_symbol1 = (symbol *)test_sequence->get_next_child())
        test_sequence_depth++ ;

if (sequence_depth > test_sequence_depth) return(false) ;

/* Initialise rows_array[]. */

for (ra_index = 0; ra_index < MEDIUM_SCRATCH_ARRAY_SIZE; ra_index++)
{
        rows_array[ra_index].B_symbol1 = NIL ;
        rows_array[ra_index].pattern_of_B_symbol1 = NIL ;
        rows_array[ra_index].al_col2 = NIL ;
        rows_array[ra_index].pattern_of_al_col2 = NIL ;
        rows_array[ra_index].match_found = false ;
}

/* Now fill in values for rows_array[]. The original pattern
of each Old row in sequence1 is compared with the original_pattern
each Old row in test sequence1. If there are no matches,
this means that sequence1 is not contained in
test_sequence. Where there are one or more matches,
these are entered in rows_array[]. */

sequence1->initialise() ;
B_symbol1 = (symbol *)sequence1->get_next_child() ;
while (B_symbol1 = (symbol *)sequence1->get_next_child())
{
        original_pattern_match_found = false ;
        test_sequence->initialise() ;
        B_symbol1 = (symbol *)test_sequence->get_next_child() ;
        while (B_symbol1 = (symbol *)
                test_sequence->get_next_child())
        {
                if (B_symbol1->original_pattern != al_col2->el_obj)
                        continue ;

                original_pattern_match_found = true ;

                /* We have a positive match of the original_patterns.
                Now record it in rows_array[]. */

                rows_array[fe_rows_array].B_symbol1 = B_symbol1 ;
                rows_array[fe_rows_array].pattern_of_B_symbol1 =
                        B_symbol1->el_obj ;
                rows_array[fe_rows_array].al_col2 = al_col2 ;
                rows_array[fe_rows_array].pattern_of_al_col2 =
                        al_col2->el_obj ;
                fe_rows_array = plus_one(fe_rows_array,
                        MEDIUM_SCRATCH_ARRAY_SIZE,
                        "rows_array[] too small in contained_in") ;
        }
        if (original_pattern_match_found == false) return(false) ;
}

/* Now check to see whether the pattern of hits for each Old row
of sequence1 matches a pattern of hits (for the same original_pattern)
in an Old row of test_sequence. The order of matching rows need not
be the same in the two sequences. It is not safe to assume that the
two sequences have the same length: if test_sequence is longer
```

```
than sequence1, it is not safe to assume that the pattern of hits
in any one row in sequence1 starts at the same symbol as
test_sequence. Likewise for the end of a pattern of hits.
If the symbols do not coincide, then, for each row, symbols
in test_sequence which are outside the pattern of hits must all
be NIL. */

/* Find an Old row in sequence1 where the first symbol in the
row is not NIL. */

sequence1->initialise() ;
B_symbol1 = (symbol *)sequence1->get_next_child() ;
while (B_symbol1 = (symbol *)sequence1->get_next_child())
{
        if (B_symbol1->symbol_sequence[0] != NIL)
        {
                pattern_of_B_symbol1 = B_symbol1->el_obj ;
                first_symbol = B_symbol1->symbol_sequence[0] ;
                break ;
        }
}

/* Now find row(s) in test_sequence with the same original_pattern and
find the position in each of these rows of the first symbol of
the original_pattern. From the one which starts furthest to the left,
we can find the offset of the start of sequence1
relative to the start of test_sequence. */

pattern_of_al_col2 = NIL ;
offset = 1000000 ;
test_sequence->initialise() ;
B_symbol1 = (symbol *)test_sequence->get_next_child() ;
while (B_symbol1 = (symbol *)test_sequence->get_next_child())
{
        if (al_col2->el_obj == pattern_of_B_symbol1)
        {
                for (col2 = 0; col2 < test_number_of_children; col2++)
                {
                        if (al_col2->symbol_sequence[col2] !=
                                first_symbol) continue ;
                        if (col2 < offset) offset = col2 ;
                        break ;
                }
                if (col2 >= test_number_of_children)
                        abort_run("Error in contained_in") ;
        }
}

if (offset >= test_number_of_children)
        abort_run("Error in contained_in") ;

ra_index = 0 ;
sequence1->initialise() ;
B_symbol1 = (symbol *)sequence1->get_next_child() ;
while (B_symbol1 = (symbol *)sequence1->get_next_child())
{
        match_found = false ;
        while (rows_array[ra_index].B_symbol1 == B_symbol1)
        {
                al_col2 = rows_array[ra_index].al_col2 ;

                /* Check that this al_col2 has
                not already been 'used'. */

                for (i = 0; i < ra_index; i++)
                        if (rows_array[i].al_col2 == al_col2
                                && rows_array[i].match_found)
                                        goto L1 ;

                /* Step over the offset in al_col2, checking that
                it contains nothing other than NIL values. */

                for (col2 = 0; col2 < offset; col2++)
                        if (al_col2->symbol_sequence[col2] != NIL)
                                goto L1 ;

                /* Now compare the pattern of hits in B_symbol1 and
                al_col2. */

                for (col1 = 0; col1 < number_of_children; col1++)
                {
                        if (col2 >= test_number_of_children) goto L1 ;
                        if (B_symbol1->symbol_sequence[col1] !=
```

```
                                        al_col2->symbol_sequence[col2])
                                                goto L1 ;
                                col2++ ;
                        }

                        /* Check that any trailing symbols in al_col2 are
                        all NIL. */

                        for (col2 = col2; col2 < test_number_of_children; col2++)
                                if (al_col2->symbol_sequence[col2] != NIL)
                                        goto L1 ;

                        match_found = true ;
                        rows_array[ra_index].match_found = true ;
                        while (rows_array[ra_index].B_symbol1 == B_symbol1)
                                if (++ra_index >= fe_rows_array) break ;

                        break ;

                        L1: ;
                        if (++ra_index >= fe_rows_array) break ;
                }
                if (!match_found) return(false) ;
        }
        return(true) ;
} /* contained_in */

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        For each of the sequences in best_sequences[], this function
**        compiles a list of the other best sequences, if any, which
**        contain the given sequence.
**
** CALLING SEQUENCE:
**
**        void find_containing_sequences()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        best_sequences[]
**
** IMPLICIT OUTPUTS:
**
**        For each best_sequence[], a list of containing sequences which
**        may be NIL.
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void find_containing_sequences()
{
        int i, j ;

        for (i = 0; i < fe_best_sequences; i++)
        {
                for (j = 0; j < fe_best_sequences; j++)
                {
                        if (i == j) continue ;
                        if (contained_in(best_sequences[i].al_el,
                                best_sequences[j].al_el))
                        {
                                fprintf(output_file, "Alignment ") ;
                                best_sequences[i].al_el->print_ID() ;
                                fprintf(output_file, " is contained in ") ;
                                best_sequences[j].al_el->print_ID() ;
                                fprintf(output_file, "\n\n") ;
                        }
                }
        }
} /* find_containing_sequences */

#endif
```

```
/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Fetches symbols by row order and symbol order from an sequence,
**        excluding any symbols in the first row (which represents New)
**        which are not aligned with any symbol in Old. Each symbol type
**        is returned only once, ie second and subsequent instances of a symbol
**        type are skipped.
**
** CALLING SEQUENCE:
**
**        symbol *get_next_symbol_from_sequence(sequence *al1, int status)
**
** FORMAL ARGUMENTS:
**
**        Return value:        The next symbol type in the sequence.
**
**        al1:                        The sequence from which symbol types are fetched.
**        status:                If START, the sequence is searched from the
**                                        beginning, otherwise it continues from where it
**                                        was previously.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

symbol *get_next_symbol_from_sequence(sequence *al1, int status)
{
        static symbol *B_symbol1 ;
        static alignment_element *al_el1 ;
        static symbol *symbol_array[MEDIUM_SCRATCH_ARRAY_SIZE] ;
        symbol *symbol1 ;
        int i, row1, sequence_depth = al1->get_sequence_depth() ;

        if (al1 == NIL) abort_run("Invalid value for al1 in \
                get_next_symbol_from_sequence") ;

        if (status == START)
        {
                // Initialise symbol_array[] (which is used for checking for
                // duplicates in symbols returned).

                for (i = 0; i < MEDIUM_SCRATCH_ARRAY_SIZE; i++)
                        symbol_array[i] = NIL ;
                return(NIL) ;
        }

        al1->initialise() ;
        while (B_symbol1 = (symbol *)al1->get_next_child())
        {
                for (row1 = 0; row1 < sequence_depth; row1++)
                {
                        al_el1 = B_symbol1->get_al_el(row1) ;
                        symbol1 = (symbol *)al_el1->get_el_obj() ;
                        if (symbol1 == NIL) continue ;

                        // Check whether al_el1 is at the top of a symbol.
                        // If it is below the top of a symbol,
                        // skip to next symbol.

                        if (al_el1->get_same_column_above() != NULL_VALUE)
                                break ;

                        // Check whether al_el1 is in the first row
                        // (corresponding to New) and is also not part part
                        // of any hit. If so, skip.

                        if (row1 == 0 && al_el1->get_same_column_below() ==
                                NULL_VALUE)
                                break ;
```

```
                        // Now check whether a matching symbol has been
                        // retrieved from this sequence before. If it has,
                        // skip to next symbol. Otherwise add symbol1 to
                        // symbol_array[] and return it as the next symbol.

                        for (i = 0; i < MEDIUM_SCRATCH_ARRAY_SIZE; i++)
                        {
                                if (symbol_array[i] == NIL)
                                {
                                        symbol_array[i] = symbol1 ;
                                        return(symbol1) ;
                                }

                                if (strcmp(symbol1->get_name(),
                                        symbol_array[i]->get_name()) == 0)

                                        // There is no need to
                                        // continue down this symbol
                                        // because only symbols of
                                        // the same type will be found.

                                        goto L1 ;
                        }

                        // If the program reaches this point, it means that
                        // symbol_array[] is too small.

                        abort_run("symbol_array[] is too small in \
                                get_next_symbol_from_sequence") ;

                }
                L1: ;
        }

        return(NIL) ;
} // get_next_symbol_from_sequence

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks whether an alignment contains one or more redundant rows.
**
**        A row is redundant if it is in Old and if every symbol in it
**        forms a hit with a symbol in one other row in Old (see sp61_od, %49).
**        No attempt is made to check for rows which may be redundant
**        because they form hits across two or more rows in Old.
**
**        If one or more rows are redundant, the alignment is edited to remove
**        the redundant rows and the function returns true. Otherwise it
**        returns false.
**
** CALLING SEQUENCE:
**
**        bool check_for_redundant_rows(sequence *al1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if al1 has been edited. false otherwise.
**
**        al1:                        The alignment to be checked.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

bool check_for_redundant_rows(sequence *al1)
{
        // Check each row in the alignment against every other row.
        // A row is redundant if:
        // * Every symbol in the row is a hit symbol.
        // * There is at least one other row which forms a hit with
```

```
//        every symbol of the given row.
//
// The second condition means that the other row must be at least
// as long as the given row. If they are equal in size, then one
// of them must be chosen arbitrarily as the redundant row.

symbol *al_col1, *al_col2,
       *new_alignment_array[LARGE_SCRATCH_ARRAY_SIZE] ;
alignment_element *al_el1 ;
int row1, row2, al1_depth = al1->get_sequence_depth(),
       al1_length = al1->get_number_of_children(),
       this_row_pattern_length, that_row_pattern_length,
       this_row_int_pos, temp_row_int_pos ;
bool redundant_rows_array[MEDIUM_SCRATCH_ARRAY_SIZE], // Used to
       // record whether a row is redundant and is to
       // be deleted (true) or is to be preserved (false).
       that_row_matches_this_row[MEDIUM_SCRATCH_ARRAY_SIZE]; // Used
// to record when 'that' row matches 'this' row.
sequence *this_row_pattern, *that_row_pattern ;
list_element *this_el_pos_start, *pos1 ;

if (al1_depth >= MEDIUM_SCRATCH_ARRAY_SIZE)
abort_run("redundant_rows_array[] is too small in \
check_for_redundant_rows") ;
if (al1_length >= LARGE_SCRATCH_ARRAY_SIZE)
abort_run("new_alignment_array[] is too short in \
       check_for_redundant_row") ;

redundant_rows_array[0] = false ; // Make sure that the first
       // row (New) is preserved in all circumstances.


for (row1 = 1; row1 < al1_depth; row1++)
{
       redundant_rows_array[row1] = false ; // By default, each row
       // is to be preserved, not edited out.
       this_row_pattern = al1->get_row_pattern(row1) ;
       this_row_pattern_length =
              this_row_pattern->get_number_of_children() ;

       // Search for the first non-NIL entry in row1.

       this_el_pos_start = NIL ;
       while (al_col1 = (symbol *)
              al1->get_next_child_by_el_pos(&this_el_pos_start))
       {
              al_el1 = al_col1->get_al_el(row1) ;
              if (al_el1->get_el_obj() != NIL) break ;
       }

       // Check whether the first hit symbol for this row is
       // the first symbol in the pattern for this row. If it
       // is not, then this row cannot be contained in any other
       // row.

       this_row_int_pos = al_el1->get_orig_patt_int_pos() ;
       if (this_row_int_pos != 0) continue ;

       // Now step along 'this' row, comparing it with every other
       // row except row 0. If there are any unmatched symbols in
       // 'this' row, break out of the loop because 'this' row
       // cannot be completely matched to any other row.
       // Whenever a mis-match appears between 'that'
       // row and 'this' row, a false value is entered into
       // that_row_matches_this_row[].

       // Initialise that_row_matches_this_row[].
       // Check the lengths of the patterns in each
       // row against the length of the pattern for 'this' row.
       // If a row contains a pattern that is shorter than the
       // pattern in 'this' row, then 'this' row cannot be
       // completely matched against it.

       for (row2 = 1; row2 < al1_depth; row2++)
       {
              if (row2 == row1)
              {
                     that_row_matches_this_row[row2] = false ;
                     continue ;
              }

              // Compare lengths of patterns.
```

```
        that_row_pattern = al1->get_row_pattern(row2) ;
        that_row_pattern_length =
                that_row_pattern->get_number_of_children() ;
        if (that_row_pattern_length < this_row_pattern_length)
        {
                that_row_matches_this_row[row2] = false ;
                continue ;
        }

        // Check whether there is a symbol in al_col1 in
        // this row.

        al_el1 = al_col1->get_al_el(row2) ;
        if (al_el1->get_el_obj() == NIL)
                that_row_matches_this_row[row2] = false ;
        else that_row_matches_this_row[row2] = true ;
}

// At this point, we know that this_row_int_pos == 0.

pos1 = this_el_pos_start ;
while (al_col1 = (symbol *)
        al1->get_next_child_by_el_pos(&pos1))
{
        al_el1 = al_col1->get_al_el(row1) ;

        // Skip over any column which does not contain
        // a symbol for row1.

        if (al_el1->get_el_obj() == NIL) continue ;

        // Check whether there are any unmatched symbols
        // in 'this' row between the current column and
        // the last one. If there is, 'this' row cannot
        // be completely contained in any other.

        temp_row_int_pos = al_el1->get_orig_patt_int_pos() ;
        if (temp_row_int_pos - this_row_int_pos > 1) break ;

        // 'this' row contains a symbol and there are no
        //        unmatched symbols preceding the current symbol
        // in this row.

        this_row_int_pos = temp_row_int_pos ;

        // Now examine all the other rows for this column.
        // If there is no symbol or an unmatched symbol
        // in any row then it cannot be a matching row
        // for 'this' row. Also, there is no point checking
        // rows which are already known to be false.

        for (row2 = 1; row2 < al1_depth; row2++)
        {
                if (that_row_matches_this_row[row2] == false)
                        continue ;

                if (row2 == row1)
                {
                        that_row_matches_this_row[row2] =
                                false ;
                        continue ;
                }

                // Check whether there is a symbol in
                // al_col1 in this row.

                al_el1 = al_col1->get_al_el(row2) ;
                if (al_el1->get_el_obj() == NIL)
                        that_row_matches_this_row[row2] =
                        false ;
                else that_row_matches_this_row[row2] = true ;
        }
}

// Now examine that_row_matches_this_row[] to see if
// there are any rows that are true. If there are, this
// means that row1 is redundant and this can be entered
// into redundant_rows_array[].

// If Old contains duplicate
// patterns (which is possible but unlikely), then there
// may be two rows which match each other completely and
// are the same length. It is assumed here that Old does
```

```
        // not contain duplicate patterns.

        for (row2 = 1; row2 < al1_depth; row2++)
                if (that_row_matches_this_row[row2] == true) break ;

                if (row2 < al1_depth)
                {
                        // This means that there is at least one row
                        // which matches row1 completely.

                        redundant_rows_array[row1] = true ;
                }
}

// Now check to see whether any rows need to be edited out and
// do it where necessary.

int new_al_depth = al1_depth ;
for (row1 = 0; row1 < al1_depth; row1++)
        if (redundant_rows_array[row1] == true) new_al_depth-- ;

        if (new_al_depth == al1_depth) return(false) ;

        // al1 is now to be edited. Do this by replacing each column by
        // a new column. The 'shell' of al1 is preserved to avoid upsetting
        // any pointers to it which may exist in the global data structure.

        alignment_element *al_el2 ;
        int col1 = -1 ;
        al1->initialise() ;
        while (al_col1 = (symbol *)al1->get_next_child())
        {
                al_col2 = new symbol(al_col1->get_name(), new_al_depth,
                        NULL_VALUE) ;
                plus_one(&col1, al1_length, "col1 is too large in \
                        check_for_redundant_rows") ;
                new_alignment_array[col1] = al_col2 ;
                row2 = -1 ;
                for (row1 = 0; row1 < al1_depth; row1++)
                {
                        if (redundant_rows_array[row1] == true) continue ;
                        // This row is to be omitted from the
                        // edited version of al1.

                        // This row is to be preserved.

                        plus_one(&row2, new_al_depth,
                                "row2 is too large in check_for_redundant_rows") ;

                        al_el1 = al_col1->get_al_el(row1) ;
                        al_el2 = al_col2->get_al_el(row2) ;
                        *al_el2 = *al_el1 ;
                }
        }

        // Delete all the columns in al1.

        while (al_col1 = (symbol *)al1->get_first_child())
        {
                al1->extract_child(al_col1) ;
                delete al_col1 ;
        }

        // Now put in the new columns currently stored in
        // new_alignment_array[].

        for (col1 = 0; col1 < al1_length; col1++)
                al1->add_child(new_alignment_array[col1]) ;

        // Re-set sequence_depth in al1.

        al1->set_sequence_depth(new_al_depth) ;

        // Re-set values in each column of al1.

        al1->initialise() ;
        int symbol_count ;
        while (al_col1 = (symbol *)al1->get_next_child())
        {
                al_col1->set_sequence_depth(new_al_depth) ;

                // Does this column contain at least one hit?
```

```
                        symbol_count = 0 ;
                        for (row1 = 0; row1 < new_al_depth; row1++)
                        {
                                al_el1 = al_col1->get_al_el(row1) ;
                                if (al_el1->get_el_obj() != NIL)
                                        symbol_count++ ;
                        }

                        if (symbol_count > 1) al_col1->set_is_a_hit(true) ;
                        else al_col1->set_is_a_hit(false) ;

                        // For each columnm, put the correct values in for
                        // same_column_above and same_column_below.

                        for (row1 = 0; row1 < new_al_depth; row1++)
                        {
                                al_el1 = al_col1->get_al_el(row1) ;
                                if (al_el1->get_el_obj() == NIL) continue ;
                                for (row2 = row1 + 1; row2 < new_al_depth; row2++)
                                {
                                        al_el2 = al_col1->get_al_el(row2) ;
                                        if (al_el2->get_el_obj() != NIL) break ;
                                }
                                if (row2 >= new_al_depth) break ;
                                al_el1->set_same_column_below(row2) ;
                                al_el2->set_same_column_above(row1) ;
                        }
                }

                return(true) ;

} // check_for_redundant_rows

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        If there is a best sequence, this function calculates the
**        probabilities of inferences from the best sequence and all
**        other sequences which encode the same symbols from New.
**
**        The function calculates 'Absolute P' which is the absolute probability
**        of each sequence (derived directly from its 'old symbols cost')
**        and 'Relative P' which is the probability of each sequence relative
**        to other sequences which encode the same symbols from New.
**
**        The function also calculates the probabilities of patterns and
**        symbols within the sequences as described in sp_ideas6, %37.
**        For each pattern found anywhere amongst the set of sequences
**        which encode the same symbols from New, the function adds up
**        the values of Small p for the sequences in which it is found,
**        giving a value of finding at least one instance of the pattern
**        in an alignment. Likewise for individual symbols.
**
** CALLING SEQUENCE:
**
**        void probabilities_of_inferences(sequence *best_alignment)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        best_alignment: The best sequence found at the end of compression.
**
** IMPLICIT INPUTS:
**
**        Global variables
**
** IMPLICIT OUTPUTS:
**
**        Prints all sequences which code the same symbols from New as
**        the best sequence and calculates probabilities of the
**        associated inferences.
**
** SIDE EFFECTS:
**
**        NONE
*/

void probabilities_of_inferences(sequence *best_alignment)
{
        symbol *B_symbol1 ;
```

```
alignment_element *al_el1 ;
sequence *pattern ;
symbol *new_hit_symbol, *symbol1, *symbol2, *hit_symbol ;
group *new_hits = new group() ; /* This is
        the head of a list of symbols from New
        which have formed hits in the best_alignment. */
sequence *al1, *al2 ;
struct best_alignment_entry *sequence_entry ;
struct patterns_entry
{
        sequence *pattern ;
        double probability ;
        bool in_current_sequence, done ;
} patterns_in_sequences[MEDIUM_SCRATCH_ARRAY_SIZE] ;

struct symbols_entry
{
        symbol *symbol1 ;
        double probability ;
        bool in_current_sequence, done ;
} symbols_in_sequences[MEDIUM_SCRATCH_ARRAY_SIZE] ;

double total_abs_P ;
int i, j, k, fe_p_in_als, fe_s_in_als, row1,
        sequence_depth ;
bool sequence_found, symbol_found ;

if (best_alignment == NIL)
{
        fprintf(output_file,
                "NO GOOD ALIGNMENTS OR INFERENCES FOUND\n\n") ;
        return ;
}

fe_best_alignments = 0 ;

best_alignments[fe_best_alignments].al1 = best_alignment ;
best_alignments[fe_best_alignments].contained_in = NIL ;
fe_best_alignments++ ;

/* Compile symbols from New which are hit_symbols. */

best_alignment->initialise() ;
while (B_symbol1 = (symbol *)best_alignment->get_next_child())
{
        al_el1 = B_symbol1->get_al_el(0) ;
        if (al_el1->get_same_column_below() == NULL_VALUE) continue ;

        hit_symbol = (symbol *)al_el1->get_el_obj() ;
        new_hits->add_child(hit_symbol) ;
}

/* Now look for sequences (other than best_alignment), which
have the same hit_symbols in New, neither more nor less.
For each one found, check for redundant rows and remove them
(see sp61_od, %49). For each alignment which is edited in this
way, check to see that it is not identical to one already in
the reference set of alignments. If it is, do not add it to the
set. */

bool alignment_is_OK ;
sequence *modified_alignments[MEDIUM_SCRATCH_ARRAY_SIZE] ;
int fe_modified_alignments = 0, modified_alignments_index ;

list_for(al1, sequence, parsing_alignments)
{
        if (al1 == best_alignment) continue ;

        new_hits->initialise() ;
        new_hit_symbol = (symbol *)new_hits->get_next_child() ;
        al1->initialise() ;
        while (B_symbol1 = (symbol *)al1->get_next_child())
        {
                al_el1 = B_symbol1->get_al_el(0) ;
                if (al_el1->get_el_obj() == NIL) continue ;
                if (al_el1->get_same_column_below() == NULL_VALUE)
                        continue ;

                hit_symbol = (symbol *)al_el1->get_el_obj() ;
                if (hit_symbol != new_hit_symbol)
                        /* There is a mis-match. */
                        goto L1 ;
```

```
                new_hit_symbol = (symbol *)new_hits->get_next_child() ;
        }

        if (new_hit_symbol != NIL) continue ;

        /* All the symbols in new_hits have been matched
        and none others - which means that this
        sequence is OK.

        Check to see whether the alignment contains
        any redundant rows. If it has, remove it or them, then place
        the modified alignment in modified_alignments[] to
        be checked later to see whether it duplicates any other
        alignment in parsing_alignments. */

        if (check_for_redundant_rows(al1))
        {
                fprintf(output_file, "%s%d%s",
                        "Alignment %",
                        al1->get_ID(),
                        " has been edited.\n\n") ;
                al1->write_alignment(output_file,
                        write_section_chars_length, NULL_VALUE,
                        alignment_format) ;
                modified_alignments[fe_modified_alignments] = al1 ;
                plus_one(&fe_modified_alignments,
                        MEDIUM_SCRATCH_ARRAY_SIZE,
                        "modified_alignments[] is too small") ;
        }
        else
        {
                best_alignments[fe_best_alignments].al1 = al1 ;
                plus_one(&fe_best_alignments,
                        MEDIUM_SCRATCH_ARRAY_SIZE,
                        "best_alignments[] too small.") ;
        }

        L1: ;
}

for (modified_alignments_index = 0; modified_alignments_index <
        fe_modified_alignments; modified_alignments_index++)
{
        al1 = modified_alignments[modified_alignments_index] ;
        alignment_is_OK = true ;
        list_for(al2, sequence, parsing_alignments)
        {
                if (al2 == al1) continue ;
                if (al2->alignment_matches(al1))
                {
                        if (verbose)
                        {
                                fprintf(output_file, "Alignment ") ;
                                al1->print_ID() ;
                                fprintf(output_file, " matches sequence ") ;
                                al2->print_ID(),
                                fprintf(output_file, " and is discarded.\n\n") ;
                                fflush(output_file) ;
                        }

                        parsing_alignments->extract_child(al1) ;
                        delete al1 ;
                        alignment_is_OK = false ;
                        break ;
                }
        }

        if (alignment_is_OK)
        {
                fprintf(output_file, "%s%d%s",
                        "After editing, alignment %",
                        al1->get_ID(),
                        " does not match any earlier alignment\n\n") ;
                al1->make_code(false) ;
                fprintf(output_file, "%s%d%s",
                        "New scores for alignment %",
                        al1->get_ID(),
                        " are:\n\n") ;
                fprintf(output_file,
                        "%s%1.2f%s%1.2f%s%1.2f%s%1.2f%s%1.12g%s",
                        "NSC = ", al1->get_new_symbols_cost(),
                        ", EC = ", al1->get_encoding_cost(),
                        ", CR = ", al1->get_compression_ratio(),
```

```
                                ", CD = ", al1->get_compression_difference(),
                                ",\nAbsolute P = ", al1->get_abs_P(),
                                "\n\n") ;

                        best_alignments[fe_best_alignments].al1 = al1 ;
                        plus_one(&fe_best_alignments,
                                MEDIUM_SCRATCH_ARRAY_SIZE,
                                "best_alignments[] too small.") ;
                }
        }

#if CONTAINED_IN

        /* For each sequence in best_alignments[], compile a list of the
        other best sequences which contain it, if any. */

        find_containing_sequences() ;

#endif

        /* Calculate total_abs_P. */

        total_abs_P = 0 ;
        for (i = 0; i < fe_best_alignments; i++)
        {
                al1 = best_alignments[i].al1 ;
                total_abs_P += al1->get_abs_P() ;
        }

        /* Calculate rel_P for each of the alignments in best_alignments[].
        Then write out the alignments and probabilities in order of
        relative probability. */

        fprintf(output_file, "%s%s%1.12g%s",
                "BEST ALIGNMENTS AND PROBABILITIES\n\n",
                "Total absolute P = ",
                total_abs_P, "\n\n") ;
        for (i = 0; i < fe_best_alignments; i++)
        {
                sequence_entry = &best_alignments[i] ;
                al1 = sequence_entry->al1 ;
                sequence_entry->rel_P = al1->get_abs_P() / total_abs_P ;
                sequence_entry->done = false ; // To be used in printing.
        }

        double biggest_rel_P, temp_rel_P ;
        int index_of_biggest ;

        while (true)
        {
                biggest_rel_P = NULL_VALUE ;
                index_of_biggest = NULL_VALUE ;
                for (i = 0; i < fe_best_alignments; i++)
                {
                        sequence_entry = &best_alignments[i] ;
                        if (sequence_entry->done == true) continue ;
                        temp_rel_P = sequence_entry->rel_P ;
                        if (temp_rel_P > biggest_rel_P)
                        {
                                biggest_rel_P = temp_rel_P ;
                                index_of_biggest = i ;
                        }
                }

                if (index_of_biggest == NULL_VALUE) break ;
                sequence_entry = &best_alignments[index_of_biggest] ;
                sequence_entry->done = true ;
                al1 = sequence_entry->al1 ;

                fprintf(output_file, "ALIGNMENT:\n\n") ;
                sequence_entry->al1->print_ID() ;
                fprintf(output_file,
                        "%s%1.2f%s%1.2f%s%1.2f%s%1.2f%s%1.12g%s%1.12g%s",
                        ", NSC = ",
                        al1->get_new_symbols_cost(),
                        ", EC = ",
                        al1->get_encoding_cost(),
                        ", CR = ",
                        al1->get_compression_ratio(),
                        ", CD = ",
                        al1->get_compression_difference(),
                        ",\nAbsolute P = ", al1->get_abs_P(),
                        ", Relative P = ", sequence_entry->rel_P,
```

```
                "\n\n") ;

        al1->write_alignment(output_file,
                write_section_chars_length, NULL_VALUE,
                alignment_format) ;
}

/* Compile a set of patterns each of which appears at least
once in one of best_alignments[]. At the same time, calculate
the probability of each pattern as the sum of the rel_P
probabilities of the sequences in which it appears. */

for (i = 0; i < MEDIUM_SCRATCH_ARRAY_SIZE; i++)
{
        patterns_in_sequences[i].pattern = NIL ;
        patterns_in_sequences[i].probability = 0 ;
        patterns_in_sequences[i].in_current_sequence = false ;
        patterns_in_sequences[i].done = false ;
}

fe_p_in_als = 0 ;

for (i = 0; i < fe_best_alignments; i++)
{
        al1 = best_alignments[i].al1 ;

        /* Initialise values of in_current_sequence in
        patterns_in_sequences[]. */

        for (j = 0; j < fe_p_in_als; j++)
                patterns_in_sequences[j].in_current_sequence =
                false ;

        B_symbol1 = (symbol *)al1->get_first_child() ;
        al1->set_current_el_pos(al1->get_first_el_pos()) ;
        sequence_depth = al1->get_sequence_depth() ;

        /* Exclude the first row of the sequence, which
        corresponds to the cnp. */

        for (row1 = 1; row1 < sequence_depth; row1++)
        {
                pattern = B_symbol1->get_row_pattern(row1) ;

                /* Check patterns_in_sequences[] to see if there
                is already an entry for pattern. If there is,
                use that entry, otherwise make a new entry. */

                sequence_found = false ;
                for (k = 0; k < fe_p_in_als; k++)
                {
                        if (patterns_in_sequences[k].pattern ==
                                pattern)
                        {
                                sequence_found = true ;
                                break ;
                        }
                }

                if (!sequence_found)
                {
                        patterns_in_sequences[fe_p_in_als].pattern =
                                pattern ;

                        k = fe_p_in_als ;
                        plus_one(&fe_p_in_als,
                                MEDIUM_SCRATCH_ARRAY_SIZE,
                                "patterns_in_sequences[] \
                                is too small") ;
                }

                /* Now add to the probability value for this
                pattern if it has not already been done in
                this sequence. */

                if (patterns_in_sequences[k].in_current_sequence ==
                        true)
                        continue ;

                patterns_in_sequences[k].in_current_sequence = true ;
                patterns_in_sequences[k].probability +=
                        best_alignments[i].rel_P ;
        }
```

```
}

fprintf(output_file, "PROBABILITIES OF PATTERNS:\n\n") ;

double biggest_probability, temp_probability ;

while (true)
{
        biggest_probability = NULL_VALUE ;
        index_of_biggest = NULL_VALUE ;
        for (i = 0; i < fe_p_in_als; i++)
        {
                if (patterns_in_sequences[i].done == true) continue ;
                temp_probability =
                        patterns_in_sequences[i].probability ;
                if (temp_probability > biggest_probability)
                {
                        biggest_probability = temp_probability ;
                        index_of_biggest = i ;
                }
        }

        if (index_of_biggest == NULL_VALUE) break ;
        patterns_in_sequences[index_of_biggest].done = true ;

        pattern = patterns_in_sequences[index_of_biggest].pattern ;
        pattern->print_ID() ;
        fprintf(output_file, "%s%1.12g%s",
                ", p = ",
                patterns_in_sequences[index_of_biggest].probability,
                " ") ;

        pattern->write_tree_object(PRINT_SEQUENCE_FREQUENCY) ;
}

/* Compile a set of symbols each of which appears at least
once in one of best_alignments[]. At the same time, calculate
the probability of each symbol as the sum of the rel_P
probabilities of the sequences in which it appears. */

for (i = 0; i < MEDIUM_SCRATCH_ARRAY_SIZE; i++)
{
        symbols_in_sequences[i].symbol1 = NIL ;
        symbols_in_sequences[i].probability = 0 ;
        symbols_in_sequences[i].in_current_sequence = false ;
        symbols_in_sequences[i].done = false ;
}

fe_s_in_als = 0 ;

for (i = 0; i < fe_best_alignments; i++)
{
        al1 = best_alignments[i].al1 ;

        /* Initialise values of in_current_sequence in
        symbols_in_sequences[]. */

        for (j = 0; j < fe_s_in_als; j++)
                symbols_in_sequences[j].in_current_sequence =
                false ;

        get_next_symbol_from_sequence(al1, START) ;
        while (symbol2 = get_next_symbol_from_sequence(al1, CONTINUE))
        {
                /* Check symbols_in_sequences[] to see if there
                is already an entry for symbol2. If there is,
                use that entry, otherwise make a new entry. */

                symbol_found = false ;
                for (k = 0; k < fe_s_in_als; k++)
                {
                        symbol1 = symbols_in_sequences[k].symbol1 ;
                        if (strcmp(symbol1->get_name(),
                                symbol2->get_name()) == 0)
                        {
                                symbol_found = true ;
                                break ;
                        }
                }

                if (!symbol_found)
                {
                        symbols_in_sequences[fe_s_in_als].symbol1 =
```

```
                                        symbol2 ;

                                k = fe_s_in_als ;
                                plus_one(&fe_s_in_als,
                                        MEDIUM_SCRATCH_ARRAY_SIZE,
                                        "symbols_in_alingments[] \
                                        is too small") ;
                        }

                        /* Now add to the probability value for this
                        symbol if it has not already been done in
                        this sequence. */

                        if (symbols_in_sequences[k].in_current_sequence ==
                                true)
                                continue ;

                        symbols_in_sequences[k].in_current_sequence = true ;
                        symbols_in_sequences[k].probability +=
                                best_alignments[i].rel_P ;
                }
        }

        fprintf(output_file, "PROBABILITIES OF SYMBOLS:\n\n") ;

        while (true)
        {
                biggest_probability = NULL_VALUE ;
                index_of_biggest = NULL_VALUE ;
                for (i = 0; i < fe_s_in_als; i++)
                {
                        if (symbols_in_sequences[i].done == true) continue ;
                        temp_probability =
                                symbols_in_sequences[i].probability ;
                        if (temp_probability > biggest_probability)
                        {
                                biggest_probability = temp_probability ;
                                index_of_biggest = i ;
                        }
                }

                if (index_of_biggest == NULL_VALUE) break ;
                symbols_in_sequences[index_of_biggest].done = true ;

                symbol2 = symbols_in_sequences[index_of_biggest].symbol1 ;
                fprintf(output_file, "%s%s%1.12g%s",
                        symbol2->get_name(), ", p = ",
                        symbols_in_sequences[index_of_biggest].probability,
                        " ") ;
                fprintf(output_file, "\n") ;
        }

        fprintf(output_file, "\n") ;

        new_hits->release_children() ;
        delete new_hits ;
} /* probabilities_of_inferences */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Using existing markings of New or Old symbols with IDENTIFICATION
**        status, this function compiles a list of symbol types that have
**        that status, excluding LEFT_BRACKET and RIGHT_BRACKET. It
**        then applies the list to patterns in Old and New,
**        marking all symbols that appear on the list as being type
**        CONTEXT_SYMBOL. All other symbols are marked
**        as type DATA_SYMBOL.
**
** CALLING SEQUENCE:
**
**        void assign_type_to_symbols()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        Patterns in Old.
**
```

```
** IMPLICIT OUTPUTS:
**
**        Marking of symbols in Old and New with type CONTEXT_SYMBOL or DATA_SYMBOL.
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void assign_type_to_symbols()
{
        group *css = new group ;
        symbol *symbol1, *symbol2, *symbol3 ;
        sequence *pattern1 ;
        char c ;

        // Compile a list of symbol types which are of CONTEXT_SYMBOL type.
        // It is assumed that these symbol types are the types of the
        // symbols in New or Old that have IDENTIFICATION status. The
        // symbols '<' and '>' are, slightly redundantly, given the type
        // LEFT_BRACKET and RIGHT_BRACKET respectively.

        list_for(pattern1, sequence, old_patterns)
        {
                list_for(symbol1, symbol, pattern1)
                {
                        c = *(symbol1->get_name()) ;
                        if (symbol1->get_status() != IDENTIFICATION) continue ;

                        // if (c == '<' || c == '>') continue ;

                        // Check whether the type of symbol1 is already registered
                        // in css. If it is, continue. If not, add a copy of symbol1
                        // to this list.

                        list_for(symbol2, symbol, css)
                                if (symbol2->name_matches(symbol1)) goto L1 ;

                        // No matching symbol type has been found. Add a copy
                        // of symbol1 to css.

                        symbol3 = symbol1->shallow_copy() ;
                        css->add_child(symbol3) ;

                        L1: ;
                }
        }

        list_for(pattern1, sequence, new_patterns)
        {
                list_for(symbol1, symbol, pattern1)
                {
                        c = *(symbol1->get_name()) ;
                        if (symbol1->get_status() != IDENTIFICATION) continue ;

                        // if (c == '<' || c == '>') continue ;

                        // Check whether the type of symbol1 is already registered
                        // in css. If it is, continue. If not, add a copy of symbol1
                        // to the list.

                        list_for(symbol2, symbol, css)
                                if (symbol2->name_matches(symbol1)) goto L2 ;

                        // No matching symbol type has been found. Add a copy
                        // of symbol1 to css.

                        symbol3 = symbol1->shallow_copy() ;
                        css->add_child(symbol3) ;

                        L2: ;
                }
        }

        // Traverse the New patterns marking all symbols as DATA symbols.

        list_for(pattern1, sequence, new_patterns)
        {
                list_for(symbol1, symbol, pattern1)
                        symbol1->set_type(DATA_SYMBOL) ;
        }
```

```
        // Now traverse Old, marking symbols as type
        // CONTEXT_SYMBOL if a copy appears in css,
        // and marking them as type DATA_SYMBOL if a copy does not appear
        // on the list. The symbols '<' and '>' are marked as
        // LEFT_BRACKET and RIGHT_BRACKET respectively.

        list_for(pattern1, sequence, old_patterns)
        {
                list_for(symbol1, symbol, pattern1)
                {
                        if (*(symbol1->get_name()) == '<')
                        {
                                symbol1->set_type(LEFT_BRACKET) ;
                                continue ;
                        }

                        if (*(symbol1->get_name()) == '>')
                        {
                                symbol1->set_type(RIGHT_BRACKET) ;
                                continue ;
                        }

                        // Check whether a copy of symbol1 appears
                        // in css. If it does, mark symbol1 as
                        // the type CONTEXT_SYMBOL. If symbol1 does not appear
                        // on the list, mark it as a DATA_SYMBOL.

                        if (css->contains_copy_of(symbol1))
                                symbol1->set_type(CONTEXT_SYMBOL) ;
                        else symbol1->set_type(DATA_SYMBOL) ;
                }
        }

        // Delete css and all its children.

        delete css ;

} // assign_type_to_symbols

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        For each pattern in Old, identifies BOUNDARY_MARKERs, IDENTIFICATION
**        symbols and CONTENTS symbols. Finds the coding cost of each
**        pattern in Old as described in sp70_od, %3 - total of the
**        bit_costs for the 'coding' symbols for each pattern.
**
**        The IDENTIFICATION symbols for any pattern are: the one
**        or more symbols at the start of the pattern which are needed to
**        identify the pattern uniquely amongst the patterns in the corpus plus
**        the termination symbol for the pattern (if any) but excluding
**        brackets at the start and end of the pattern.
**
**        The status of symbols in New are either the default status of
**        CONTENTS, or they are explicitly marked in the input, or they
**        acquired their status in a previous iteration of
**        created_patterns_and_grammars().
**
** CALLING SEQUENCE:
**
**        void assign_status_to_symbols()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void assign_status_to_symbols()
```

```
{
        sequence *pattern1, *matching_pattern ;
        symbol *symbol1, *symbol2 ;
        bool end_of_leading_code_sequence_found ;
        char c, *name ;
        list_element *el_pos1 ;

        list_for(pattern1, sequence, new_patterns)
        {
                list_for(symbol1, symbol, pattern1)
                        symbol1->set_status(IDENTIFICATION) ;
        }

        list_for(pattern1, sequence, old_patterns)
        {
                end_of_leading_code_sequence_found = false ;
                el_pos1 = NIL ;
                list_for_el_pos(symbol1, symbol, pattern1, el_pos1)
                {
                        if (symbol1->name_is("<"))
                        {
                                symbol2 = (symbol *)pattern1->get_first_child() ;
                                if (symbol1 == symbol2) continue ;
                        }

                        symbol1->set_status(IDENTIFICATION) ;
                        if (is_unique(pattern1, symbol1, &matching_pattern))
                        {
                                end_of_leading_code_sequence_found = true ;
                                break ;
                        }
                }

                if (!end_of_leading_code_sequence_found)
                {
                        fprintf(output_file, "FIRST PATTERN: ") ;
                        pattern1->print_ID() ;
                        fprintf(output_file, "\n") ;
                        fprintf(output_file, "SECOND PATTERN: ") ;
                        matching_pattern->print_ID() ;
                        fprintf(output_file, "\n\n") ;
                        abort_run("Non-unique pattern found in old_patterns") ;
                }

                // Now mark the remaining symbols in pattern1 as CONTENTS,
                // except the last symbol if its contents character string
                // begins with '#' or is '>'.

                while (symbol1 = (symbol *)pattern1->get_next_child())
                {
                        if (pattern1->this_is_last_child(symbol1))
                        {
                                name = symbol1->get_name() ;
                                c = *name ;
                                if (c == '#' || strcmp(name, ">") == 0)
                                {
                                        symbol1->set_status(IDENTIFICATION) ;
                                        break ;
                                }
                        }
                        symbol1->set_status(CONTENTS) ;
                }
        }
} // assign_status_to_symbols

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Using the frequencies and costs of symbol types recorded in
**        set_of_symbols, this function assigns these values to
**        instances of symbols in the corpus.
**
** CALLING SEQUENCE:
**
**        void assign_symbol_frequencies_and_costs(group *set_of_symbols)
**
** FORMAL ARGUMENTS:
**
**        Return value:                void
**
**        set_of_symbols:                The alphabet of symbol types (each with values
```

```
**                                          for frequency and bit_cost) that
**                                          will be used in this function. This will either
**                                          be original_symbols_in_corpus or it will be
**                                          symbol_types_in_old.
**
** IMPLICIT INPUTS:
**
**        original_symbols_in_corpus
**
** IMPLICIT OUTPUTS:
**
**        frequency values of symbols in old_patterns
**
** SIDE EFFECTS:
**
**        NONE
*/

void assign_symbol_frequencies_and_costs(group *set_of_symbols)
{
        sequence *pattern1 ;
        symbol *symbol_type, *symbol1 ;

        /* Run through all symbols in the corpus and, for each one,
        look for the matching symbol type in set_of_symbols. When
        a match is found, give the symbol in the corpus the frequency
        of the matching symbol type in set_of_symbols and the costs of
        that symbol type. */

        fprintf(output_file,
                "NEW FREQUENCIES AND COSTS ASSIGNED TO PATTERNS AND SYMBOLS:\n\n") ;

        list_for(pattern1, sequence, new_patterns)
        {
                list_for(symbol1, symbol, pattern1)
                {
                        list_for(symbol_type, symbol, set_of_symbols)
                        {
                                if (strcmp(symbol_type->get_name(),
                                        symbol1->get_name()) != 0)
                                                continue ;
                                symbol1->set_frequency(symbol_type->
                                        get_frequency()) ;
                                symbol1->set_bit_cost(symbol_type->
                                        get_bit_cost()) ;
                                break ;
                        }
                }
        }

        list_for(pattern1, sequence, old_patterns)
        {
                list_for(symbol1, symbol, pattern1)
                {
                        list_for(symbol_type, symbol, set_of_symbols)
                        {
                                if (strcmp(symbol_type->get_name(),
                                        symbol1->get_name()) != NIL)
                                        continue ;
                                symbol1->set_frequency(symbol_type->
                                        get_frequency()) ;
                                symbol1->set_bit_cost(symbol_type->
                                        get_bit_cost()) ;
                                break ;
                        }
                }
        }

} /* assign_symbol_frequencies_and_costs */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        This function finds the frequencies of symbol types by adding
**        the frequency of the pattern in which it appears to the
**        symbol type for each appearance of a symbol type in the corpus.
**
** CALLING SEQUENCE:
**
**        void find_symbol_frequencies(group *symbol_set, bool do_new,
**                bool do_old, sequence *cnp)
```

```
**
** FORMAL ARGUMENTS:
**
**        Return value:                        void
**
**        symbol_set:                                The alphabet of symbols whose frequencies
**                                                        are to be set.
**        do_new:                                   Include new_patterns in the count.
**        do_old:                                   Include old_patterns in the count.
**        cnp:               The last New pattern to be processed in
**                                                        the main program. If the value is NIL,
**                                                        all New patterns are to be processed.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        frequencies of symbols recorded in symbol_set.
**
** SIDE EFFECTS:
**
**        NONE
*/

void find_symbol_frequencies(group *symbol_set, bool do_new,
        bool do_old, sequence *cnp)
{
        symbol *symbol1 ;

        /* Initialise frequencies of symbols in symbol_set. */

        symbol_set->initialise() ;
        while (symbol1 = (symbol *)symbol_set->get_next_child())
                symbol1->set_frequency(0) ;

        if (do_new) new_patterns->compile_frequencies(symbol_set, cnp) ;

        if (do_old) old_patterns->compile_frequencies(symbol_set, NIL) ;

} /* find_symbol_frequencies */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Write out New and Old patterns for reference.
**
** CALLING SEQUENCE:
**
**        void write_new_and_old()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        Data structures
**
** IMPLICIT OUTPUTS:
**
**        Output to output_file.
**
** SIDE EFFECTS:
**
**        NONE
*/

void write_new_and_old()
{
        /* Identify alignments in New and Old and write them out for reference,
        with the coding costs for patterns in Old. */

        fprintf(output_file, "PATTERNS IN NEW:\n\n") ;
        new_patterns->initialise() ;
        if (new_patterns->is_empty()) fprintf(output_file, "NONE\n\n") ;
        else new_patterns->write_patterns_with_details() ;
        fprintf(output_file, "%s%d%s",
                "Size of New (symbols): ", number_of_symbols_in_new, "\n") ;
        fprintf(output_file, "%s%d%s",
```

```
                "Size of New (patterns): ", number_of_patterns_in_new, "\n\n") ;

        fprintf(output_file, "ORIGINAL PATTERNS IN OLD:\n\n") ;
        old_patterns->initialise() ;
        if (old_patterns->is_empty()) fprintf(output_file, "NONE\n\n") ;
        else old_patterns->write_patterns_with_details() ;
        fprintf(output_file, "%s%d%s",
                "Size of Old (symbols): ", number_of_symbols_in_old, "\n") ;
        fprintf(output_file, "%s%d%s",
                "Size of Old (patterns): ", original_number_of_patterns_in_old, "\n\n") ;

        fflush(output_file) ;

} /* write_new_and_old */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Called from find_containing_sequences.
**
** CALLING SEQUENCE:
**
**        bool contained_in(sequence *sequence1, sequence *test_sequence)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if sequence1 is contained
**                                   in test_sequence,
**                                   false otherwise.
**
**        sequence1:               The given sequence.
**        test_sequence:        The sequence being tested to see whether it
**                                   contains the given sequence or not.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

#if CONTAINED_IN

bool contained_in(sequence *sequence1, sequence *test_sequence)
{
        symbol *B_symbol1, *al_col2 ;
        sequence *pattern_of_B_symbol1, *pattern_of_al_col2 ;
        symbol *first_symbol ;
        int col1, col2, number_of_children, test_number_of_children,
                sequence_depth = 1, test_sequence_depth = 1,
                ra_index, fe_rows_array = 0, i, offset ;
        struct rows_entry
        {
                symbol *B_symbol1, *al_col2 ;
                sequence *pattern_of_B_symbol1, *pattern_of_al_col2 ;
                bool match_found ;
        } rows_array[MEDIUM_SCRATCH_ARRAY_SIZE] ;
        bool match_found, original_pattern_match_found ;

        if (sequence1 == NIL || test_sequence == NIL)
                abort_run("NIL parameter(s) for contained_in") ;
        if (typeof(sequence1) != typeof(test_sequence)) return(false) ;
        if (typeof(sequence) == SYMBOL || typeof(test_sequence) == SYMBOL)
                abort_run("SYMBOL parameter(s) for contained_in") ;

        /* Compare the lengths of the sequences. */

        number_of_children = sequence1->number_of_children ;
        test_number_of_children = test_sequence->number_of_children ;

        if (number_of_children > test_number_of_children) return(false) ;

        /* Find sequence depth. */
```

```
sequence1->initialise() ;
while (B_symbol1 = (symbol *)sequence1->get_next_child())
        sequence_depth++ ;

/* Find test_sequence_depth and compare with sequence_depth. */

test_sequence->initialise() ;
while (B_symbol1 = (symbol *)test_sequence->get_next_child())
        test_sequence_depth++ ;

if (sequence_depth > test_sequence_depth) return(false) ;

/* Initialise rows_array[]. */

for (ra_index = 0; ra_index < MEDIUM_SCRATCH_ARRAY_SIZE; ra_index++)
{
        rows_array[ra_index].B_symbol1 = NIL ;
        rows_array[ra_index].pattern_of_B_symbol1 = NIL ;
        rows_array[ra_index].al_col2 = NIL ;
        rows_array[ra_index].pattern_of_al_col2 = NIL ;
        rows_array[ra_index].match_found = false ;
}

/* Now fill in values for rows_array[]. The original pattern
of each Old row in sequence1 is compared with the original_pattern
each Old row in test sequence1. If there are no matches,
this means that sequence1 is not contained in
test_sequence. Where there are one or more matches,
these are entered in rows_array[]. */

sequence1->initialise() ;
B_symbol1 = (symbol *)sequence1->get_next_child() ;
while (B_symbol1 = (symbol *)sequence1->get_next_child())
{
        original_pattern_match_found = false ;
        test_sequence->initialise() ;
        B_symbol1 = (symbol *)test_sequence->get_next_child() ;
        while (B_symbol1 = (symbol *)
                test_sequence->get_next_child())
        {
                if (B_symbol1->original_pattern != al_col2->el_obj)
                        continue ;

                original_pattern_match_found = true ;

                /* We have a positive match of the original_patterns.
                Now record it in rows_array[]. */

                rows_array[fe_rows_array].B_symbol1 = B_symbol1 ;
                rows_array[fe_rows_array].pattern_of_B_symbol1 =
                        B_symbol1->el_obj ;
                rows_array[fe_rows_array].al_col2 = al_col2 ;
                rows_array[fe_rows_array].pattern_of_al_col2 =
                        al_col2->el_obj ;
                fe_rows_array = plus_one(fe_rows_array,
                        MEDIUM_SCRATCH_ARRAY_SIZE,
                        "rows_array[] too small in contained_in") ;
        }
        if (original_pattern_match_found == false) return(false) ;
}

/* Now check to see whether the pattern of hits for each Old row
of sequence1 matches a pattern of hits (for the same original_pattern)
in an Old row of test_sequence. The order of matching rows need not
be the same in the two sequences. It is not safe to assume that the
two sequences have the same length: if test_sequence is longer
than sequence1, it is not safe to assume that the pattern of hits
in any one row in sequence1 starts at the same symbol as
test_sequence. Likewise for the end of a pattern of hits.
If the symbols do not coincide, then, for each row, symbols
in test_sequence which are outside the pattern of hits must all
be NIL. */

/* Find an Old row in sequence1 where the first symbol in the
row is not NIL. */

sequence1->initialise() ;
B_symbol1 = (symbol *)sequence1->get_next_child() ;
while (B_symbol1 = (symbol *)sequence1->get_next_child())
{
        if (B_symbol1->symbol_sequence[0] != NIL)
        {
                pattern_of_B_symbol1 = B_symbol1->el_obj ;
```

```
                first_symbol = B_symbol1->symbol_sequence[0] ;
                break ;
        }
}

/* Now find row(s) in test_sequence with the same original_pattern and
find the position in each of these rows of the first symbol of
the original_pattern. From the one which starts furthest to the left,
we can find the offset of the start of sequence1
relative to the start of test_sequence. */

pattern_of_al_col2 = NIL ;
offset = 1000000 ;
test_sequence->initialise() ;
B_symbol1 = (symbol *)test_sequence->get_next_child() ;
while (B_symbol1 = (symbol *)test_sequence->get_next_child())
{
        if (al_col2->el_obj == pattern_of_B_symbol1)
        {
                for (col2 = 0; col2 < test_number_of_children; col2++)
                {
                        if (al_col2->symbol_sequence[col2] !=
                                first_symbol) continue ;
                        if (col2 < offset) offset = col2 ;
                        break ;
                }
                if (col2 >= test_number_of_children)
                        abort_run("Error in contained_in") ;
        }
}

if (offset >= test_number_of_children)
        abort_run("Error in contained_in") ;

ra_index = 0 ;
sequence1->initialise() ;
B_symbol1 = (symbol *)sequence1->get_next_child() ;
while (B_symbol1 = (symbol *)sequence1->get_next_child())
{
        match_found = false ;
        while (rows_array[ra_index].B_symbol1 == B_symbol1)
        {
                al_col2 = rows_array[ra_index].al_col2 ;

                /* Check that this al_col2 has
                not already been 'used'. */

                for (i = 0; i < ra_index; i++)
                        if (rows_array[i].al_col2 == al_col2
                                && rows_array[i].match_found)
                                        goto L1 ;

                /* Step over the offset in al_col2, checking that
                it contains nothing other than NIL values. */

                for (col2 = 0; col2 < offset; col2++)
                        if (al_col2->symbol_sequence[col2] != NIL)
                                goto L1 ;

                /* Now compare the pattern of hits in B_symbol1 and
                al_col2. */

                for (col1 = 0; col1 < number_of_children; col1++)
                {
                        if (col2 >= test_number_of_children) goto L1 ;
                        if (B_symbol1->symbol_sequence[col1] !=
                                al_col2->symbol_sequence[col2])
                                        goto L1 ;
                        col2++ ;
                }

                /* Check that any trailing symbols in al_col2 are
                all NIL. */

                for (col2 = col2; col2 < test_number_of_children; col2++)
                        if (al_col2->symbol_sequence[col2] != NIL)
                                goto L1 ;

                match_found = true ;
                rows_array[ra_index].match_found = true ;
                while (rows_array[ra_index].B_symbol1 == B_symbol1)
                        if (++ra_index >= fe_rows_array) break ;
```

```
                        break ;

                        L1: ;
                        if (++ra_index >= fe_rows_array) break ;
                }
                if (!match_found) return(false) ;
        }
        return(true) ;
} /* contained_in */

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        For each of the sequences in best_sequences[], this function
**        compiles a list of the other best sequences, if any, which
**        contain the given sequence.
**
** CALLING SEQUENCE:
**
**        void find_containing_sequences()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        best_sequences[]
**
** IMPLICIT OUTPUTS:
**
**        For each best_sequence[], a list of containing sequences which
**        may be NIL.
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void find_containing_sequences()
{
        int i, j ;

        for (i = 0; i < fe_best_sequences; i++)
        {
                for (j = 0; j < fe_best_sequences; j++)
                {
                        if (i == j) continue ;
                        if (contained_in(best_sequences[i].al_el,
                                best_sequences[j].al_el))
                        {
                                fprintf(output_file, "Alignment ") ;
                                best_sequences[i].al_el->print_ID() ;
                                fprintf(output_file, " is contained in ") ;
                                best_sequences[j].al_el->print_ID() ;
                                fprintf(output_file, "\n\n") ;
                        }
                }
        }
} /* find_containing_sequences */

#endif

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Called from grammar::tidy_up_code_symbols(). This method looks
**        for CLASS_SYMBOLs and renumbers them from 1. Likewise for
**        UNIQUE_ID_SYMBOLs.
**
** CALLING SEQUENCE:
**
**        void grammar::renumber_code_symbols(int symbol_type)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
```

```
**        symbol_type:        The type of symbol to be renumbered (it should
**                                    be CONTEXT_SYMBOL or UNIQUE_ID_SYMBOL.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void grammar::renumber_code_symbols(int symbol_type)
{
        if (symbol_type != CONTEXT_SYMBOL
                && symbol_type != UNIQUE_ID_SYMBOL)
                        abort_run("Invalid parameter for \
                                grammar::renumber_code_symbols().") ;

        int index, index_limit, new_name_number ;
        symbol *symbol1 ;

        // Scan over set_of_symbols putting the numeric part of the
        // names of symbols with type CONTEXT_SYMBOL into the first column of
        // names_array[]. Only numeric parts of names of the symbols
        // are listed.

        index = 0 ;
        char *symbol_name ;
        list_for(symbol1, symbol, set_of_symbols)
        {
                if (symbol1->get_type() != symbol_type) continue ;
                symbol_name = symbol1->get_name() ;

                if (index >= LARGE_SCRATCH_ARRAY_SIZE)
                        abort_run("Overflow of names_array[] in \
                                grammar::tidy_up_code_symbols()") ;

                if (symbol_type == CONTEXT_SYMBOL)
                        names_array[index++].old_name_number_chars =
                                symbol_name ;
                else names_array[index++].old_name_number_chars =
                        symbol_name + 1 ;
        }

        index_limit = index ;

        // Scan over names_array[] putting in numbers for new names.

        new_name_number = 0 ;
        for (index = 0; index < index_limit; index++)
                names_array[index].new_name_number = ++new_name_number ;

        // Now scan over symbols in list_of_patterns changing names
        // of symbol_type symbols.

        sequence *pattern1 ;
        char new_name[SMALL_SCRATCH_ARRAY_SIZE],
                *old_name_number_chars ;
        int new_name_length ;
        list_for(pattern1, sequence, list_of_patterns)
        {
                list_for(symbol1, symbol, pattern1)
                {
                        if (symbol1->get_type() != symbol_type) continue ;
                        symbol_name = symbol1->get_name() ;

                        // Find the entry in names_array[] where old_name_number_chars
                        // matches the corresponding part of symbol_name.

                        if (symbol_type == CONTEXT_SYMBOL)
                        {
                                old_name_number_chars = symbol_name ;

                                for (index = 0; index < index_limit; index++)
                                {
                                        if (strcmp(names_array[index].old_name_number_chars,
                                                old_name_number_chars) == 0) break ;
                                }
```

```
                        if (index >= index_limit)
                                abort_run("Symbol name not found in \
                                        grammar::tidy_up_code_symbols()") ;

                        new_name_length = sprintf(new_name, "%d",
                                                        names_array[index].new_name_number) ;
                }
                else
                {
                        old_name_number_chars = symbol_name + 1 ;

                        for (index = 0; index < index_limit; index++)
                        {
                                if (strcmp(names_array[index].old_name_number_chars,
                                        old_name_number_chars) == 0) break ;
                        }

                        if (index >= index_limit)
                                abort_run("Symbol name not found in \
                                        grammar::tidy_up_code_symbols()") ;

                        new_name_length =
                                sprintf(new_name, "%c%d",
                                        '#',
                                        names_array[index].new_name_number) ;
                }

                if (new_name_length + 1 >= SMALL_SCRATCH_ARRAY_SIZE
                        || new_name_length < 0)
                                abort_run("Error in name construction in \
                                        grammar::tidy_up_code_symbols()") ;

                symbol_name = NIL ; // To avoid dangling pointer to
                        // deleted old name.

                symbol1->set_name(new_name) ;
            }
        }
} // grammar::renumber_code_symbols

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Adds a pattern to a grammar checking that there are no identical
**        patterns or patterns that contain the same symbols. The method
**        also checks that the pattern to be added does not have exactly
**        the same CONTENTS symbols as any pattern already in the grammar.
**        If it does, its IDENTIFICATION symbols (other than its
**        UNIQUE_ID_SYMBOL) are copied into the pattern already in the
**        grammar, avoiding duplicates.
**
** CALLING SEQUENCE:
**
**        bool grammar::add_pattern(sequence *pattern1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if pattern1 is added, false otherwise.
**
**        pattern1:                The pattern to be added.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

bool grammar::add_pattern(sequence *pattern1)
{
        sequence *pattern2 ;

        // Check that pattern1 is not identical with any
        // pattern already in the grammar.
```

```
        list_for(pattern2, sequence, list_of_patterns)
                if (pattern2 == pattern1) return(false) ;

        // Now check to see whether pattern1 has all and only the
        // same CONTENTS symbols as any pattern already in the grammar.
        // If it does, it is not added to the grammar but its CONTENTS
        // symbols, other than its UNIQUE_ID_SYMBOL, are copied into
        // the pattern already in the grammar, avoiding duplicates.

        symbol *symbol1, *symbol2 ;
        int type1, type2 ;
        bool symbol_match_found ;
        int id_number_array[MEDIUM_SCRATCH_ARRAY_SIZE],
                fe_id_number_array, i ;
        list_for(pattern2, sequence, list_of_patterns)
        {
                if (pattern1->contents_symbols_match(pattern2) == false)
                        continue ;

                // A CONTENTS symbols match has been found.
                // Copy over non-duplicate IDENTIFICATION symbols.

                fe_id_number_array = 0 ;
                list_for(symbol1, symbol, pattern1)
                {
                        type1 = symbol1->get_type() ;
                        if (type1 == LEFT_BRACKET) continue ;
                        if (type1 == UNIQUE_ID_SYMBOL
                                || symbol1->get_status() == IDENTIFICATION)
                                        break ;

                        symbol_match_found = false ;
                        list_for(symbol2, symbol, pattern2)
                        {
                                type2 = symbol1->get_type() ;
                                if (type2 == LEFT_BRACKET) continue ;
                                if (type2 == UNIQUE_ID_SYMBOL
                                        || symbol2->get_status() == IDENTIFICATION)
                                                break ;
                                if (symbol1->name_matches(symbol2))
                                {
                                        symbol_match_found = true ;
                                        break ;
                                }
                        }

                        if (symbol_match_found == false)
                        {
                                id_number_array[fe_id_number_array] =
                                        atoi(symbol1->get_name()) ;
                                plus_one(&fe_id_number_array,
                                        MEDIUM_SCRATCH_ARRAY_SIZE,
                                        "Array overflow in grammar::add_pattern().") ;
                        }
                }

                // Now add symbols to pattern2 with IDENTIFICATION symbol
                // names generated from id_number_array[].

                for (i = 0; i < fe_id_number_array; i++)
                        pattern2->add_context_symbol(id_number_array[i]) ;

                return(false) ;
        }

        list_of_patterns->add_child(pattern1) ;

        return(true) ;

} // grammar::add_pattern

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        This function renames the IDENTIFICATION symbols in a grammar so
**        that numbering starts from 1. UNIQUE_ID_SYMBOLs have a different
**        sequence from ID symbols that are CONTEXT_SYMBOLS.
**        The bit_costs of symbols are not changed.
**
** CALLING SEQUENCE:
```

```
**
**        void grammar::tidy_up_code_symbols(int original_ID)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        original_ID                Where a grammar has been copied, this is the ID
**                                        of the original grammar to be used in writing out.
**                                        If it is NULL_VALUE, the ID of the current grammar
**                                        is used.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void grammar::tidy_up_code_symbols(int original_ID)
{
        int alphabet_size ;

        names_array = new struct names_entry[LARGE_SCRATCH_ARRAY_SIZE] ;

        set_of_symbols = list_of_patterns->
                                        compile_alphabet(&alphabet_size) ; // This compiles
                // a list of symbols in list_of_patterns and sorts them
                // in name order.

        renumber_code_symbols(CONTEXT_SYMBOL) ;
        renumber_code_symbols(UNIQUE_ID_SYMBOL) ;

        // Finally, print out list_of_patterns with new names of symbols.

        fprintf(output_file, "TIDIED UP GR") ;
        if (original_ID == NULL_VALUE)
                fprintf(output_file, "%d", this->get_ID()) ;
        else fprintf(output_file, "%d", original_ID) ;
        fprintf(output_file, "\n\n") ;

        write_grammar(false, false) ;

        delete set_of_symbols ;
        delete[] names_array ;

} // grammar::tidy_up_code_symbols

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        This function, called from compile_alternative_grammars(),
**        adds Old patterns in a 'full' alignment to the patterns in
**        the grammar, checking that there are no duplicates in the grammar,
**        either the same pattern entered twice or two different patterns
**        that contain exactly the same symbols.
**
**        If include_code is true, the code pattern derived from the
**        alignment is included in the grammar.
**
** CALLING SEQUENCE:
**
**        void grammar::compile_grammar(sequence *full_alignment)
**
** FORMAL ARGUMENTS:
**
**        Return value:                void
**
**        full_alignment:                The 'full' alignment whose patterns are to
**                                                be added to the grammar (without duplicates).
**
** IMPLICIT INPUTS:
**
**        NONE
```

```
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void grammar::compile_grammar(sequence *full_alignment)
{
        int row, sequence_depth = full_alignment->get_sequence_depth() ;
        sequence *row_pattern ; //, *copy_pattern ;

        for (row = 1; row < sequence_depth; row++)
        {
                row_pattern = full_alignment->get_row_pattern(row) ;
                // copy_pattern = new sequence(*row_pattern) ;
                add_pattern(row_pattern) ;
        }

        // Diagnostic

        #if DIAGNOSTIC4
        fprintf(output_file, "DIAGNOSTIC:\n\n") ;
        write_grammar(true) ;
        fprintf(output_file, "END DIAGNOSTIC\n\n") ;
        fflush(output_file) ;
        #endif

} // grammar::compile_grammar

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Writes a recursive trace of how a compilation of grammars
**        was formed.
**
** CALLING SEQUENCE:
**
**        void grammar::write_trace(int indentation)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        indentation:        The distance from the left margin to start.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void grammar::write_trace(int indentation)
{
        int i ;
        grammar *child ;

        for (i = 0; i < indentation; i++)
                fprintf(output_file, "  ") ;

        if (indentation == 0)
                fprintf(output_file, "Root\n") ;
        else fprintf(output_file, "%s%d%s%d%s%3.2f%s%3.2f%s%3.2f%s",
                "GR",
                ID,
                " (",
                derived_from_grammar,
                "), G = ",
                G,
                ", E = ",
```

```
                E,
                ", score = ",
                score,
                "\n") ;

        if (basis_for != NIL)
        {
                list_for(child, grammar, basis_for)
                        child->write_trace(indentation + 1) ;
        }

} // grammar::write_trace

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Writes out the patterns in the grammar with scores etc.
**
** CALLING SEQUENCE:
**
**        void grammar::write_grammar(bool write_derived_from,
**                bool write_details)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        write_derived_from:               If true, 'derived_from' field is written,
**                                                  otherwise it is not.
**        write_details:                       The details of the grammar patterns are
**                                                  are written if true, not otherwise.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        Output to output_file
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void grammar::write_grammar(bool write_derived_from,
        bool write_details)
{
        sequence *pattern1 ;

        fprintf(output_file, "%s%d%s",
                "GRAMMAR GR",
                ID,
                " (") ;
        print_pattern_cycle(false, current_new_pattern) ;

        if (write_details)
        fprintf(output_file, ") WITH DETAILS\n") ;
        else fprintf(output_file, ")\n") ;

        if (write_derived_from)
        {
                fprintf(output_file, "derived from grammar ") ;

                if (derived_from_grammar == NULL_VALUE)
                        fprintf(output_file, "NULL") ;
                else fprintf(output_file, "%s%d",
                        "GR",
                        derived_from_grammar) ;

                fprintf(output_file, " and alignment:\n") ;
                if (derived_from_alignment == NIL)
                        fprintf(output_file, "NIL\n") ;
                else derived_from_alignment->write_pattern(true, false) ;
                fprintf(output_file, "G = ") ;
        }
        else fprintf(output_file, "G = ") ;

        fprintf(output_file, "%3.2f%s%3.2f%s%3.2f%s",
                G,
```

```
                ", E = ",
                E,
                ", score = ",
                score,
                ".\n\nGrammar patterns:\n\n") ;

        list_for(pattern1, sequence, list_of_patterns)
        {
                if (write_details)
                        pattern1->write_with_details(true) ;
                else pattern1->write_pattern(true, true) ;
        }
} // grammar::write_grammar

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Makes a copy of a grammar. Notice that the list_of_patterns is
**        copied, but the individual patterns within the list are *not*
**        copied.
**
** CALLING SEQUENCE:
**
**        grammar::grammar(grammar &gr) : base_object(gr)
**
** FORMAL ARGUMENTS:
**
**        Return value:        A new grammar.
**
**        gr:                                The grammar to be copied.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

grammar::grammar(grammar &gr) : base_object(gr)
{
        derived_from_grammar = gr.get_derived_from_grammar() ;
        derived_from_alignment = gr.get_derived_from_alignment() ;
        G = gr.get_G();
        E = gr.get_E();
        score = gr.get_score();
        ID = grammar_ID_number++ ;
        basis_for = NIL ;
        list_of_patterns = new group ;

        sequence *pattern1 ;
        gr.initialise() ;
        while (pattern1 = gr.get_next_pattern())
                list_of_patterns->add_child(pattern1) ;

} // grammar::grammar(grammar &gr)

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Copies the details of one grammar into another and makes literal
**        copies of the patterns in the one grammar for inclusion in the
**        other.
**
** CALLING SEQUENCE:
**
**        void grammar::copy_details_and_patterns(grammar *gr)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void.
**
**        gr:                                The grammar whose details and patterns are to
```

```
**                                      be copied.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void grammar::copy_details_and_patterns(grammar *gr)
{
        derived_from_grammar = gr->get_derived_from_grammar() ;
        derived_from_alignment = gr->get_derived_from_alignment() ;
        G = gr->get_G();
        E = gr->get_E();
        score = gr->get_score();
        basis_for = NIL ;

        sequence *pattern1, *pattern2 ;
        gr->initialise() ;
        while (pattern1 = gr->get_next_pattern())
        {
                pattern2 = new sequence(*pattern1) ;
                list_of_patterns->add_child(pattern2) ;
        }

} // grammar::copy_details_and_patterns

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Computes G a grammar.
**
** CALLING SEQUENCE:
**
**        void grammar::compute_G()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        Value for G for the grammar.
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void grammar::compute_G()
{
        sequence *pattern1 ;
        double cumulative_G = 0 ;

        list_for(pattern1, sequence, list_of_patterns)
                cumulative_G += pattern1->get_total_cost() ;

        G = cumulative_G ;
} // grammar::compute_G

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Creates an alignment from a list of basic alignments.
**
** CALLING SEQUENCE:
```

```
**
**        sequence *combination::make_composite_alignment(sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        cnp:                        The 'current' New pattern
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        An encoding of the compposite alignment is added to
**        current_code_patterns.
**
** SIDE EFFECTS:
**
**        NONE
**
*/

sequence *combination::make_composite_alignment(sequence *cnp)
{
        sequence *sub_alignment,
                *composite_alignment = new sequence(COMPOSITE_ALIGNMENT) ;
        int overall_depth = 0, ba_depth, new_row1, new_row2, old_row ;
        symbol *sub_alignment_symbol, *newly_made_column,
                *symbol1 ;
        alignment_element *al_el1, *al_el2 ;
        char *symbol_name ;

        composite_alignment->set_composite_alignment(true) ;

        // Calculate depth of new composite alignment as
        // D = SUM (d_i - 1) + 1, summing over the set of basic alignments
        // and with d representing the depth of each individual alignment.
        // The reasoning is that row 0 is the same for all alignments so
        // we don't need to include it in the depth of each alignment, but
        // we need to add in row 0.

        list_for(sub_alignment, sequence, sub_alignment_list)
        {
                ba_depth = sub_alignment->get_sequence_depth() - 1 ;
                overall_depth += ba_depth ;
        }

        overall_depth += 1 ; // Add 1 for the New pattern in row 0.

        composite_alignment->set_sequence_depth(overall_depth) ;

        // Now copy the basic alignments into a new composite alignment
        // with each symbol having a depth equal to overall_depth and
        // with each row of each basic alignment given a unique row
        // in the composite alignment.

        symbol *first_symbol, *last_symbol ;

        new_row1 = 1 ; // Row 0 is the same in the original symbol and
                // the new symbol. So the first row in the new symbol
                // for purposes of copying is row 1.

        list_for(sub_alignment, sequence, sub_alignment_list)
        {
                ba_depth = sub_alignment->get_sequence_depth() ;

                first_symbol = (symbol *)sub_alignment->get_first_child() ;
                last_symbol = (symbol *)sub_alignment->get_last_child() ;

                list_for(sub_alignment_symbol, symbol, sub_alignment)
                {
                        new_row2 = new_row1 ;

                        symbol_name = sub_alignment_symbol->get_name() ;

                        newly_made_column = new symbol(symbol_name, overall_depth,
                                NULL_VALUE) ;
                        newly_made_column->set_bit_cost(sub_alignment_symbol->
                                                                        get_bit_cost()) ;
                        newly_made_column->set_is_a_hit(sub_alignment_symbol->
                                                                        is_a_hit()) ;
```

```
                newly_made_column->set_status(sub_alignment_symbol->
                                                                        get_status()) ;
                newly_made_column->set_type(sub_alignment_symbol->
                                                                        get_type()) ;

                // Now set values for row 0.

                al_el1 = sub_alignment_symbol->get_al_el(0) ;
                al_el2 = newly_made_column->get_al_el(0) ;
                al_el2->set_el_obj(al_el1->get_el_obj()) ;
                al_el2->set_original_pattern(al_el1->
                        get_original_pattern()) ;

                // Set values for other rows.

                for (old_row = 1; old_row < ba_depth; old_row++)
                {
                        if (new_row2 >= overall_depth)
                                abort_run("Anomaly in value of new_row \
                                        in make_composite_alignment()") ;
                        al_el1 = sub_alignment_symbol->get_al_el(old_row) ;
                        al_el2 = newly_made_column->get_al_el(new_row2) ;
                        al_el2->set_el_obj(al_el1->get_el_obj()) ;
                        al_el2->set_original_pattern(al_el1->
                                get_original_pattern()) ;
                        new_row2++ ;
                }

                // Set values for same_column_above,
                // same_column_below and symbol_is_a_hit.

                newly_made_column->set_symbol_matches() ;

                composite_alignment->add_child(newly_made_column) ;
        } // End of one basic alignment
        new_row1 = new_row2 ;
} // End of sub_alignment_list

composite_alignment->mark_parent_and_int_positions_non_recursive() ;

// Fill in values for original_pattern in every row of every
// column. First, find the first column that has a non-NIL
// value for original_pattern in this row.

sequence *original_pattern ;
for (new_row1 = 0; new_row1 < overall_depth; new_row1++)
{
        // Find the original_pattern for this row.

        for (symbol1 = (symbol *)composite_alignment->get_first_child();
                symbol1 != NIL; symbol1 = (symbol *)
                        composite_alignment->get_next_child())
        {
                original_pattern = symbol1->get_row_pattern(new_row1) ;
                if (original_pattern != NIL) break ;
        }

        // Now fill it in right across the alignment.

        for (symbol1 = (symbol *)composite_alignment->get_first_child();
                symbol1 != NIL; symbol1 = (symbol *)
                        composite_alignment->get_next_child())
        {
                al_el1 = symbol1->get_al_el(new_row1) ;
                al_el1->set_original_pattern(original_pattern) ;
        }
}

// Calculate values for NSC, EC, CD, CR and abs_P for
// the alignment.

composite_alignment->make_code(false) ;

composite_alignment->set_symbol_matches() ;

// Print new composite alignment as alignment and as
// a flat pattern, with and without details.

composite_alignment->write_out_fully("COMPOSITE ALIGNMENT",
        NIL, write_section_chars_length, NULL_VALUE, true,
        cnp) ;

if (show_al_structure)
```

```
                composite_alignment->show_al_structure(NIL) ;

        return(composite_alignment) ;

} // combination::make_composite_alignment

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Tests whether the list of basic alignments in 'this' is a subset
**        of the list of basic alignments in test_comb.
**
** CALLING SEQUENCE:
**
**                bool combination::is_subset_of(combination *test_comb)
**
** FORMAL ARGUMENTS:
**
**                Return value:        true if the basic alignments in 'this' are a subset
**                                             of the basic alignments in test_comb, false otherwise.
**
**                test_comb:                The combination against which 'this' is to be tested.
**
** IMPLICIT INPUTS:
**
**                NONE
**
** IMPLICIT OUTPUTS:
**
**                NONE
**
** SIDE EFFECTS:
**
**                NONE
**
*/

bool combination::is_subset_of(combination *test_comb)
{
        group *test_sub_alignment_list =
                test_comb->get_sub_alignment_list() ;
        sequence *ba_this, *ba_test ;

        // 'this' combination must not be the same as test_comb.

        if (this == test_comb)
                abort_run("Invalid comparison in combination::is_subset_of().") ;

        list_for(ba_this, sequence, sub_alignment_list)
        {
                list_for(ba_test, sequence, test_sub_alignment_list)
                {
                        if (ba_this == ba_test) break ;
                }
                if (ba_test == NIL) return(false) ; // An instance of ba_this
                                        // has been found which is not present on
                                        // test_sub_alignment_list. Therefore,
                                        // sub_alignment_list is *not* a subset of
                                        // test_sub_alignment_list.
        }
        return(true) ; // Every instance of ba_this is found on
                               // test_sub_alignment_list. Therefore,
                               // sub_alignment_list *is* a subset of
                               // test_sub_alignment_list.
} // combination::is_subset_of

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Prints the score and the list of basic alignments in a combination.
**
** CALLING SEQUENCE:
**
**        void combination::print_combination()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
```

```
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        Output to file
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void combination::print_combination()
{
        sequence *sub_alignment ;

        fprintf(output_file, "%s%d%s%1.2f%s",
                "Combination C_ID",
                C_ID,
                " (score ",
                combination_score,
                ") sub-alignments:\n\n") ;

        bool first_in_list = true ;
        list_for(sub_alignment, sequence, sub_alignment_list)
        {
                if (first_in_list)
                {
                        fprintf(output_file, "%s%d",
                                "ID",
                                sub_alignment->get_ID()) ;
                        first_in_list = false ;
                }
                else
                {
                        fprintf(output_file, "%s%d",
                                ", ID",
                                sub_alignment->get_ID()) ;
                }
        }

        fprintf(output_file, "\n\n") ;
} // combination::print_sub_alignments

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Copy constructor for class combination.
**
** CALLING SEQUENCE:
**
**        combination::combination(combination &comb1) : base_object(comb1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        Creates a copy of the input combination
**
**        comb1:                        The combination to be copied.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

combination::combination(combination &comb1) : base_object(comb1)
{
        sequence *sub_alignment ;
        int i ;

        C_ID = combination_ID_number++ ;
```

```
        combination_score = comb1.combination_score ;

        len_fit_seq = comb1.len_fit_seq ;

        if (len_fit_seq >= MEDIUM_SCRATCH_ARRAY_SIZE)
                abort_run("len_fit_seq too long in combination") ;
        for (i = 0; i < len_fit_seq; i++)
                fitting_sequence[i] = comb1.fitting_sequence[i] ;

        group *temp_list = comb1.sub_alignment_list ;
        sub_alignment_list = new group ;
        for (sub_alignment = (sequence *)temp_list->get_first_child();
                sub_alignment != NIL; sub_alignment = (sequence *)
                        temp_list->get_next_child())
        {
                sub_alignment_list->add_child(sub_alignment) ;
        }
} // combination::combination(combination &comb1)

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        This method in class combination tests whether or not a given basic
**        alignment 'fits' into the combination. A basic combination fits if
**        all the symbols from New that it covers have not already been marked
**        as true in the fitting _sequence of 'this'.
**
** CALLING SEQUENCE:
**
**        bool combination::can_accept(int pos_new_first,
**                int pos_new_last)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if sub_alignment fits into
**                                     fitting _sequence, otherwise false.
**
**        pos_new_first:        The integer position of the first hit
**                                     symbol from current_new_pattern in bal1.
**        pos_new_last:        The integer position of the last hit
**                                     symbol from current_new_pattern in bal1.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

bool combination::can_accept(int pos_new_first,
        int pos_new_last)
{
        for (int i = pos_new_first; i <= pos_new_last; i++)
                if (fitting_sequence[i] == true) return(false) ;

        return(true) ;
} // combination::can_accept

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Adds a 'basic' alignment to a combination.
**
** CALLING SEQUENCE:
**
**        void combination::add_sub_alignment(sequence *bal1,
**                int bal1_pos_new_first, int bal1_pos_new_last,
**                sequence *cnp)
**
** FORMAL ARGUMENTS:
**
```

```
**        Return value:               void
**
**        bal1:                               The sub_alignment to be added to 'this'.
**        bal1_pos_new_first:        The integer position of the first hit
**                                             symbol from current_new_pattern in bal1.
**        bal1_pos_new_last:        The integer position of the last hit
**                                             symbol from current_new_pattern in bal1.
**        cnp:                               The New pattern being processed.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        Addition to the structure of 'this'. The 'coverage' of the
**        current pattern from New by the newly-added basic alignment
**        is marked in fitting_sequence.
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void combination::add_sub_alignment(sequence *bal1,
        int bal1_pos_new_first, int bal1_pos_new_last,
        sequence *cnp)
{
        sequence *bal2 ;
        symbol *symbol1, *symbol_new ;
        int bal2_pos_new_first ;

        // Add sub_alignment to sub_alignment_list in the
        // position in the list that corresponds to the order
        // of the alignments relative to the current pattern from New.
        // This is done to facilitate the building of composite alignments
        // in make_composite_alignment.

        list_for(bal2, sequence, sub_alignment_list)
        {
                // Find the first symbol in bal2 that contains a symbol
                // from current_new_pattern and find the position of
                // that symbol in current_new_pattern.

                list_for(symbol1, symbol, bal2)
                {
                        symbol_new = symbol1->get_row_symbol(0) ;
                        if (symbol_new != NIL) break ;
                }

                if (symbol1 == NIL) abort_run("Error in add_sub_alignment()") ;

                bal2_pos_new_first = symbol1->get_row_orig_int_pos(0) ;

                // Compare this with the corresponding integer position
                // of the first hit symbol from current_new_pattern
                // in sub_alignment.

                if (bal1_pos_new_first < bal2_pos_new_first) break ;
        }

        if (bal2 == NIL) sub_alignment_list->add_child(bal1) ;
        else sub_alignment_list->precede(bal1, bal2) ;

        combination_score += bal1->get_compression_difference() ;

        // Now mark the entries in 'fitting_sequence' that are
        // 'covered' by the newly-added basic alignment.

        for (int i = bal1_pos_new_first; i <= bal1_pos_new_last; i++)
                fitting_sequence[i] = true ;

        #if DIAGNOSTIC5

        fprintf(output_file, "%s%d%s",
                "Combination C_ID",
                C_ID,
                " is derived from alignments: ") ;

        sub_alignment_list->write_IDs(NULL_VALUE, cnp) ;

        #endif
```

```
} // combination::add_sub_alignment

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks to see whether a given 'reference' class symbol (with
**        status CONTENTS) occurs once or more than once as an IDENTIFICATION
**        symbol in the same grammar. If a single pattern is found that contains
**        the given class symbol as an IDENTIFICATION symbol, then the function
**        returns that pattern. If more than one pattern is found, the function
**        returns NIL.
**
** CALLING SEQUENCE:
**
**        sequence *single_reference(symbol *ref1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

sequence *single_reference(symbol *ref1, grammar *gr1)
{
        sequence *pattern1, *single_referent ;
        symbol *symbol1 ;
        int ref_counter = 0, id_counter = 0 ;
        list_element *el_pos1, *el_pos2 ;
        group *list_of_patterns = gr1->list_of_patterns ;

        // Check that the ref1 symbol type occurs only once as
        // a CONTENTS symbol and only once as an IDENTIFICATION symbol.

        el_pos1 = NIL ;
        list_for_el_pos(pattern1, sequence, list_of_patterns, el_pos1)
        {
                if (pattern1->get_origin() == CODE_PATTERN) continue ;
                el_pos2 = NIL ;
                list_for_el_pos(symbol1, symbol, pattern1, el_pos2)
                {
                        if (symbol1->name_matches(ref1))
                        {
                                if (symbol1->get_status() == CONTENTS)
                                        ref_counter++ ;
                                else
                                {
                                        id_counter++ ;
                                        single_referent = pattern1 ;
                                }
                        }
                }
        }

        if (ref_counter == 1 && id_counter == 1)
                return(single_referent) ;
        else return(NIL) ;
} // single_reference

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Examines a grammar to find patterns that are referenced from one
**        context and only one context and where there is no other pattern
**        that is referenced from that context. Any such pattern is merged
**        with the context pattern in a newly-created pattern that
**        replaces the context pattern and the contained pattern as
**        they appear in the grammar. Corresponding adjustments are made
```

```
**        in the CODE_PATTERNs.
**
**        This kind of merging can mean that other patterns fall into the
**        same category. So the process needs to be repeated to check for
**        that kind of possibility - and so on until no more merging can be
**        done.
**
** CALLING SEQUENCE:
**
**        void grammar::merge_patterns()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void grammar::merge_patterns()
{
        sequence *pattern1, *new_pattern, *single_ref ;
        symbol *symbol1, *symbol2, *symbol3, *symbol4,
                *left_bracket, *right_bracket ;
        list_element *el_pos1, *el_pos2, *el_pos3 ;
        bool done = false, patterns_have_been_merged = false ;
        int fr_single_ref, fr_pattern1, lower_frequency ;

        while (done == false)
        {
                done = true ;
                el_pos1 = NIL ;
                list_for_el_pos(pattern1, sequence, list_of_patterns, el_pos1)
                {
                        if (pattern1->get_origin() == CODE_PATTERN) continue ;
                        el_pos2 = NIL ;
                        list_for_el_pos(symbol1, symbol, pattern1, el_pos2)
                        {
                                if (symbol1->get_status() != CONTENTS) continue ;
                                if (symbol1->get_type() != CONTEXT_SYMBOL) continue ;
                                single_ref = single_reference(symbol1, this) ;
                                if (single_ref == NIL) continue ;

                                // Merge single_ref with pattern1 (in a new copy).

                                new_pattern = new sequence(*pattern1) ;

                                // Find the 'reference' to single_ref, copy in the
                                // CONTENTS symbols from single_ref and
                                // extract the 'reference' symbols.

                                list_for(symbol2, symbol, new_pattern)
                                {
                                        if (symbol2->get_type() == LEFT_BRACKET)
                                                left_bracket = symbol2 ;
                                        if (symbol2->name_matches(symbol1))
                                        {
                                                right_bracket =
                                                        (symbol *)new_pattern->get_next_child() ;
                                                if (right_bracket->get_type() != RIGHT_BRACKET)
                                                        abort_run("Anomaly in \
                                                                grammar::merge_patterns().") ;
                                                break ;
                                        }
                                }

                                if (symbol2 == NIL)
                                        abort_run("Anomaly in grammar::merge_patterns().") ;

                                // Copy the CONTENTS symbols of single_ref into new_pattern
                                // immediately before left_bracket.

                                list_for(symbol3, symbol, single_ref)
                                {
```

```
                                if (symbol3->get_status() != CONTENTS) continue ;
                                symbol4 = new symbol(*symbol3) ;
                                new_pattern->precede(symbol4, left_bracket) ;
                        }

                        // Delete the 'reference' symbols in new_pattern.

                        new_pattern->delete_child(left_bracket) ;
                        new_pattern->delete_child(symbol2) ;
                        new_pattern->delete_child(right_bracket) ;

                        new_pattern->
                                mark_parent_and_int_positions_non_recursive() ;

                        // Set the frequency of new_pattern to be the lower
                        // of single_ref and pattern1.

                        fr_single_ref = single_ref->get_frequency() ;
                        fr_pattern1 = pattern1->get_frequency() ;
                        lower_frequency = (fr_pattern1 < fr_single_ref)
                                                        ? fr_pattern1 : fr_single_ref ;
                        new_pattern->set_frequency(lower_frequency) ;

                        list_of_patterns->delete_child(pattern1) ;
                        list_of_patterns->delete_child(single_ref) ;
                        list_of_patterns->add_child(new_pattern) ;

                        patterns_have_been_merged = true ;
                        done = false ;
                        goto L1 ;
                }
        }
        L1: ;
}

if (patterns_have_been_merged)
{
        // Examine the CODE_PATTERNs and remove all
        // CLASS_SYMBOLs that do not appear anywhere in
        // the main grammar.

        // First, compile a list of the CLASS_SYMBOLs in
        // the CODE_PATTERNs.

        group *class_symbols_in_code_patterns = new group ;
        bool class_symbol_found ;

        el_pos3 = NIL ;
        list_for_el_pos(pattern1, sequence, list_of_patterns, el_pos3)
        {
                if (pattern1->get_origin() != CODE_PATTERN) continue ;
                list_for(symbol1, symbol, pattern1)
                {
                        if (symbol1->get_type() != CONTEXT_SYMBOL) continue ;
                        if (class_symbols_in_code_patterns->
                                contains_copy_of(symbol1) == false)
                        {
                                symbol2 = new symbol(*symbol1) ;
                                class_symbols_in_code_patterns->add_child(symbol2) ;
                        }
                }
        }

        // Now look at the symbols in class_symbols_in_code_patterns
        // to find any that do not have copies in the patterns of
        // the main grammar. Any such symbols are removed from the
        // code patterns.

        list_for(symbol1, symbol, class_symbols_in_code_patterns)
        {
                el_pos1 = NIL ;
                class_symbol_found = false ;
                list_for_el_pos(pattern1, sequence, list_of_patterns, el_pos1)
                {
                        if (pattern1->get_origin() == CODE_PATTERN) continue ;

                        // pattern1 is in the main grammar.

                        el_pos2 = NIL ;
                        list_for_el_pos(symbol2, symbol, pattern1, el_pos2)
                        {
                                if (symbol1->name_matches(symbol2))
                                {
```

```
                                        class_symbol_found = true ;
                                        goto L2 ;
                                }
                        }
                }
                L2: if (class_symbol_found) continue ;

                // Delete copies of symbol1 wherever they
                // occur in the CODE_PATTERNs.

                el_pos3 = NIL ;
                list_for_el_pos(pattern1, sequence, list_of_patterns, el_pos3)
                {
                        if (pattern1->get_origin() != CODE_PATTERN) continue ;
                        while (symbol2 = pattern1->contains_copy_of(symbol1))
                                pattern1->delete_child(symbol2) ;
                }
        }

        class_symbols_in_code_patterns->delete_children() ;
        delete class_symbols_in_code_patterns ;

        initialise() ;
        while (pattern1 = get_next_pattern())
                pattern1->compute_costs() ;

        compute_G() ;
        sum_G_and_E() ;

        // Write out the grammar in its new form.

        fprintf(output_file, "%s%d%s",
                "GRAMMAR GR",
                this->get_ID(),
                " AFTER MERGING OF PATTERNS\n\n") ;

        write_grammar(false, false) ;
    }
    else fprintf(output_file, "%s%d%s",
            "No patterns to be merged in grammar GR",
            this->get_ID(),
            "\n\n") ;

} // grammar::merge_patterns

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**      Determines whether or not a given class symbol (from
**      a CODE_PATTERN) is used as an ID symbol in the main grammar.
**
** CALLING SEQUENCE:
**
**      bool grammar::is_ID_symbol(symbol *s1)
**
** FORMAL ARGUMENTS:
**
**      Return value:       true if s1 is an ID symbol within the grammar, false
**                                  otherwise.
**
**      s1:                             The class symbol to be tested.
**
** IMPLICIT INPUTS:
**
**      NONE
**
** IMPLICIT OUTPUTS:
**
**      NONE
**
** SIDE EFFECTS:
**
**      NONE
*/

bool grammar::is_ID_symbol(symbol *s1)
{
    sequence *pattern1 ;
    symbol *symbol1 ;
    list_element *el_pos1, *el_pos2 ;
```

```
        el_pos1 = NIL ;
        list_for_el_pos(pattern1, sequence, list_of_patterns, el_pos1)
        {
                if (pattern1->get_origin() == CODE_PATTERN) continue ;
                el_pos2 = NIL ;
                list_for_el_pos(symbol1, symbol, pattern1, el_pos2)
                {
                        if (symbol1->get_type() == LEFT_BRACKET) continue ;
                        if (symbol1->get_status() == CONTENTS) break ;
                        if (symbol1->name_matches(s1)) return(true) ;
                }
        }

        return(false) ;
} // grammar::is_ID_symbol

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Determines whether or not a given class symbol is referenced within
**        the grammar patterns.
**
** CALLING SEQUENCE:
**
**        bool grammar::is_referenced(symbol *s1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if s1 is referenced within the grammar, false
**                                    otherwise.
**
**        s1:                             The class symbol to be tested.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

bool grammar::is_referenced(symbol *s1)
{
        sequence *pattern1 ;
        symbol *symbol1 ;
        list_element *el_pos1, *el_pos2 ;

        el_pos1 = NIL ;
        list_for_el_pos(pattern1, sequence, list_of_patterns, el_pos1)
        {
                el_pos2 = NIL ;
                list_for_el_pos(symbol1, symbol, pattern1, el_pos2)
                {
                        if (symbol1->get_type() == LEFT_BRACKET) continue ;
                        if (symbol1->get_status() != CONTENTS) continue ;
                        if (symbol1->name_matches(s1)) return(true) ;
                }
        }

        return(false) ;
} // grammar::is_referenced

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        This method checks each pattern in the grammar to ensure that
**        it has enough CONTEXT_SYMBOLs to ensure that it can be
**        identified uniquely in the grammar ***but not more than that***.
**        If too many context symbols are found, the excess are removed
**        provided that they are not used as 'references' in any of the
**        patterns in the grammar. Each pattern retains its
**        UNIQUE_ID_SYMBOL.
**
** CALLING SEQUENCE:
```

```
**
**        void grammar::clean_up(int original_ID)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        original_ID                Where a grammar has been copied, this is the ID
**                                        of the original grammar to be used in writing out.
**                                        If it is NULL_VALUE, the ID of the current grammar
**                                        is used.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void grammar::clean_up(int original_ID)
{
        sequence *pattern1 ;
        symbol *symbol1 ;
        list_element *el_pos1, *el_pos2 ;
        symbol *scratch_array[MEDIUM_SCRATCH_ARRAY_SIZE] ;
        int fe_scratch_array, i, number_of_CLASS_ID_symbols ;

        el_pos1 = NIL ;
        list_for_el_pos(pattern1, sequence, list_of_patterns, el_pos1)
        {
                fe_scratch_array = 0 ;
                el_pos2 = NIL ;
                list_for_el_pos(symbol1, symbol, pattern1, el_pos2)
                {
                        if (symbol1->get_type() != CONTEXT_SYMBOL) continue ;
                        if (symbol1->get_status() != IDENTIFICATION) break ;

                        // symbol1 is an IDENTIFICATION CONTEXT_SYMBOL.

                        if (is_referenced(symbol1)) continue ; // This symbol
                                // is referenced from within the grammar.

                        // Make an entry in scratch_array[].

                        scratch_array[fe_scratch_array] = symbol1 ;
                        plus_one(&fe_scratch_array, MEDIUM_SCRATCH_ARRAY_SIZE,
                                "Array overflow in grammar::clean_up().") ;
                }

                // Now delete all the non-referenced IDENTIFICATION CLASS_SYMBOLs
                // making sure that the pattern contains at least one
                // IDENTIFICATION CONTEXT_SYMBOL.

                for (i = 0; i < fe_scratch_array; i++)
                {
                        number_of_CLASS_ID_symbols =
                                pattern1->find_number_of_CLASS_ID_symbols() ;
                        if (number_of_CLASS_ID_symbols > 1)
                                pattern1->delete_child(scratch_array[i]) ;
                        else break ;
                }
        }

        initialise() ;
        while (pattern1 = get_next_pattern())
                pattern1->compute_costs() ;

        compute_G() ;
        sum_G_and_E() ;

        // Now write out the grammar in its cleaned-up form.

        fprintf(output_file, "CLEANED UP GRAMMAR GR") ;
        if (original_ID == NULL_VALUE)
                fprintf(output_file, "%d", this->get_ID()) ;
        else fprintf(output_file, "%d", original_ID) ;
        fprintf(output_file, "\n\n") ;
```

```
        write_grammar(false, false) ;

} // grammar::clean_up

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Gets the integer position of the original pattern associated
**        with this alignment element.
**
** CALLING SEQUENCE:
**
**        int alignment_element::get_orig_patt_int_pos()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

int alignment_element::get_orig_patt_int_pos()
{
        symbol *s1 = (symbol *)get_el_obj() ;
        if (s1 == NIL) return(NULL_VALUE) ;
        else return(original_pattern->find_int_pos_by_child(s1)) ;
} // alignment_element::get_orig_patt_int_pos

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        For diagnostic purposes.
**
** CALLING SEQUENCE:
**
**        void alignment_element::show_al_structure()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        Printed output
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void alignment_element::show_al_structure()
{
        symbol *symbol1 ;

        if (el_obj != NIL)
        {
                fprintf(output_file, "Symbol ") ;
                symbol1 = (symbol *)el_obj ;

                fprintf(output_file, "%s", symbol1->get_name()) ;
        }
        else fprintf(output_file, "No symbol") ;
        fprintf(output_file, "%s%d%s%d%s",
```

```
                ", same_column_above ", same_column_above,
                ", same_column_below ", same_column_below,
                ", original_pattern ") ;
        if (original_pattern != NIL)
                original_pattern->print_ID() ;
        else fprintf(output_file, "NIL") ;
        fprintf(output_file, "%s%d%s",
                ",\norig_patt_int_pos ", get_orig_patt_int_pos(), "\n") ;
} // alignment_element::show_al_structure

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        From information in symbol_types_in_old this method assigns
**        a frequency value to 'this' symbol and a bit_cost.
**
** CALLING SEQUENCE:
**
**        void symbol::assign_frequency_and_cost()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        symbol_types_in_old
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void symbol::assign_frequency_and_cost()
{
        symbol *symbol_type ;

        list_for(symbol_type, symbol, symbol_types_in_old)
        {
                if (strcmp(symbol_type->get_name(), this->get_name()) != NIL)
                        continue ;
                set_frequency(symbol_type->get_frequency()) ;
                set_bit_cost(symbol_type->get_bit_cost()) ;
                return ;
        }

        // No match was found for 'this' symbol in symbol_types_in_old
        // so it must be given arbitrary values for frequency
        // and bit_cost.

        set_frequency(1) ;
        if (type == LEFT_BRACKET || type == RIGHT_BRACKET)
                set_bit_cost(BRACKET_BIT_COST) ;
        else if (type == CONTEXT_SYMBOL || type == UNIQUE_ID_SYMBOL)
                set_bit_cost(average_CONTEXT_SYMBOL_type_cost) ;
        else set_bit_cost(average_symbol_type_cost) ;
} // symbol::assign_frequency_and_cost

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Deletion of a symbol
**
** CALLING SEQUENCE:
**
**        symbol::~symbol()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
```

```
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

symbol::~symbol()
{
        delete[] name ;
        delete[] al_el_array ; // Notice
                // that deletion of al_el_array does *not* delete
                // any el_obj which may be attached to one or
                // alignment elements in the al_el_array.
} // symbol::~symbol

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Constructor for class symbol.
**
** CALLING SEQUENCE:
**
**        symbol::symbol(char *nm, int al_depth, int hn_ID) : base_object()
**
** FORMAL ARGUMENTS:
**
**        Return value:
**
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

symbol::symbol(char *nm, int al_depth, int hn_ID) : base_object()
{
        name = NIL ;
        if (nm == NIL) {type = NULL_VALUE ;}
        else set_name(nm) ;
        bit_cost = NULL_VALUE ;
        status = NULL_VALUE ;
        type = NULL_VALUE ;
        sequence_depth = al_depth ;
        h_node_ID = hn_ID ;
        symbol_is_a_hit = false ;
        frequency = 1 ;
        if (al_depth == NULL_VALUE || al_depth == 1) al_el_array = NIL ;
        else
        {
                al_el_array = new alignment_element[al_depth] ;
                for (int i = 0; i < al_depth; i++)
                        al_el_array[i].set_el_obj(NIL) ;
        }
} // symbol::symbol

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Makes a copy of a symbol. If the symbol contains 2 or more rows,
**        it contains a new al_el_array[] but all the pointers within this
**        array are to the original structures pointed to from the symbol
**        which is being copied. Those structures are not themselves copied.
**
** CALLING SEQUENCE:
**
**        symbol::symbol(symbol &s) : base_object(s)
**
```

```
** FORMAL ARGUMENTS:
**
**        Return value:        NONE
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

symbol::symbol(symbol &s) : base_object(s)
{
        // The value of el_obj for the copy points to the start of
        // a new array of al_els.

        int row1 ;

        name = NIL ;
        name = new char[strlen(s.get_name()) + 1] ;
        strcpy(name, s.get_name()) ;
        bit_cost = s.get_bit_cost() ;
        type = s.get_type() ;
        status = s.get_status() ;
        sequence_depth = s.get_sequence_depth() ;
        h_node_ID = s.get_h_node_ID() ;
        symbol_is_a_hit = s.is_a_hit() ;
        frequency = s.get_frequency() ;

        if (sequence_depth != NULL_VALUE && sequence_depth != 1)
        {
                // Patterns and symbols with a depth of 1
                // do not have any al_el_array[].

                al_el_array = new alignment_element[sequence_depth] ;

                for (row1 = 0; row1 < sequence_depth; row1++)
                        *(al_el_array + row1) = *((s.al_el_array) + row1) ;
        }
        else al_el_array = NIL ;
} // symbol::symbol(&s)

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks down the column of an alignment to see whether there
**        are any symbols in the column that have the status CONTENTS. If
**        there is one or more such symbols, the method returns true,
**        otherwise it returns false.
**
** CALLING SEQUENCE:
**
**        bool symbol::contains_contents_symbol()
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if the column contains a CONTENTS symbol,
**                                    false otherwise.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

bool symbol::contains_contents_symbol()
```

```
{
        symbol *symbol1 ;
        int i ;

        for (i = 0; i < sequence_depth; i++)
        {
                symbol1 = get_row_symbol(i) ;
                if (symbol1 != NIL)
                {
                        if (symbol1->get_status() == CONTENTS) return(true) ;
                }
        }

        return(false) ;

} // symbol::contains_contents_symbol

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Makes a copy of 'this' but without any information about rows. It
**        assumes a depth of 1.
**
** CALLING SEQUENCE:
**
**        symbol *symbol::shallow_copy()
**
** FORMAL ARGUMENTS:
**
**        Return value:
**
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

symbol *symbol::shallow_copy()
{
        symbol *copy_symbol = new symbol(this->get_name(), 1, NULL_VALUE) ;

        copy_symbol->set_bit_cost(this->get_bit_cost()) ;
        copy_symbol->set_type(this->get_type()) ;
        copy_symbol->set_status(this->get_status()) ;
        copy_symbol->set_sequence_depth(1) ;
        copy_symbol->set_h_node_ID(this->get_h_node_ID()) ;
        copy_symbol->set_is_a_hit(this->is_a_hit()) ;
        copy_symbol->set_al_el_array(NIL) ;

        return(copy_symbol) ;

} // symbol::shallow_copy

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        For a column within an alignment, puts in the correct values in
**        for same_column_above and same_column_below. The method also
**        sets symbol_is_a_hit as false if the column contains one symbol
**        and sets it as true if the column contains more than one symbol.
**
** CALLING SEQUENCE:
**
**        void symbol::set_symbol_matches()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
```

```
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        Values for same_column_above and same_column_below in
**        column1.
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void symbol::set_symbol_matches()
{
        int i, j, column_depth = this->get_sequence_depth(),
                symbol_counter = 0 ;
        alignment_element *al_el1, *al_el2 ;

        for (i = 0; i < column_depth; i++)
        {
                al_el1 = this->get_al_el(i) ;
                if (al_el1->get_el_obj() == NIL) continue ;
                symbol_counter++ ;
                for (j = i + 1; j < column_depth; j++)
                {
                        al_el2 = this->get_al_el(j) ;
                        if (al_el2->get_el_obj() != NIL) break ;
                }
                if (j >= column_depth) break ;
                al_el1->set_same_column_below(j) ;
                al_el2->set_same_column_above(i) ;
        }

        if (symbol_counter == 1) symbol_is_a_hit = false ;
        else if (symbol_counter > 1) symbol_is_a_hit = true ;
        else abort_run("Invalid value for symbol_count \
                in symbol::set_symbol_matches()") ;
} // symbol::set_symbol_matches

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        For diagnostic purposes.
**
** CALLING SEQUENCE:
**
**        void symbol::show_al_structure()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        Printout of the name of the column
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void symbol::show_al_structure()
{
        int i ;

        fprintf(output_file, "%s%d%s",
                "sequence_depth = ", sequence_depth,
                ", hit node ID = ") ;

        if (h_node_ID == 0) fprintf(output_file, "NIL") ;
        else if (h_node_ID == NULL_VALUE)
                fprintf(output_file, "NULL_VALUE") ;
        else fprintf(output_file, "%d",
                h_node_ID) ;
```

```
        fprintf(output_file, ", is_a_hit = ") ;

        if (symbol_is_a_hit)
                fprintf(output_file, "true") ;
        else fprintf(output_file, "false") ;

        fprintf(output_file, "\n") ;
        for (i = 0; i < sequence_depth; i++)
                al_el_array[i].show_al_structure() ;
        fprintf(output_file, "\n") ;
} // symbol::show_al_structure

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Recursive copy constructor for a tree object.
**
** CALLING SEQUENCE:
**
**        tree_object::tree_object(tree_object &tr_obj) : base_object(tr_obj)
**
** FORMAL ARGUMENTS:
**
**        Return value:        NONE
**
**        tr_obj:         The object to be copied.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

tree_object::tree_object(tree_object &tr_obj) : base_object(tr_obj)
{
        base_object *child, *new_child ;

        first_element = NIL ;
        current_el_pos = NIL ;
        number_of_children = tr_obj.number_of_children ;

        // Now make a 'deep' copy of the structure of sequence columns and
        // sequence elements.

        tr_obj.initialise() ;
        while (child = tr_obj.get_next_child())
        {
                // The following statement makes a copy of al_col2 including
                // copies of the constituent al_els.

                new_child = child->clone() ;

                this->add_child(new_child) ;
        }

        this->mark_parent_and_int_positions_non_recursive() ; // take care!
                // This alters the value of the 'parent' field.

} // tree_object::tree_object(tree_object &)

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Sorts the children of a tree_object by their compression_difference
**        scores.
**
**        It is assumed that all the children of 'this' are tree objects
**        (eg patterns).
**
** CALLING SEQUENCE:
```

```
**
**        void tree_object::sort_children()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        Children in sort order
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void tree_object::sort_children()
{
        cost_tree_object *child, *best_child ;
        group *group_of_children = new group ;
        double best_CD, child_CD ;

        while (true)
        {
                if (this->is_empty()) break ;

                // Find the child with the highest CD.

                best_CD = LOW_VALUE ;
                best_child = NIL ;
                list_for(child, cost_tree_object, this)
                {
                        child_CD = child->get_compression_difference() ;
                        if (child_CD > best_CD)
                        {
                                best_CD = child_CD ;
                                best_child = child ;
                        }
                }

                if (best_child == NIL)
                        abort_run("No best child found in tree_object::sort_children()") ;

                this->extract_child(best_child) ;
                group_of_children->add_child(best_child) ;
        }

        // Now put the children back.

        while (child = (cost_tree_object *)group_of_children->
                extract_first_child())
                        this->add_child(child) ;

        delete group_of_children ;

} // tree_object::sort_children

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Arranges the symbols in a sequence or a group in alpha-numeric order of
**        their name fields. The method is not valid (and will fail) if
**        the sequence or group is not composed entirely of symbols.
**
** CALLING SEQUENCE:
**
**        void tree_object::sort_by_name()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
```

```
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void tree_object::sort_by_name()
{
        int i, string_comparison, fe_symbol_array ;
        symbol *symbol_array[LARGE_SCRATCH_ARRAY_SIZE], *symbol1, *symbol2 ;
        const char *s1, *s2 ;
        base_object *obj1 ;

        // Transfer children from 'this' to symbol_array[] and release
        // from 'this'.

        this->initialise() ;
        i = 0 ;
        while (obj1 = this->get_next_child())
        {
                if (obj1->is_symbol() == false)
                        abort_run("Invalid object in sort_by_name()") ;
                symbol1 = (symbol *)obj1 ;
                symbol_array[i++] = symbol1 ;
                if (i >= LARGE_SCRATCH_ARRAY_SIZE)
                        abort_run("Array overflow in sort_by_name") ;
        }
        fe_symbol_array = i ;
        this->release_children() ;

        // Now transfer them back to this, inserting them in
        // alpha-numeric order.

        for (i = 0; i < fe_symbol_array; i++)
        {
                symbol1 = symbol_array[i] ;
                if (this->is_empty())
                {
                        this->add_child(symbol1) ;
                        continue ;
                }

                // There is at least one symbol in this ;

                this->initialise() ;
                while (symbol2 = (symbol *)this->get_next_child())
                {
                        s1 = symbol1->get_name() ;
                        s2 = symbol2->get_name() ;
                        string_comparison = strcmp(s1, s2) ;
                        if (string_comparison < 0)
                        {
                                this->precede(symbol1, symbol2) ;
                                goto L1 ;
                        }
                }

                // If this point is reached, this means that symbol1 comes
                // after the last symbol in this.

                this->add_child(symbol1) ;

                L1: ;
        }
} // tree_object::sort_by_name

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Adds a new object to the end of the message-receiving
**        object. If the new object is NIL, nothing happens.
**
**        Notice that the position of current_el_pos
**        is not changed by adding a new child to an object.
**
** CALLING SEQUENCE:
**
```

```
**        void tree_object::add_child(base_object *child)
**
** FORMAL ARGUMENTS:
**
**        Return value:        None.
**
**        child:                        The new base_object to be added.
**
** IMPLICIT INPUTS:
**
**        Functions:                new_element
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE. Notice, in particular, that the position of current_el_pos
**        is not changed by adding a new child to an object.
*/

void tree_object::add_child(base_object *child)
{
        list_element *temp1, *temp2, *next ;

        if (child == NIL) return ;
        temp1 = new list_element(child) ;
        temp2 = first_element ;
        if (temp2 == NIL)
        {
                first_element = temp1 ;
                this->mark_parent_and_int_positions_non_recursive() ; // take
                // care! This alters the value of the 'parent' field.
                return ;
        }
        while (next = temp2->get_next()) temp2 = next;
        temp2->set_next(temp1) ;
        this->mark_parent_and_int_positions_non_recursive() ; // take
        // care! This alters the value of the 'parent' field.
} /* tree_object::add_child */

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Adds a new object to the beginning of 'this' object.
**        If the new object is NIL, nothing happens.
**
** CALLING SEQUENCE:
**
**        void tree_object::add_child_at_start(base_object *child)
**
** FORMAL ARGUMENTS:
**
**        Return value:        None.
**
**        child:                        The new object to be added.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void tree_object::add_child_at_start(base_object *child)
{
        list_element *temp1, *temp2 ;

        if (child == NIL)
                abort_run("NIL child for add_child_at_start") ;
        temp1 = new list_element(child) ;
        temp2 = first_element ;
        first_element = temp1 ;
        temp1->set_next(temp2) ;
```

```
        this->mark_parent_and_int_positions_non_recursive() ; // take
        // care! This alters the value of the 'parent' field.
} /* tree_object::add_child_at_start */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Extracts (unlinks) an specific object from a composite object.
**
** CALLING SEQUENCE:
**
**        void tree_object::extract_child(base_object *child) ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        void.
**
**        child:                        Node to be extracted.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        The object is removed from the parent structure but is not deleted
**        (i.e. it's heap storage is still allocated).
*/

void tree_object::extract_child(base_object *child)
{
        list_element *position, *posx;
        base_object *temp;

        if (child == NIL) abort_run("Invalid parameter to extract_child") ;
        position = NIL ;
        temp = this->get_next_child_by_el_pos(&position) ;

        if (temp == child)
        {
                first_element = position->get_next();
                delete position ;
                this->mark_parent_and_int_positions_non_recursive() ; // take
                // care! This alters the value of the 'parent' field.
                return ;
        }

        while (true)
        {
                posx = position;
                temp = this->get_next_child_by_el_pos(&position) ;
                if (temp == NIL)
                        abort_run("Child not found in extract_child") ;
                if (temp == child) break;
        }

        posx->set_next(position->get_next()) ;
        delete position ;
        this->mark_parent_and_int_positions_non_recursive() ; // take
        // care! This alters the value of the 'parent' field.
} /* tree_object::extract_child */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        'Gets' the children of an object in succession, ie it fetches
**        their addresses without disturbing their positions in the object.
**        If the value of *position is NIL, the function
**        gets the first child and sets *position to the position of that
**        child. Otherwise it gets the child following *position and
**        updates *position. It returns NIL when there are no more children
**        to be fetched.
**
** CALLING SEQUENCE:
**
```

```
**        base_object *tree_object::get_next_child()
**
** FORMAL ARGUMENTS:
**
**        Return value:        The child being fetched. NIL if there are no
**                                    more children to be fetched.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

base_object *tree_object::get_next_child()
{
        if (current_el_pos == NIL)
        {
                /* This is the signal to get the first child of 'this',
                if any. */

                current_el_pos = this->first_element;
                if (current_el_pos == NIL) return(NIL) ;
                else return((current_el_pos)->get_el_obj()) ;
        }
        else
        {
                /* In this case up-date current_el_pos
                and get the corresponding child. */

                current_el_pos = (current_el_pos)->get_next() ;
                if (current_el_pos == NIL)
                        return(NIL) ;
                else return((current_el_pos)->get_el_obj()) ;
        }
} /* tree_object::get_next_child */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        'Gets' the children of an object in succession, ie it fetches
**        their addresses without disturbing their positions in the object.
**        If the value of *position is NIL, the function
**        gets the first child and sets *position to the position of that
**        child. Otherwise it gets the child following *position and
**        updates *position. It returns NIL when there are no more children
**        to be fetched.
**
** CALLING SEQUENCE:
**
**        base_object *tree_object::get_next_child_by_el_pos(list_element
**                                    **position)
**
** FORMAL ARGUMENTS:
**
**        Return value:        The child being fetched. NIL if there are no
**                                    more children to be fetched.
**
**        position:               The address of the address of the element
**                                    of the child which was last returned.
**
** IMPLICIT INPUTS:
**
**        position
**
** IMPLICIT OUTPUTS:
**
**        The value of *position in the function which calls this
**        function is up-dated.
**
** SIDE EFFECTS:
**
**        NONE
*/
```

```
base_object *tree_object::get_next_child_by_el_pos(list_element **position)
{
        if (*position == NIL)
        {
                /* This is the signal to get the first child of 'this',
                if any. */

                *position = this->first_element;
                if (*position == NIL) return(NIL) ;
                else return((*position)->get_el_obj()) ;
        }
        else
        {
                /* In this case up-date *position and get the corresponding
                child. */

                *position = (*position)->get_next() ;
                if (*position == NIL)
                        return(NIL) ;
                else return((*position)->get_el_obj()) ;
        }
} /* tree_object::get_next_child_by_el_pos */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        'Gets' the list elements of an object in (reverse) succession,
**        ie it fetches their addresses without disturbing their positions
**        in the object. If the value of position is NIL, the function
**        gets the last el_pos. Otherwise it gets the el_pos preceding *position.
**        It returns NIL when there are no more list_elements to be fetched.
**
** CALLING SEQUENCE:
**
**        list_element *tree_object::get_preceding_el_pos(list_element *position)
**
** FORMAL ARGUMENTS:
**
**        Return value:        The list_element being fetched. NIL if there are no
**                                    more list_elements to be fetched.
**
**        position:               The reference position.
**
** IMPLICIT INPUTS:
**
**        position
**
** IMPLICIT OUTPUTS:
**
**        NONE.
**
** SIDE EFFECTS:
**
**        NONE
*/

list_element *tree_object::get_preceding_el_pos(list_element *position)
{
        list_element *pos1, *pos2 ;

        if (first_element == NIL)
                return(NIL) ;

        if (position == NIL)
        {
                // This is the signal to get the last list_element of 'this',
                // if any.

                return(get_last_el_pos()) ;
        }

        // position is not NIL.

        pos1 = NIL ;
        pos2 = first_element ;
        while (pos2 != position)
        {
                pos1 = pos2 ;
                pos2 = pos2->get_next() ;
        }
```

```
        return(pos1) ;

} // tree_object::get_preceding_el_pos

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        'Gets' the child of an object preceding a given child. If the
**        given child is the first child, or if there is only one child,
**        the function returns NIL. If the given child is NIL,
**        the function returns an error.
**
** CALLING SEQUENCE:
**
**        base_object *tree_object::get_preceding_child(base_object *child)
**
** FORMAL ARGUMENTS:
**
**        Return value:        The child before child. NIL if there is
**                                    only one child in 'this'.
**
**        child:                       The given child whose predecessor is to be found.
**
** IMPLICIT INPUTS:
**
**        position
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

base_object *tree_object::get_preceding_child(base_object *child)
{
        base_object *preceding_child = NIL, *child1 ;
        list_element *pos1 ;

        if (child == NIL) abort_run("NIL parameter for get_preceding_child") ;

        pos1 = NIL ;
        while (child1 = this->get_next_child_by_el_pos(&pos1))
        {
                if (child1 == child) return(preceding_child) ;
                preceding_child = child1 ;
        }

        abort_run("child not found in get_preceding_child") ;

        return(NIL) ;
} /* tree_object::get_preceding_child */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        'Gets' the children of an object in (reverse) succession, ie it fetches
**        their addresses without disturbing their positions in the object.
**        If the value of position is NIL, the function
**        gets the last child and sets *position to the position of that
**        child. Otherwise it gets the child preceding *position and
**        updates *position. It returns NIL when there are no more children
**        to be fetched.
**
** CALLING SEQUENCE:
**
**        base_object *tree_object::
**                get_preceding_child_by_el_pos(list_element **position)
**
** FORMAL ARGUMENTS:
**
**        Return value:        The child being fetched. NIL if there are no
**                                    more children to be fetched.
**
**        position:                The address of the address of the element
**                                    of the child which was last returned.
**
```

```
** IMPLICIT INPUTS:
**
**        position
**
** IMPLICIT OUTPUTS:
**
**        The value of *position in the function which calls this
**        function is up-dated.
**
** SIDE EFFECTS:
**
**                NONE
*/

base_object *tree_object::get_preceding_child_by_el_pos(list_element **position)
{
        base_object *child ;
        list_element *pos1, *pos2 ;

        if (first_element == NIL)
        {
                *position = NIL ;
                return(NIL) ;
        }

        if (*position == NIL)
        {
                /* This is the signal to get the last child of 'this',
                if any. */

                *position = this->get_last_el_pos() ;
                child = this->get_child_by_el_pos(position) ;
                return(child) ;
        }

        /* *position is not NIL. Up-date *position and get the
        corresponding child. */

        pos1 = NIL ;
        pos2 = first_element ;
        while (pos2 != *position)
        {
                pos1 = pos2 ;
                pos2 = pos2->get_next() ;
        }

        *position = pos1 ;
        if (pos1 == NIL) return(NIL) ;
        else return(pos1->get_el_obj()) ;

} /* tree_object::get_preceding_child_by_el_pos */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Tests whether a child object is the last child of 'this' parent.
**
** CALLING SEQUENCE:
**
**        bool tree_object::this_is_last_child(base_object *child)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if child is the last child, false otherwise.
**
**        child:                        The child to be tested.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/
```

```
bool tree_object::this_is_last_child(base_object *child)
{
        if (child == NIL) abort_run("NIL child in this_is_last_child") ;
        list_element *el_pos1 = this->get_el_pos_by_child(child) ;
        if (el_pos1->get_next() == NIL) return(true) ;
        else return(false) ;
} // tree_object::this_is_last_child

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Recursively marks the 'parent' and 'position' fields of a tree_object.
**        This function also sets the value of 'number_of_children'.
**
** CALLING SEQUENCE:
**
**        void tree_object::mark_parent_and_int_positions_recursive()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void tree_object::mark_parent_and_int_positions_recursive()
{
        list_element *pos1 ;
        int position = 0 ;
        tree_object *obj ;

        pos1 = first_element ;
        if (pos1 == NIL) return ;
        while (pos1 != NIL)
        {
                pos1->set_parent(this) ;
                pos1->set_position(position++) ;
                obj = (tree_object *)pos1->get_el_obj() ;
                obj->mark_parent_and_int_positions_recursive() ;
                pos1 = pos1->get_next() ;
        }
        number_of_children = position ;
} /* tree_object::mark_parent_and_int_positions_recursive */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Marks the 'parent' and 'position' fields of a tree_object. This is the
**        non-recursive version of
**        tree_object::mark_parent_and_int_positions_recursive().
**
**        Notice that this method does not disturb the value of
**        current_el_pos.
**
**        But take care! This method alters the value of the 'parent' field.
**
** CALLING SEQUENCE:
**
**        void tree_object::mark_parent_and_int_positions_non_recursive()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
```

```
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void tree_object::mark_parent_and_int_positions_non_recursive()
{
        list_element *pos1 ;
        int position = 0 ;

        pos1 = first_element ;
        if (pos1 == NIL) return ;
        while (pos1 != NIL)
        {
                pos1->set_parent(this) ;
                pos1->set_position(position++) ;
                pos1 = pos1->get_next() ;
        }
        number_of_children = position ;
} /* tree_object::mark_parent_and_int_positions_non_recursive */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        'Gets' the child of an object by its ordinal position, ie it fetches
**        its address without disturbing its position in the object, using
**        ordinal position to determine which is accessed.
**
** CALLING SEQUENCE:
**
**        base_object *tree_object::get_child_by_int_pos(int driving_patt_int_pos)
**
** FORMAL ARGUMENTS:
**
**        Return value:                        The child being fetched. NIL if there is no
**                                                    child at the given position.
**
**        driving_patt_int_pos:        The ordinal position (first position is 0).
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        The value of current_el_pos in 'this' is set to the
**        value corresponding to the child which is returned.
**
** SIDE EFFECTS:
**
**        NONE
*/

base_object *tree_object::get_child_by_int_pos(int driving_patt_int_pos)
{
        base_object  *child ;
        int counter = 0 ;

        this->initialise() ;
        child = this->get_next_child() ;
        if (child == NIL) return(NIL) ;
        while (counter++ < driving_patt_int_pos)
        {
                child = this->get_next_child() ;
                if (child == NIL) return(NIL) ;
        }
        return(child) ;
} /* tree_object ::get_child_by_int_pos */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Return a pointer (and element) to the last object within 'this'
**        object. If there are no children, the function returns
**        NIL.
**
```

```
** CALLING SEQUENCE:
**
**        base_object *tree_object::get_last_child() ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        Pointer to the last object within 'this' object.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        None, but see 'position' above.
**
** SIDE EFFECTS:
**
**        NONE
*/

base_object *tree_object::get_last_child()
{
        list_element *position = first_element ;
        if (position == NIL) return(NIL) ;
        while (position != NIL)
        {
                current_el_pos = position ;
                position = position->get_next() ;
        }
        return(current_el_pos->get_el_obj()) ;
} /* tree_object::get_last_child */

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Adds a new child to 'this' node so that it immediately
**        precedes another specified child within the parent object.
**        This is the complementary function to 'follow'.
**
** CALLING SEQUENCE:
**
**        void tree_object::precede(base_object *child1, base_object *child2)
**
** FORMAL ARGUMENTS:
**
**        Return value:        None.
**
**        child1:                 The new object to be inserted before child2.
**        child2:                 The child that child1 is to be inserted
**                                        in front of.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void tree_object::precede(base_object *child1, base_object *child2)
{
        list_element *tempx, *temp1, *temp2 ;

        if (child1 == NIL || child2 == NIL || child1 == child2)
                abort_run("Invalid parameter(s) for precede") ;
        temp2 = first_element ;
        if (temp2 == NIL) abort_run("child2 not found in precede") ;

        if (temp2->get_el_obj() == child2)
        {
                temp1 = new list_element(child1) ;
                temp1->set_next(temp2) ;
                first_element = temp1 ;

                // Revise position values for list_elements which follow
```

```
                // the inserted child1.

                this->mark_parent_and_int_positions_non_recursive() ; // take
                // care! This alters the value of the 'parent' field.
                return ;
        }

        // child2 is not the first object in the list.

        tempx = temp2;
        temp2 = temp2->get_next() ;
        while (temp2 != NIL)
        {
                if (temp2->get_el_obj() == child2)
                {
                        temp1 = new list_element(child1) ;
                        tempx->set_next(temp1) ;
                        temp1->set_next(temp2) ;

                        // Revise position values for list_elements which
                        // follow the inserted child1.

                        this->mark_parent_and_int_positions_non_recursive() ;
                        // take care! This alters the value of
                        // the 'parent' field.
                        return;
                }
                tempx = temp2;
                temp2 = temp2->get_next() ;
        }

        abort_run("child2 not found in precede") ;
} /* tree_object::precede */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Adds a new child to 'this' node so that it immediately
**        follows another specified child within the parent object.
**        This is the complementary function to 'precede'.
**
** CALLING SEQUENCE:
**
**        void tree_object::follow(base_object *child1, base_object *child2)
**
** FORMAL ARGUMENTS:
**
**        Return value:        None.
**
**        child1:                 child within the parent object that child2
**                                        is to follow
**        child2:                        Object to be inserted after child1.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void tree_object::follow(base_object *child1, base_object *child2)
{
        list_element *temp1, *temp2 ;

        if (child1 == NIL || child2 == NIL || child1 == child2)
                abort_run("Invalid parameter(s) for follow") ;;
        temp1 = first_element ;
        while (temp1 != NIL)
        {
                if (temp1->get_el_obj() == child1)
                {
                        temp2 = new list_element(child2) ;
                        temp2->set_next(temp1->get_next()) ;
                        temp1->set_next(temp2) ;
```

```
                        // Revise position values for list_elements which
                        // follow the inserted child.

                        this->mark_parent_and_int_positions_non_recursive() ;
                        // take care! This alters the value of
                        // the 'parent' field.
                        return;
                }
                temp1 = temp1->get_next() ;
        }

        abort_run("child1 not found in follow") ;
} /* tree_object::follow */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Returns a stored integer representing the ordinal position of a child
**        within its parent (starting at 0).
**
** CALLING SEQUENCE:
**
**        int tree_object::get_int_pos_by_child(base_object *child) ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        Position of child.
**
**        child:                        The child whose position is to be found.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

int tree_object::get_int_pos_by_child(base_object *child)
{
        list_element *pos1 ;

        if (child == NIL) abort_run("NIL child in get_int_pos_by_child") ;
        pos1 = first_element ;
        while ((pos1) != NIL)
        {
                if ((pos1)->get_el_obj() == child) break;
                pos1 = (pos1)->get_next() ;
        }
        if ((pos1) == NIL)
                abort_run("No child found in get_int_pos_by_child") ;
        return((pos1)->get_position()) ;
} /* tree_object::get_int_pos_by_child */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Returns a stored integer representing the ordinal position of a child
**        within its parent (starting at 0). By contrast, with
**        tree_object::get_int_pos_by_child(), this method obtains the
**        position by counting. It does not set the value in the
**        list element corresponding to the child.
**
** CALLING SEQUENCE:
**
**        int tree_object::find_int_pos_by_child(base_object *child) ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        Position of child.
**
**        child:                        The child whose position is to be found.
**
** IMPLICIT INPUTS:
```

```
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

int tree_object::find_int_pos_by_child(base_object *child)
{
        list_element *pos1 ;
        int counter = 0 ;

        if (child == NIL)
                abort_run("NIL child in find_int_pos_by_child().") ;
        pos1 = first_element ;
        while ((pos1) != NIL)
        {
                if ((pos1)->get_el_obj() == child) break;
                pos1 = (pos1)->get_next() ;
                counter++ ;
        }
        if ((pos1) == NIL)
                abort_run("No child found in find_int_pos_by_child().") ;
        return(counter) ;
} /* tree_object::find_int_pos_by_child */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Unlinks the top-level children of 'this' object.  The storage
**        for the children is left allocated but the elements between the
**        'this' and the children are deleted.
**
** CALLING SEQUENCE:
**
**        void tree_object::release_children()
**
** FORMAL ARGUMENTS:
**
**        Return value:        None.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        The tree_object storage for the children is still allocated.
*/

void tree_object::release_children()
{
        base_object *child ;
        list_element *position ;

        position = NIL ;
        child = this->get_next_child_by_el_pos(&position) ;
        while (child != NIL)
        {
                first_element = position->get_next() ;
                delete position ;
                position = NIL ;
                child = this->get_next_child_by_el_pos(&position) ;
        }
} /* tree_object::release_children */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Completes the deletion of a tree_object or hit_node. This function is
```

```
**        called from ~hit_node() or ~tree_object().
**
** CALLING SEQUENCE:
**
**        void tree_object::finish_deletion()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void tree_object::finish_deletion()
{
        list_element *position, *temp ;
        base_object *child ;

        if (first_element == NIL) return ;

        position = NIL ;
        child = this->get_next_child_by_el_pos(&position) ;
        while (child != NIL)
        {
                delete child ;
                temp = position ;
                child = this->get_next_child_by_el_pos(&position) ;
                delete temp ;
        }
} // tree_object::finish_deletion.

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks whether the pattern contains any ID symbol that is a
**        copy of a given symbol.
**
** CALLING SEQUENCE:
**
**        bool sequence::contains_copy_of_ID_symbol(symbol *symbol1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if the pattern contains an ID symbol
**                                    that is a copy of symbol1, false otherwise.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

bool sequence::contains_copy_of_ID_symbol(symbol *symbol1)
{
        symbol *symbol2 ;
        int status ;

        /* Run over set until symbol1 is found or CONTENTS symbols
        are reached */

        list_for(symbol2, symbol, this)
        {
                status = symbol2->get_status() ;
```

```
                if (status == CONTENTS) return(false) ;
                if (status != IDENTIFICATION) continue ;

                if (symbol1->name_matches(symbol2))
                        return(true) ;
        }

        return(false) ;

} // sequence::contains_copy_of_ID_symbol

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks first to see whether or not the given pattern has a
**        UNIQUE_ID_SYMBOL. If it has, the method does nothing and exits.
**        If it has not, it creates a unique_id_symbol and adds it to the
**        pattern just before the CONTENTS symbols.
**
** CALLING SEQUENCE:
**
**        void sequence::add_unique_id_symbol()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void sequence::add_unique_id_symbol()
{
        if (has_unique_id_symbol()) return ;

        symbol *symbol1 = make_unique_id_symbol(), *symbol2 ;

        list_for(symbol2, symbol, this)
        {
                if (symbol2->get_status() == CONTENTS
                        || (symbol2->get_type() == RIGHT_BRACKET
                                && symbol2->get_status() == BOUNDARY_MARKER))
                                        break ;
        }

        if (symbol2 == NIL)
                abort_run("Anomaly in sequence::add_unique_id_symbol().") ;

        this->precede(symbol1, symbol2) ;

        fprintf(output_file, "%s%s%s",
                "Unique ID symbol ",
                symbol1->get_name(),
                " added to pattern ") ;

        write_pattern(true, false) ;

} // sequence::add_unique_id_symbol

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks that hits with New symbols form a coherent sequence.
**
** CALLING SEQUENCE:
**
**        bool sequence::new_hits_form_coherent_sequence()
**
** FORMAL ARGUMENTS:
**
```

```
**        Return value:        true if there are no gaps between hit symbols
**                                     within the New pattern, false otherwise.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

bool sequence::new_hits_form_coherent_sequence()
{
        if (sequence_depth == 1) return(false) ; // This is not
                // an alignment and so there are no hits with New
                // symbols.

        symbol *new_symbol1, *col1 ;
        int int_pos1, int_pos2 = NULL_VALUE ;

        list_for(col1, symbol, this)
        {
                new_symbol1 = col1->get_row_symbol(0) ;
                if (new_symbol1 == NIL) continue ;

                int_pos1 = col1->get_row_orig_int_pos(0) ;
                if (int_pos2 != NULL_VALUE)
                {
                        if (int_pos1 - int_pos2 > 1) return(false) ;
                }

                int_pos2 = int_pos1 ;
        }

        return(true) ;

} // sequence::new_hits_form_coherent_sequence

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks whether the CONTENTS symbol of 'this' sequence represent
**        a single reference comprising a LEFT_BRACKET and CONTEXT_SYMBOL
**        and a RIGHT_BRACKET in that order with no other CONTENTS symbols.
**
** CALLING SEQUENCE:
**
**        bool sequence::check_for_single_reference()
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if the sequence is a single reference,
**                                     false otherwise.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

bool sequence::check_for_single_reference()
{
        // Check whether 'this' sequence comprises a single reference
        // to another pattern or class, meaning a LEFT_BRACKET followed
        // by a CONTEXT_SYMBOL followed by a RIGHT_BRACKET and nothing
        // else.

        symbol *symbol1, *found_class_symbol ;
        int counter = 0 ;
```

```
        bool single_reference_found = false ;
        list_for(symbol1, symbol, this)
        {
                counter++ ;

                if (counter == 1)
                {
                        if (symbol1->get_type() == LEFT_BRACKET)
                                continue ;
                        else return(false) ;
                }

                if (counter == 2)
                {
                        if (symbol1->get_type() == CONTEXT_SYMBOL)
                        {
                                found_class_symbol = symbol1 ;
                                continue ;
                        }
                        else return(false) ;
                }

                if (counter == 3)
                {
                        if (symbol1->get_type() == RIGHT_BRACKET)
                        {
                                single_reference_found = true ;
                                continue ;
                        }
                        else return(false) ;
                }
        }

        if (counter == 3 && single_reference_found)
        {
                set_single_reference(true) ;
                return(true) ;
        }
        else return(false) ;

} // sequence::check_for_single_reference

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Using the values in symbol_types_in_old, this method assigns frequency
**        values and bit_cost values to each symbol in 'this' pattern.
**
** CALLING SEQUENCE:
**
**        void sequence::assign_frequencies_and_costs()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        symbol_types_in_old
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void sequence::assign_frequencies_and_costs()
{
        symbol *symbol1 ;

        list_for(symbol1, symbol, this)
                symbol1->assign_frequency_and_cost() ;

} // sequence::assign_frequencies_and_costs

/*****************************************************************/

/*
```

```
** FUNCTIONAL DESCRIPTION:
**
**        Tests the alignment to see whether the alignment has more than
**        one LEFT_BRACKET symbols with the status of BOUNDARY_MARKER. If id
**        does, the alignment is marked as a composite alignment.
**
** CALLING SEQUENCE:
**
**        void sequence::check_for_composite_structure()
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if the alignment is legal as described above.
**                                   false otherwise.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        'this' alignment may be marked as composite.
**
** SIDE EFFECTS:
**
**        NONE
*/

void sequence::check_for_composite_structure()
{
        symbol *symbol1 ;
        int counter = 0 ;

        list_for(symbol1, symbol, this)
        {
                if (symbol1->get_type() == LEFT_BRACKET)
                {
                        if (symbol1->get_status() == BOUNDARY_MARKER)
                                counter++ ;
                }
        }

        if (counter > 1)
                composite_alignment = true ;

} // sequence::check_for_composite_structure

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Tests the alignment to see whether there are any LEFT_BRACKET symbols
**        at a depth greater than 1 that have BOUNDARY_MARKER status. Any such
**        alignment is illegal.
**
** CALLING SEQUENCE:
**
**        bool sequence::is_legal()
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if the alignment is legal as described above.
**                                   false otherwise.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

bool sequence::is_legal()
{
        symbol *symbol1 ;
        int depth = 0 ;
```

```
        list_for(symbol1, symbol, this)
        {
                if (symbol1->get_type() == LEFT_BRACKET)
                {
                        depth++ ;
                        if (depth > 1)
                        {
                                if (symbol1->get_status() == BOUNDARY_MARKER)
                                        return(false) ;
                        }
                }

                if (symbol1->get_type() == RIGHT_BRACKET)
                        depth-- ;
        }

        return(true) ;

} // sequence::is_legal

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Adds a 'reference' to 'this' sequence, where a reference is a left
**        bracket followed by a class symbol followed by a right bracket, all
**        with status CONTENTS.
**
** CALLING SEQUENCE:
**
**        void sequence::add_reference(int context_number)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        context_number:        A number from which the name of a CONTEXT_SYMBOL
**                                        is made for the reference.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        Reference symbols are added to 'this' pattern.
**
** SIDE EFFECTS:
**
**        NONE
*/

void sequence::add_reference(int context_number)
{
        symbol *bracket, *cs ;
        char symbol_name[STRING_LENGTH] ;

        bracket = make_bracket_symbol(LEFT_BRACKET, CONTENTS, 1) ;
        add_child(bracket) ;

        // Make new context symbol.

        cs = new symbol("", 1, NULL_VALUE) ;
        sprintf(symbol_name, "%d", context_number) ;
        cs->set_type(CONTEXT_SYMBOL) ;
        cs->set_status(CONTENTS) ;
    cs->set_name(symbol_name) ;
        cs->assign_frequency_and_cost() ;
        add_child(cs) ;

        bracket = make_bracket_symbol(RIGHT_BRACKET, CONTENTS, 1) ;
        add_child(bracket) ;

} // sequence::add_reference

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Adds a CONTEXT_SYMBOL at the start of the list of
**        IDENTIFICATION symbols at the start of the pattern. The method
```

```
**        checks to see whether a given CONTEXT_SYMBOL is already present
**        before it adds it.
**
** CALLING SEQUENCE:
**
**        bool sequence::add_context_symbol(symbol *context_symbol)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if the context_symbol is added,
**                                     false otherwise.
**
**        context_symbol:        The symbol to be added.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

bool sequence::add_context_symbol(int context_number)
{
        symbol *symbol1, *cs ;
        char symbol_name[STRING_LENGTH] ;

        // Check whether a copy of context_number is already present.
        // If it is, return false, otherwise add the context_symbol
        // and return true.

        sprintf(symbol_name, "%d", context_number) ;
        list_for(symbol1, symbol, this)
        {
                if (symbol1->get_status() == CONTENTS) break ;

                if (symbol1->name_matches_string(symbol_name))
                {
                        fprintf(output_file, "%s%s%s",
                                "Context number ",
                                symbol_name,
                                " duplicates symbol in pattern ") ;
                        write_pattern(true, false) ;
                        return(false) ;
                }
        }

        // Make new context symbol.

        cs = new symbol("", 1, NULL_VALUE) ;
        cs->set_type(CONTEXT_SYMBOL) ;
    cs->set_name(symbol_name) ;
        cs->assign_frequency_and_cost() ;
        cs->set_status(IDENTIFICATION) ;

        // Add the context symbol after the BOUNDARY_MARKER if there is
        // one or at the start of the pattern.

        symbol1 = (symbol *)this->get_first_child() ;
        if (symbol1 != NIL)
        {
                if (symbol1->get_status() == BOUNDARY_MARKER)
                        follow(symbol1, cs) ;
                else this->add_child_at_start(cs) ;
        }
        else this->add_child_at_start(cs) ;

        fprintf(output_file, "%s%s%s",
                "Context symbol ",
                symbol_name,
                " added to pattern ") ;

        write_pattern(true, false) ;

        return(true) ;

} // sequence::add_context_symbol
```

```
/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Using the context symbols in generalisation_list1 (which were
**        created during sequence::create_patterns()), this function
**        generalises the Old patterns by adding context symbols to them
**        according to the following procedure:
**
**        1 For each context symbol in generalisation_list1, compile a list
**        of *other* context symbols that are found in patterns that
**        contain the given context symbol. Makes sure that there are no
**        duplicates in this generalisation_list2.
**
**        2 For each pattern that contains one or more of the context symbols
**        on generalisation_list2, add a copy of the 'given' context symbol
**        under 1. Make sure that there are no duplicates.
**
** CALLING SEQUENCE:
**
**        void generalise_patterns()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        generalisation_list1
**
** IMPLICIT OUTPUTS:
**
**        Context symbols added to Old patterns.
**
** SIDE EFFECTS:
**
**        NONE
*/

void generalise_patterns()
{
        symbol *symbol1, *symbol2, *symbol3 ;
        sequence *pattern1 ;
        int status ;

        #if DIAGNOSTIC6
        fprintf(output_file, "Diagnostic: generalisation_list1:\n\n") ;
        list_for(symbol1, symbol, generalisation_list1)
        {
                fprintf(output_file, "%s%s",
                        symbol1->get_name(),
                        " ") ;
        }
        fprintf(output_file, "\n\n") ;
        #endif

        list_for(symbol1, symbol, generalisation_list1)
        {
                list_for(pattern1, sequence, old_patterns)
                {
                        if (pattern1->contains_copy_of_ID_symbol(symbol1) == false)
                                continue ;

                        // pattern1 contains a copy of symbol1 and symbol1
                        // is an ID-symbol in pattern1. Now add copies
                        // of *other* ID symbols in the pattern to
                        // generalisation_list2, avoiding duplicates.

                        list_for(symbol2, symbol, pattern1)
                        {
                                if (symbol2->get_type() != CONTEXT_SYMBOL)
                                        continue ;
                                status = symbol2->get_status() ;
                                if (status == CONTENTS) break ;
                                if (status != IDENTIFICATION) continue ;
                                if (symbol2->name_matches(symbol1))
                                        continue ;
                                if (generalisation_list2->contains_copy_of(symbol2))
                                        continue ;
                                symbol3 = new symbol(*symbol2) ;
                                generalisation_list2->add_child(symbol3) ;
                        }
```

```
                }

                // Now that generalisation_list2 has been compiled for
                // symbol1, step through the list and, for each symbol,
                // add a copy of symbol1 to every Old pattern that
                // contains a copy of the given symbol, avoiding
                // duplicates.

                #if DIAGNOSTIC6
                fprintf(output_file, "%s%s%s",
                        "Diagnostic: generalisation_list2 for symbol ",
                        symbol1->get_name(),
                        ":\n\n") ;
                list_for(symbol3, symbol, generalisation_list2)
                {
                        fprintf(output_file, "%s%s",
                                symbol3->get_name(),
                                " ") ;
                }
                fprintf(output_file, "\n\n") ;
                #endif

                list_for(symbol2, symbol, generalisation_list2)
                {
                        list_for(pattern1, sequence, old_patterns)
                        {
                                if (pattern1->contains_copy_of_ID_symbol(symbol2) == false)
                                        continue ;

                                // Add a copy of symbol1 to pattern1 unless the
                                // pattern already contains a copy of that symbol.

                                if (pattern1->contains_copy_of(symbol1))
                                        continue ;

                                symbol3 = new symbol(*symbol1) ;
                                symbol3->set_status(IDENTIFICATION) ;
                                symbol3->set_type(CONTEXT_SYMBOL) ;
                                pattern1->add_context_symbol_at_start(symbol3) ;
                        }
                }

                // Delete the children of generalisation_list2 ready
                // for next symbol1.

                generalisation_list2->delete_children()  ;
        }

        // Now step through the Old patterns renumbering positions.

        list_for(pattern1, sequence, old_patterns)
                pattern1->mark_parent_and_int_positions_non_recursive() ;

        // Delete the symbols in generalisation_list1 ready for
        // the next invocation of sequence::create_patterns() and
        // resulting generalisations.

        generalisation_list1->delete_children() ;

} // generalise_patterns

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Derives an encoding of the driving pattern of 'this' alignment
**        when the CONTENTS symbols of the target pattern are not completely
**        matched or the New pattern is not completely matched, directly or
**        indirectly, by a single Old pattern. This method assumes that there
**        is a single target pattern (by contrast with alignments created by
**        combine_alignments() that may contain two or more target patterns).
**
**        The tentative rule embodied in this method is:
**
**        "Make a new pattern from every coherent sequence of two or more
**        matched CONTENTS symbols in the target pattern and
**        likewise for coherent unmatched sequences. Make a new pattern
**        from every coherent sequence of one or more unmatched New symbols.
**        Derive an abstract pattern from the resulting patterns.
**        Include a code pattern for the New pattern." (see sp71_od, %62).
**
**        In SP71, v 7.3, this method has been augmented so that, when there
```

```
**        is a null element in the New pattern or the Old pattern, it creates
**        an explicit null element with a corresponding abstract pattern (as now)
**        but it also creates an abstract pattern without a reference to the
**        disjunctive class containing the null element (see sp71_od, %65). In
**        v 7.4, the null patterns have been dropped.
**
**        In SP71, v 7.7, the method is modified so that every alignment is
**        treated as if all the Old patterns had been unified to make a single
**        Old pattern, as described in sp71_od, %79.
**
** CALLING SEQUENCE:
**
**        void sequence::create_patterns()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void sequence::create_patterns()
{
        // Print out preliminary details.

        fprintf(output_file, "%s%d%s",
                "Learning from alignment ID",
                ID,
                " (") ;
        print_pattern_cycle(false, current_new_pattern) ;
        fprintf(output_file, "):\n\n") ;

        write_alignment(output_file, write_section_chars_length,
                NULL_VALUE, alignment_format) ;

        if (composite_alignment == true)
        {
                fprintf(output_file,
                        "The alignment is composite. Learning not yet implemented.\n\n") ;
                return ;
        }

        if (degree_of_matching == FULL_A)
        {
                fprintf(output_file,
                        "The alignment is complete and there is no learning to be done.\n\n") ;
                return ;
        }

        // Create an array to store details of the alignment. Both the test
        // pattern (== driving pattern) and the target pattern are
        // expanded so that all symbols are shown. Breaks in the matching
        // sequence in the driving pattern or the target pattern or both
        // are marked by a divider column with NIL values.

        struct al_entry
        {
                sequence *new_d_pattern, *new_t_pattern ;
        } al_array[MEDIUM_SCRATCH_ARRAY_SIZE] ;

        int fe_al_array = 0, i, d_int_pos, t_int_pos, previous_d_int_pos,
                previous_t_int_pos, bottom_row = sequence_depth - 1 ;

        sequence *d_pattern = get_row_pattern(0),
                *t_pattern = get_row_pattern(bottom_row)
                *new_d_pattern, *new_t_pattern ;

        symbol *symbol1, *d_symbol, *t_symbol ;

        for (i = 0; i < MEDIUM_SCRATCH_ARRAY_SIZE; i++)
        {
                new_d_pattern = new_t_pattern = NIL ;
        }
```

```
        // Traverse 'this' alignment filling in values in al_array[].

        d_pattern->initialise() ;
        t_pattern->initialise() ;
        list_for(symbol1, symbol, this)
        {
                d_symbol = symbol1->get_row_symbol(0) ;
                if (d_symbol != NIL)
                {

                }



                t_symbol = symbol1->get_row_symbol(bottom_row) ;
        }

} // sequence::create_patterns

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Writes out a pattern or alignment as a flat sequence of symbols.
**
** CALLING SEQUENCE:
**
**        void sequence::write_pattern(bool new_line, bool write_from)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        new_line:                If true, two line-feed characters are printed at
**                                       the end, otherwise they are not.
**        write_from:               true if '(from alignment ID)' is to be written,
**                                       otherwise false.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        Output to output_file
**
** SIDE EFFECTS:
**
**        NONE
*/

void sequence::write_pattern(bool new_line, bool write_from)
{
        symbol *symbol1 ;

        fprintf(output_file, "%s%d", "ID", ID) ;
        if (write_from && derived_from_parsing != NULL_VALUE)
                fprintf(output_file, "%s%d%s",
                " (from ID",
                derived_from_parsing,
                ") (") ;
        else fprintf(output_file, " (") ;

        list_for(symbol1, symbol, this)
        {
                fprintf(output_file, "%s", symbol1->get_name()) ;
                if (current_el_pos->get_next() == NIL)
                        fprintf(output_file, ")*") ;
                else fprintf(output_file, " ") ;
        }

        fprintf(output_file, "%d", frequency) ;
        if (new_line) fprintf(output_file, "\n\n") ;
} // sequence::write_pattern

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
```

```
**        Makes a 'shallow' copy of 'this' sequence, meaning that there
**        is only one row and the depth is 1.
**
** CALLING SEQUENCE:
**
**        sequence *sequence::shallow_copy()
**
** FORMAL ARGUMENTS:
**
**        Return value:        A copy of 'this'.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

sequence *sequence::shallow_copy()
{
        symbol *copy_symbol, *symbol1 ;
        sequence *copy_sequence = new sequence(BASIC_PATTERN) ;

        list_for(symbol1, symbol, this)
        {
                copy_symbol = symbol1->shallow_copy() ;
                copy_sequence->add_child(copy_symbol) ;
        }

        copy_sequence->set_sequence_depth(1) ;
        copy_sequence->set_origin(origin) ;
        copy_sequence->set_total_cost(total_cost) ;

        return(copy_sequence) ;
} // sequence::shallow_copy

//*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Compares two sequences to see whether they match or not. This alignment
**        assumes that the sequence is only a sequence of simple symbols
**        without rows. If there are rows, the method flags an error. An error
**        is also flagged if the two sequences are in fact the same sequence.
**
** CALLING SEQUENCE:
**
**        bool sequence::is_copy_of(sequence *seq1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if the two sequences match, false otherwise.
**
**        seq1:                        The second of the two sequences to be compared.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

bool sequence::is_copy_of(sequence *seq1)
{
        symbol *symbol1, *symbol2 ;
        int seq1_length, seq1_depth ;

        if (seq1 == NIL) abort_run("NIL value for seq1 in \
                sequence::alignment_matches") ;
```

```
        // Check whether the two sequences are, in fact, the same sequence.

        if (seq1 == this)
                abort_run("One sequence compared with itself in is_copy_of().") ;

        /* Compare the lengths of the sequences. */

        seq1_length = seq1->get_number_of_children() ;

        if (number_of_children != seq1_length) return(false) ;

        // Check that sequence depth is not greater than 1 for either sequence.

        seq1_depth = seq1->get_sequence_depth() ;

        if (sequence_depth != 1 || seq1_depth != 1) return(false) ;

        // Check for a match in the basic sequence of symbols.

        symbol2 = (symbol *)seq1->get_first_child() ;
        for (symbol1 = (symbol *)this->get_first_child();symbol1 != NIL ;
                symbol1 = (symbol *)this->get_next_child())
        {
                if (symbol1->name_matches(symbol2) == false) return(false) ;
                symbol2 = (symbol *)seq1->get_next_child() ;
                if (symbol2 == NIL)
                {
                        symbol1 = (symbol *)this->get_next_child() ;
                        break ;
                }
        }

        if (symbol1 != NIL || symbol2 != NIL)
                return(false) ;

        return(true) ;
} /* sequence::is_copy_of */

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        For each column in 'this' alignment, the method finds the values
**        of frequency and bit_cost for the column (from any symbol within
**        the column) and assigns these values to the column itself.
**
** CALLING SEQUENCE:
**
**        void sequence::correct_column_values()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        Corrected values for frequency and bit_cost for each column in 'this'.
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void sequence::correct_column_values()
{
        symbol *col1, *symbol1 ;
        int i ;

        list_for(col1, symbol, this)
        {
                for (i = 1; i < sequence_depth; i++)
                {
                        symbol1 = col1->get_row_symbol(i) ;
                        if (symbol1 != NIL) break ;
                }

                col1->set_frequency(symbol1->get_frequency()) ;
```

```
                col1->set_bit_cost(symbol1->get_bit_cost()) ;
        }
} // sequence::correct_column_values

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Tests whether or not all the symbols from cnp
**        are matched in the alignment.
**
** CALLING SEQUENCE:
**
**        bool sequence::all_new_symbols_matched(sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if all the symbols in cnp
**                                    are matched, false otherwise.
**
**        cnp:                        The New pattern being processed.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

bool sequence::all_new_symbols_matched(sequence *cnp)
{
        // Check that this is an alignment and not a flat pattern.

        if (sequence_depth < 2)
                abort_run("Invalid alignment in sequence::all_new_symbols_matched()") ;

        symbol *col1, *symbol_new1, *symbol_new2 ;
        sequence *pattern_new = get_row_pattern(0) ;

        this->initialise() ;
        list_for(symbol_new1, symbol, pattern_new)
        {
                while (col1 = (symbol *)this->get_next_child())
                {
                        symbol_new2 = col1->get_row_symbol(0) ;
                        if (symbol_new2 != NIL) break ;
                }

                if (col1 == NIL) // The alignment has been exhausted without
                        // finding all the symbols in cnp.
                                return(false) ;

                if (col1->is_a_hit() == false) return(false) ;
                if (symbol_new1 != symbol_new2) return(false) ;
        }

        return(true) ;
} // sequence::all_new_symbols_matched

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Adds LEFT_BRACKET and RIGHT_BRACKET symbols with BOUNDARY_MARKER
**        status to the beginning and end of a pattern if they are
**        not already present. Adds a CONTEXT_SYMBOL and a
**        'discrimination symbol' is added if required.
**
**        At the beginning of this method is a check to see whether the
**        CONTENTS symbols of 'this' pattern comprises a single 'reference'
**        to another pattern or class. If it is, the method returns
**        the CONTEXT_SYMBOL of that reference and indicates, via
**        is_single_reference, that a single reference has been found.
**
```

```
** CALLING SEQUENCE:
**
**        void sequence::add_ID_symbols(symbol *context_symbol,
**                bool add_uid_symbol)
**
** FORMAL ARGUMENTS:
**
**        Return value:                void.
**
**        context_symbol:                A symbol that provides a name for a context.
**                                       If the value is NIL, no context symbol is
**                                       created.
**        add_uid_symbol:                If true, a unique ID symbol is added, otherwise
**                                       it is not.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void sequence::add_ID_symbols(symbol *context_symbol,
        bool add_uid_symbol)
{
        symbol *cs, *first_bracket, *last_bracket, *unique_id_symbol,
                *symbol1 ;
        char *csn ;

        // Create a context symbol, with a name that
        // is either newly-minted or taken from context_symbol.

        if (context_symbol == NIL) cs = NIL ;
        else
        {
                csn = context_symbol->get_name() ;
                cs = new symbol(csn, 1, NULL_VALUE) ;
                cs->set_status(IDENTIFICATION) ;
                cs->set_type(CONTEXT_SYMBOL) ;
                cs->set_bit_cost(context_symbol->get_bit_cost()) ;
        }

        first_bracket =
                make_bracket_symbol(LEFT_BRACKET, BOUNDARY_MARKER, 1) ;
        this->add_child_at_start(first_bracket) ;
        last_bracket = make_bracket_symbol(RIGHT_BRACKET, BOUNDARY_MARKER, 1) ;
        this->add_child(last_bracket) ;

        // If add_uid_symbol, create a unique id symbol and add
        // it to the pattern.

        if (add_uid_symbol)
        {
                unique_id_symbol = make_unique_id_symbol() ;
                this->follow(first_bracket, unique_id_symbol) ;
        }

        // Make a check to see that (a copy of) the cs is not already
        // present in the pattern (see sp70_od, %133). If it is not
        // already present, put the cs after the first bracket.

        if (cs != NIL)
        {
                bool cs_is_already_present = false ;
                list_for(symbol1, symbol, this)
                {
                        if (symbol1->get_type() == LEFT_BRACKET) continue ;
                        if (symbol1->get_status() == CONTENTS) break ;
                        if (symbol1->name_matches(cs))
                        {
                                cs_is_already_present = true ;
                                break ;
                        }
                }

                if (cs_is_already_present == false)
```

```
                        this->follow(first_bracket, cs) ;
        }

        this->mark_parent_and_int_positions_non_recursive() ;
        this->compute_costs() ;

} // sequence::add_ID_symbols

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks whether a newly-formed alignment matches any one formed
**        previously, on this cycle or previous cycles.
**
** CALLING SEQUENCE:
**
**        sequence *sequence::matches_earlier_alignment()
**
** FORMAL ARGUMENTS:
**
**        Return value:        The matching sequence if a match is found,
**                                    NIL otherwise.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

sequence *sequence::matches_earlier_alignment()
{
        sequence *al2 ;
        hit_node *node1 ;
        int row, leaf_index ;

        /* The program searches for matches amongst the sequences
        formed so far in the current cycle. */

        row = 0 ;
        while (node1 = get_leaf_nodes_in_order(&row, &al2, &leaf_index))
        {
                if (node1->get_compression_difference() == fail_score)
                        continue ;
                if (al2 == NIL) continue ;

                // Because each new sequence is always matched
                // against sequences formed in the current cycle, it is
                // assumed that there are no other duplicates amongst
                // the new sequences so it is safe to break out of the
                // loop now.

                if (al2->alignment_matches(this)) return(al2) ;
        }

        // Test whether sequences formed in all previous cycles are
        // to be matched as well.

        // if (!CHECK_ALL_ALIGNMENTS) return(NIL) ;

        // Check whether the newly-formed sequence matches anything
        // produced in previous cycles.

        list_for(al2, sequence, parsing_alignments)
        {
                // Because each new sequence is always matched
                // against sequences formed in the earlier cycles, it is
                // assumed that there are no other duplicates amongst the
                // earlier sequences so it is safe to break out of
                // the loop if a match is found now.

                if (al2->alignment_matches(this)) return(al2) ;
        }
```

```
        return(NIL) ;
} // matches_earlier_alignment

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Writes out a sequence as a flat pattern without details and as
**        a flat pattern with details. If the pattern is an alignment,
**        the rows of the alignment are written out. The method
**        also writes out any code derived from the alignment or sequence.
**
** CALLING SEQUENCE:
**
**        void sequence::write_out_fully(char *header,
**                hit_node *h_node, int write_section_chars_length,
**                int sections_per_page, bool flat_pattern_and_details,
**                sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:                                        void
**
**        header:                                                      The heading.
**        hit_node:                                               The hit node in the case of
**                                                                       alignments derived from the
**                                                                       hit structure. It may be NIL.
**        write_section_chars_length:               The number of character columns
**                                                                       that can be written before the
**                                                                       alignment must be broken into
**                                                                       a new section.
**        sections_per_page:                                 The number of sections of the
**                                                                       alignment that can be written
**                                                                       before a new page must be started
**                                                                       (shown with a Latex command for
**                                                                       use in latex_file).
**                                                                       If the value is NULL_VALUE, no
**                                                                       page break will be inserted.
**        flat_pattern_and_details:               If true, the alignment is printed
**                                                                       as a flat pattern and
**                                                                       also with its details. Otherwise,
**                                                                       these things are not printed.
**        cnp:                                                     The 'current' New pattern
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        Output to output_file
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void sequence::write_out_fully(char *header, hit_node *h_node,
        int write_section_chars_length, int sections_per_page,
        bool flat_pattern_and_details, sequence *cnp)
{
        if (header != NIL)
        {
                fprintf(output_file, "%s%s",
                        header,
                        " (") ;
                print_pattern_cycle(true, cnp) ;
                fprintf(output_file, ")\n\n") ;
        }

        this->print_ID() ;

        if (h_node != NIL)
        {
                fprintf(output_file, " : ") ;
                (h_node->get_driving_pattern())->print_ID() ;
                fprintf(output_file, " : ") ;
                (h_node->get_target_pattern())->print_ID() ;
                fprintf(output_file, " : ") ;
                h_node->print_ID() ;
        }
```

```
        if (sequence_depth > 1)
        {
                fprintf(output_file,
                        "%s%1.2f%s%1.2f%s%1.2f%s%1.2f%s%1.12g%s",
                        ": NSC = ", this->get_new_symbols_cost(),
                        ", EC = ", this->get_encoding_cost(),
                        ", CR = ", this->get_compression_ratio(),
                        ", CD = ", this->get_compression_difference(),
                        ",\nAbsolute P = ", this->get_abs_P(),
                        "\n\n") ;

                this->write_alignment(output_file,
                        write_section_chars_length, sections_per_page,
                        alignment_format) ;
        }
        else fprintf(output_file, ": ") ;

        if (flat_pattern_and_details)
        {
                if (sequence_depth > 1)
                {
                        fprintf(output_file, "Sequence ") ;
                        this->print_ID() ;
                        fprintf(output_file, " as flat pattern:\n\n") ;
                }

                this->write_with_details(true) ;
        }

} // sequence::write_out_fully

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Writes out an alignment of symbols.
**
** CALLING SEQUENCE:
**
**        void sequence::write_alignment_horizontal(FILE *output_file,
**                int write_section_chars_length, int sections_per_page)
**
** FORMAL ARGUMENTS:
**
**        Return value:                        void
**
**        output_file:                                The output_file or the latex_file.
**        write_section_chars_length:           The number of character columns
**                                                    that can be written before the
**                                                    alignment must be broken into
**                                                    a new section.
**        sections_per_page:                                The number of sections of the
**                                                    alignment that can be written
**                                                    before a new page must be started
**                                                    (shown with a Latex command for
**                                                    use in latex_file).
**                                                    If the value is NULL_VALUE, no
**                                                    page break will be inserted.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE.
**
** SIDE EFFECTS:
**
**        NONE
*/

void sequence::write_alignment_horizontal(FILE *output_file,
        int write_section_chars_length, int sections_per_page)
{
        symbol *al_col1, *symbol1 ;
        sequence *pattern1 ;
        int alignment_depth = this->get_sequence_depth(),
                number_of_children = this->get_number_of_children() ;
        int s_row1, s_row2, c_row1, c_col1, end_col,
                symbol_length, c_col_for_link, al_char_depth, max_c_col,
```

```
        temp_c_col, column_width, s_row_lowest_symbol, al_c_depth ;
char *c, *symbol_name ;
alignment_element *al_el1 ;
list_element *temp_el_pos ;
bool highest_symbol_has_been_found, write_char_limit_reached ;

al_c_depth = (alignment_depth * 2) - 1 ; // Calculate the
// depth of the alignment in terms of written out characters.

if (al_c_depth > write_alignment_char_rows)
        abort_run("write_al_chars[][] has too few rows.") ;

if (number_of_children >= max_alignment_length)
        abort_run("max_alignment_length has been exceeded") ;

/* Clear write_al_chars[][], row_array[] and write_al_vacant_slots[][].
The first of these arrays will contain the alignment as the
characters are to be written out. The second one keeps track of
how far through the alignment (from left to right), the
process has got. The third one records (with 'x'),
the character slots which may not contain any non-hit symbol. */

al_col1 = (symbol *)this->get_first_child() ;

if (al_col1 == NIL) abort_run("Anomaly in write_alignment") ;

for (s_row1 = 0; s_row1 < alignment_depth; s_row1++)
{
        row_array[s_row1].row_pattern =
                al_col1->get_row_pattern(s_row1) ;
        row_array[s_row1].patt_last_el_pos = NIL ;
        row_array[s_row1].fe_col_in_write_al_chars = 0 ;
        row_array[s_row1].pattern_is_finished = false ;
        c_row1 = s_row1 * 2 ;
        for (c_col1 = 0; c_col1 < write_al_chars_length; c_col1++)
        {
                write_al_vacant_slots[s_row1][c_col1] = ' ' ;
                write_al_chars[c_row1][c_col1] = ' ' ;
                write_al_chars[c_row1 + 1][c_col1] = ' ' ;
        }
}

// Now fill in characters in write_al_chars[][].

max_c_col = 0 ;
list_for(al_col1, symbol, this)
{
        // if (!al_col1->is_a_hit()) continue ; // EDIT: line commented out.

        highest_symbol_has_been_found = false ;

        // Find the lowest row which contains a symbol.

        s_row_lowest_symbol = NULL_VALUE ;

        for (s_row2 = alignment_depth - 1; s_row2 >= 0; s_row2--)
        {
                symbol1 = al_col1->get_row_symbol(s_row2) ;
                if (symbol1 != NIL)
                {
                        s_row_lowest_symbol = s_row2 ;
                        break ;
                }
        }

        if (s_row_lowest_symbol == NULL_VALUE)
                abort_run("Anomaly in value of s_row_lowest_symbol") ;

        // Find the length of the symbol
        // type for this column.

        if (use_abbreviations == LETTERS || use_abbreviations == DIGITS)
                symbol_name = find_symbol_abbreviation(al_col1->
                get_name()) ;
        else symbol_name = al_col1->get_name() ;

        symbol_length = strlen(symbol_name) ;
        column_width = symbol_length + 1 ;

        // For those rows which contain a hit symbol,
        // fill in the non-hit symbols which precede the hit
        // symbol.
```

```
write_char_limit_reached = false ;
for (s_row1 = 0; s_row1 < alignment_depth; s_row1++)
{
        if (s_row1 > 0) break ; // EDIT: line added.
        symbol1 = al_col1->get_row_symbol(s_row1) ;
        if (symbol1 == NIL) continue ;

        // Fill in the non-hit symbols from after
        // the previous hit for this row (or from the
        // beginning of the pattern) up to but
        // not including the current hit_symbol.
        // If necessary, update the value of max_c_col.

        if (fill_in_non_hit_symbols(s_row1, symbol1,
                &temp_c_col))
                write_char_limit_reached = true ;

        if (temp_c_col > max_c_col) max_c_col = temp_c_col ;
}

if (write_char_limit_reached) goto L1 ;

// Allowing for the width of the 'hit' column, check that
// we are still within the bounds of write_al_chars[][].

if (max_c_col + column_width >= write_al_chars_length)
{
        write_char_limit_reached = true ;
        goto L1 ;
}

// Blank off those sections of write_al_vacant_slots[][]
// which correspond to a NIL symbol in this column.

if (al_col1->is_a_hit()) // EDIT: code added.
{
        end_col = max_c_col + column_width ;
        for (s_row1 = 0; s_row1 < alignment_depth; s_row1++)
        {
                symbol1 = al_col1->get_row_symbol(s_row1) ;
                if (symbol1 == NIL)
                {
                        for (c_col1 = max_c_col; c_col1 < end_col; c_col1++)
                                write_al_vacant_slots[s_row1][c_col1] = 'x' ;
                }
        }
}

// Now, for each row which contains a hit symbol,
// fill in the hit symbol. If the hit symbol is the
// last symbol in that row, mark the row as finished.

// Calculate the column position of the 'link' symbol ('|')
// which connects the hit symbols.

c_col_for_link = max_c_col +
        (column_width - (symbol_length / 2) - 2) ;

for (s_row1 = 0; s_row1 <= s_row_lowest_symbol; s_row1++)
{
        c_row1 = s_row1 * 2 ;
        al_el1 = al_col1->get_al_el(s_row1) ;
        symbol1 = (symbol *)al_el1->get_el_obj() ;
        pattern1 = al_el1->get_original_pattern() ;

        if (symbol1 == NIL)
        {
                if (!highest_symbol_has_been_found) continue ;

                // Fill in a linking symbol for this c_row1
                // and the one below it. We can be sure
                // the lowest symbol has not yet been reached
                // because, if it had, symbol1 would not be
                // equal to NIL.

                write_al_chars[c_row1][c_col_for_link] = '|' ;
                write_al_chars[c_row1+1][c_col_for_link] = '|' ;
                continue ;
        }
        else highest_symbol_has_been_found = true ;

        // symbol1 is not NIL. Now fill in the name
        // of the 'hit' symbol for this row. But first,
```

```
                // mark this row as finished if the hit symbol
                // is the last symbol in the row.

                temp_el_pos = row_array[s_row1].patt_last_el_pos ;
                pattern1->set_current_el_pos(temp_el_pos) ;
                if (pattern1->this_is_last_child(symbol1))
                        row_array[s_row1].pattern_is_finished =
                        true ;

                c_col1 = max_c_col ;
                for (c = symbol_name; *c != '\0'; c++)
                        write_al_chars[c_row1][c_col1++] = *c ;

                if (s_row1 < s_row_lowest_symbol)
                        write_al_chars[c_row1+1][c_col_for_link] = '|' ;

                // Update the value for the first empty column for
                // this row stored in write_al_char_cols[].

                row_array[s_row1].fe_col_in_write_al_chars =
                        c_col1 + 1 ; // Add 1 to allow for the
                // space character following
                // the symbol.
        }

        max_c_col += column_width ;
}

// Finish off any remaining non-hit symbols in each row of
// the alignment.

write_char_limit_reached = false ;
for (s_row1 = 0; s_row1 < alignment_depth; s_row1++)
{
        if (s_row1 > 0) break ; // EDIT: line added.
        if (fill_in_non_hit_symbols(s_row1, NIL, &temp_c_col))
                write_char_limit_reached = true ;
        if (temp_c_col > max_c_col) max_c_col = temp_c_col ;
}

L1: ;

if (write_char_limit_reached)
{
        /* The writing out of the alignment will be
        truncated at this point. The rows which are
        not finished are marked with "...". */

        for (s_row1 = 0; s_row1 < alignment_depth; s_row1++)
        {
                if (row_array[s_row1].pattern_is_finished)
                        continue ;
                c_col1 = max_c_col ;
                c_row1 = s_row1 * 2 ;
                write_al_chars[c_row1][c_col1++] = '.' ;
                write_al_chars[c_row1][c_col1++] = '.' ;
                write_al_chars[c_row1][c_col1++] = '.' ;
        }
        max_c_col += 3 ;
}

/* Now write out the characters in write_al_chars[][]. */

// int sections_written = 0 ;
int first_section_column = 0, last_section_column,
        al_char_length = max_c_col - 1,section_counter = 0 ;

// Calculate the number of character rows in the alignment.

al_char_depth = (alignment_depth * 2) - 1 ;

// Adjust the value of write_section_chars_length in the
// light of the row numbers to be written at the beginning and
// end of each row. For each number, one or two columns is needed
// for the number itself, plus one blank column between the
// number and the alignment.

int last_row = alignment_depth - 1, cols_for_number ;
if (last_row < 10) cols_for_number = 2 ;
else if (last_row >=10 && last_row < 20) cols_for_number = 3 ;
else abort_run("Too many rows in alignment") ;

write_section_chars_length -= (cols_for_number * 2) ;
```

```
if (write_section_chars_length < 10)
        abort_run("write_section_chars_length is too short") ;

// Check whether what remains of the alignment will fit across
// the breadth of the page. If so, set the value of
// last_section_column to correspond to the last character column
// of the alignment. If not, find the last blank column
// in the current section of write_al_chars[] and set the
// value of last_section_column to point to the column before
// that blank column.

while (true)
{
        last_section_column = first_section_column +
                write_section_chars_length - 1 ;
        if (last_section_column > al_char_length)
                last_section_column = al_char_length - 1 ;
        else
        {
                for (c_col1 = first_section_column +
                        write_section_chars_length - 1;
                c_col1 >= first_section_column; c_col1--)
                {
                        // See whether the column in
                        // write_al_chars[] corresponding
                        // to c_col1 is blank. If so, break.

                        for (c_row1 = 0; c_row1 < al_char_depth;
                        c_row1++)
                        {
                                if (write_al_chars[c_row1][c_col1]
                                        != ' ')
                                        break ;
                        }

                        if (c_row1 >= al_char_depth) break ;
                }

                if (c_col1 <= first_section_column)
                        abort_run("Anomaly in last_section_column") ;

                last_section_column = c_col1 - 1 ;
        }

        // Write out the current section. Write the row number
        // at the beginning and end of each row.

        bool number_required = false ;

        s_row1 = 0 ;

        for (c_row1 = 0; c_row1 < al_char_depth; c_row1++)
        {
                if (number_required == false) number_required = true ;
                else number_required = false ;

                if (number_required)
                {
                        if (cols_for_number == 2)
                                fprintf(output_file, "%1d", s_row1) ;
                        else fprintf(output_file, "%2d", s_row1) ;
                        fprintf(output_file, " ") ;
                }
                else
                {
                        if (cols_for_number == 2)
                                fprintf(output_file, "  ") ;
                        else fprintf(output_file, "   ") ;
                }

                for (c_col1 = first_section_column; c_col1 <=
                        last_section_column; c_col1++)
                {
                        fprintf(output_file, "%c",
                                write_al_chars[c_row1][c_col1]) ;
                }

                if (number_required)
                {
                        fprintf(output_file, " ") ;
                        if (cols_for_number == 2)
                                fprintf(output_file, "%d%s", s_row1,
                                "\n") ;
```

```
                                else fprintf(output_file, "%2d%s", s_row1,
                                        "\n") ;

                                s_row1++ ;
                        }
                        else fprintf(output_file, "\n") ;
                }

                first_section_column = last_section_column + 2 ;
                if (first_section_column >= al_char_length)
                {
                        fprintf(output_file, "\n") ;
                        break ;
                }

                if (sections_per_page != NULL_VALUE)
                {
                        section_counter++ ;
                        if (section_counter >= sections_per_page)
                        {
                                fprintf(output_file, "\\newpage\n") ;
                                section_counter = 0 ;
                        }
                        else fprintf(output_file, "\n\n") ;
                }
                else fprintf(output_file, "\n\n") ;
        }

        fflush(output_file) ;
} // sequence::write_alignment_horizontal

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Writes out an alignment in 'vertical' format: there is a column
**        for each row in the alignment and symbols are written horizontally.
**
** CALLING SEQUENCE:
**
**        void sequence::write_alignment_vertical(FILE *output_file)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        output_file:        The file into which the alignment is to be written.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void sequence::write_alignment_vertical(FILE *output_file)
{
        struct row_record
        {
                int row_width ;
                int row_position ; // The position of the first character
                        // in the column, numbered from the left. The first
                        // character is in position 0.
                symbol *next_hit_symbol ;
                list_element *this_el_pos, *row_el_pos ;
                sequence *row_pattern ;
                int non_hit_symbols_to_be_written ;
                bool new_values_needed ;
        } ;

        struct row_record *write_vertical_array ;
        char one_line[MEDIUM_SCRATCH_ARRAY_SIZE] ;

        write_vertical_array = new row_record[sequence_depth] ;
```

```
// Find the widths and positions of rows. For any row, the width
// is the length of the longest symbol that will go in the row, plus
// GAP_BETWEEN_AL_COLS.

sequence *row_pattern ;
symbol *al_symbol1, *al_symbol2, *row_symbol1, *row_symbol2,
                *next_hit_symbol ;
int longest_name_length = 0, name_length, index1,
        index2, index3, index4, non_hit_symbols_count, size_of_gap ;

write_vertical_array[0].row_position = 0 ;

int row, line_length = 0, row_width ;
for (row = 0; row < sequence_depth; row++)
{
        longest_name_length = 0 ;
        write_vertical_array[row].row_pattern = row_pattern =
                this->get_row_pattern(row) ;
        for (row_symbol1 = (symbol *)row_pattern->get_first_child();
                row_symbol1 != NIL ; row_symbol1 = (symbol *)row_pattern->
                        get_next_child())
        {
                name_length = strlen(row_symbol1->get_name()) ;
                if (name_length > longest_name_length)
                        longest_name_length = name_length ;
        }

        write_vertical_array[row].row_width = row_width = longest_name_length +
                GAP_BETWEEN_AL_COLS ;
        if (row + 1 < sequence_depth)
        {
                write_vertical_array[row + 1].row_position =
                        write_vertical_array[row].row_position + row_width ;
        }

        line_length += row_width ;

        write_vertical_array[row].new_values_needed = true ;
        write_vertical_array[row].this_el_pos = NIL ;
        write_vertical_array[row].row_el_pos = NIL ;
}

line_length -= GAP_BETWEEN_AL_COLS ; // The last set of blank
        // characters in the line are not needed.
if (line_length + 1 >= MEDIUM_SCRATCH_ARRAY_SIZE)
        abort_run("Line is too long in write_alignment_vertical()") ;

// Now write out the alignment.

char *symbol_name, c ;
int max_non_hit_symbols, name_index ;
list_element *this_el_pos1 = NIL, *this_el_pos2, *row_el_pos ;

list_for_el_pos(al_symbol1, symbol, this, this_el_pos1)
{
        // Where needed, find the next hit symbol in each row and
        // the number of non-hit symbols that precede it. If there
        // is no next hit symbol for a given row, count the number
        // of trailing non-hit symbols there are and add that
        // number to write_vertical_array[row].non_hit_symbols_to_be_written.

        for (row = 0; row < sequence_depth; row++)
        {
                if (row > 0) break ; // EDIT: line added.
                if (write_vertical_array[row].new_values_needed == false)
                        continue ;

                next_hit_symbol = NIL ;
                this_el_pos2 = write_vertical_array[row].this_el_pos ;

                list_for_el_pos(al_symbol2, symbol, this, this_el_pos2)
                {
                        if (al_symbol2->is_a_hit() == false) continue ;
                        next_hit_symbol = al_symbol2->get_row_symbol(row) ;
                        if (next_hit_symbol != NIL) break ;
                }

                // We have found the next hit symbol in the row or
                // got to the end of 'this' without finding another
                // hit symbol.

                write_vertical_array[row].next_hit_symbol = next_hit_symbol ;
                non_hit_symbols_count = 0 ;
```

```
        row_el_pos = write_vertical_array[row].row_el_pos ;
        row_pattern = write_vertical_array[row].row_pattern ;
        while (row_symbol1 = (symbol *)row_pattern->
                get_next_child_by_el_pos(&row_el_pos))
        {
                if (row_symbol1 == next_hit_symbol) break ;
                non_hit_symbols_count++ ;
        }

        write_vertical_array[row].non_hit_symbols_to_be_written =
                non_hit_symbols_count ;

        write_vertical_array[row].new_values_needed = false ;
        write_vertical_array[row].next_hit_symbol = next_hit_symbol ;
}

// EDIT: next line commented out.
// if (al_symbol1->is_a_hit() == false) continue ;

// Find out the maximum number of non-hit symbols that must
// be fitted in before the current hit.

max_non_hit_symbols = 0 ;
for (row = 0; row < sequence_depth; row++)
{
        if (row > 0) break ; // EDIT: line added.
        non_hit_symbols_count = 0 ;
        row_symbol1 = al_symbol1->get_row_symbol(row) ;
        if (row_symbol1 == NIL) continue ;

        // We have a hit symbol in the current row and column.

        row_pattern = write_vertical_array[row].row_pattern ;

        row_el_pos = write_vertical_array[row].row_el_pos ;
        while (row_symbol2 = (symbol *)row_pattern->
                get_next_child_by_el_pos(&row_el_pos))
        {
                if (row_symbol2 == row_symbol1) break ;
                non_hit_symbols_count++ ;
        }
        write_vertical_array[row].non_hit_symbols_to_be_written =
                non_hit_symbols_count ;
        if (non_hit_symbols_count > max_non_hit_symbols)
                max_non_hit_symbols = non_hit_symbols_count ;
}

// Now write out lines until the available space for non-hit
// symbols has been used up or there are no more non-hit symbols.

while (max_non_hit_symbols > 0)
{
        // Clear one_line[], the array that will hold the
        // characters for one line.

        for (index1 = 0; index1 < line_length; index1++)
                one_line[index1] = ' ' ;
        one_line[line_length] = '\0' ;

        // For each row, write out non-hit symbols, provided
        // max_non_hit_symbols has not been reached or available
        // non-hit symbols have been exhausted.

        for (row = 0; row < sequence_depth; row++)
        {
                if (row > 0) break ; // EDIT: line added.
                row_el_pos = write_vertical_array[row].row_el_pos ;
                row_pattern = write_vertical_array[row].row_pattern ;
                row_symbol1 = (symbol *)row_pattern->
                        get_next_child_by_el_pos(&row_el_pos) ;
                if (row_symbol1 == NIL) continue ;
                next_hit_symbol = write_vertical_array[row].next_hit_symbol ;
                if (write_vertical_array[row].non_hit_symbols_to_be_written <= 0)
                        continue ;

                // Put characters from row_symbol1 into line1.

                index1 = write_vertical_array[row].row_position ;
                name_index = 0 ;
                symbol_name = row_symbol1->get_name() ;
                c = symbol_name[name_index++] ;
                while (c != '\0')
                {
```

```
                        one_line[index1++] = c ;
                        c = symbol_name[name_index++] ;
                }

                (write_vertical_array[row].non_hit_symbols_to_be_written)-- ;
                write_vertical_array[row].row_el_pos = row_el_pos ;
        }

        max_non_hit_symbols-- ; // Subtract 1 from the
                // maximum number of non-hit symbols to be written.
        fprintf(output_file, "%s%c", one_line, '\n') ;
}

// Now write out the hit symbols in the current column.

// Clear one_line[], the array that will hold the
// characters for one line.

for (index1 = 0; index1 < line_length; index1++)
        one_line[index1] = ' ' ;
one_line[line_length] = '\0' ;

for (row = 0; row < sequence_depth; row++)
{
        row_symbol1 = al_symbol1->get_row_symbol(row) ;
        if (row_symbol1 == NIL) continue ;

        // Put characters from row_symbol1 into line1.

        index1 = write_vertical_array[row].row_position ;
        name_index = 0 ;
        symbol_name = row_symbol1->get_name() ;
        c = symbol_name[name_index++] ;
        while (c != '\0')
        {
                one_line[index1++] = c ;
                c = symbol_name[name_index++] ;
        }

        write_vertical_array[row].row_el_pos = row_el_pos ;
        row_pattern = write_vertical_array[row].row_pattern ;
        write_vertical_array[row].this_el_pos = this_el_pos1 ;
        write_vertical_array[row].new_values_needed = true ;
        write_vertical_array[row].row_el_pos =
                row_pattern->get_el_pos_by_child(row_symbol1) ;
}

// Put in dashes to link hit symbols in the current line.
// First scan across to the first non-blank character.

for (index2 = 0; index2 < line_length; index2++)
        if (one_line[index2] != ' ') break ;

if (index2 >= line_length)
        abort_run("Blank line in write_alignment_vertical()") ;

for (index1 = index2; index1 < line_length; index1++)
{
        if (one_line[index1] != ' ') continue ;

        // We have a blank character in one_line[].

        index2 = index1 ;

        // Scan ahead to the next non-blank character, if any.

        size_of_gap = 1 ;
        index2++ ;

        while (index2 < line_length)
        {
                if (one_line[index2] != ' ') break ;
                index2++ ;
                size_of_gap++ ;
        }

        if (index2 >= line_length) break ;

        if (size_of_gap == 1) one_line[index1] = '.' ;
        else if (size_of_gap == 2)
        {
                one_line[index1] = '.' ;
                one_line[index1 + 1] = '.' ;
```

```
                }
                else
                {
                        index3 = index1 + 1 ;
                        index4 = index2 - 1 ;

                        while (index3 < index4)
                        {
                                one_line[index3] = '-' ;
                                index3++ ;
                        }
                }
                index1 = index2 ;
        }

        fprintf(output_file, "%s%c", one_line, '\n') ;
}

// Do trailing non-hit symbols (if any). First find out the
// maximum number of trailing non-hit symbols in any row.

max_non_hit_symbols = 0 ;
for (row = 0; row < sequence_depth; row++)
{
        if (row > 0) break ; // EDIT: line added.
        row_el_pos = write_vertical_array[row].row_el_pos ;
        row_pattern = write_vertical_array[row].row_pattern ;
        non_hit_symbols_count = 0 ;
        while (row_symbol1 = (symbol *)row_pattern->
                get_next_child_by_el_pos(&row_el_pos))
                        non_hit_symbols_count++ ;
        write_vertical_array[row].non_hit_symbols_to_be_written =
                non_hit_symbols_count ;
        if (non_hit_symbols_count > max_non_hit_symbols)
                max_non_hit_symbols = non_hit_symbols_count ;
}

while (max_non_hit_symbols > 0)
{
        // Clear one_line[], the array that will hold the
        // characters for one line.

        for (index1 = 0; index1 < line_length; index1++)
                one_line[index1] = ' ' ;
        one_line[line_length] = '\0' ;

        // For each row, write out non-hit symbols if there are
        // any left.

        for (row = 0; row < sequence_depth; row++)
        {
                if (write_vertical_array[row].non_hit_symbols_to_be_written <= 0)
                        continue ;
                row_el_pos = write_vertical_array[row].row_el_pos ;
                row_pattern = write_vertical_array[row].row_pattern ;
                row_symbol1 = (symbol *)row_pattern->
                        get_next_child_by_el_pos(&row_el_pos) ;

                // Put characters from row_symbol1 into line1.

                index1 = write_vertical_array[row].row_position ;
                name_index = 0 ;
                symbol_name = row_symbol1->get_name() ;
                c = symbol_name[name_index++] ;
                while (c != '\0')
                {
                        one_line[index1++] = c ;
                        c = symbol_name[name_index++] ;
                }

                (write_vertical_array[row].non_hit_symbols_to_be_written)-- ;
                write_vertical_array[row].row_el_pos = row_el_pos ;
        }
        fprintf(output_file, "%s%c", one_line, '\n') ;
        max_non_hit_symbols-- ;
}

fprintf(output_file, "\n") ;

delete[] write_vertical_array ;

} // sequence::write_alignment_vertical
```

```
/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Writes out the symbols in a pattern with details of
**        the type, status and other information about symbols.
**
** CALLING SEQUENCE:
**
**        void sequence::write_with_details(bool write_from)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        write_from:                true if '(from alignment ID)' is to be written,
**                                        otherwise false.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        Output of the pattern with details of symbols
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void sequence::write_with_details(bool write_from)
{
        symbol *symbol1 ;
        int counter = 0, symbol_frequency, tp ;
        char type[SMALL_SCRATCH_ARRAY_SIZE],
                status[SMALL_SCRATCH_ARRAY_SIZE] ;

        write_pattern(true, write_from) ;

        if (sequence_depth > 1) // The following details are to be written
                // out only if 'this' pattern is an alignment.
        {
                fprintf(output_file,
                        "%s%1.2f%s%1.2f%s%1.2f%s%1.2f%s",
                        "NSC = ",
                        new_symbols_cost,
                        ", EC = ",
                        encoding_cost,
                        ", CR = ",
                        compression_ratio,
                        ", CD = ",
                        compression_difference,
                        "\n\n") ;
        }

        list_for(symbol1, symbol, this)
        {
                counter++ ;
                if (counter >= 3)
                {
                        fprintf(output_file, "\n") ;
                        counter = 1 ;
                }

                tp = symbol1->get_type() ;

                if (tp == CONTEXT_SYMBOL) strcpy(type, "CS") ;
                else if (tp == LEFT_BRACKET) strcpy(type, "LB") ;
                else if (tp == RIGHT_BRACKET) strcpy(type, "RB") ;
                else if (tp == UNIQUE_ID_SYMBOL) strcpy(type, "UID") ;
                else strcpy(type, "DATA") ;

                if (symbol1->get_status() == IDENTIFICATION)
                        strcpy(status, "ID") ;
                else if (symbol1->get_status() == BOUNDARY_MARKER)
                        strcpy(status, "BM") ;
                else strcpy(status, "CN") ;

                symbol_frequency = symbol1->get_frequency() ;
```

```
                fprintf(output_file, "%s%s%d%s%1.2f%s%s%s%s",
                        symbol1->get_name(),
                        " (fr = ", symbol_frequency, ", ",
                        symbol1->get_bit_cost(),
                        ", ", type,
                        ", ", status) ;

                if (this->this_is_last_child(symbol1))
                        fprintf(output_file, "%s%d%s",
                                "), frequency = ",
                                frequency,
                                ".\n\n") ;
                else fprintf(output_file, "), ") ;
        }
} // sequence::write_with_details

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Tests whether the CONTENTS symbols in one sequence of symbols
**        match the CONTENTS symbols in another sequence.
**
** CALLING SEQUENCE:
**
**        bool sequence::contents_symbols_match(sequence *pattern1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        If the CONTENTS symbols match, true, otherwise false.
**
**        pattern1:                The sequence to be matched with 'this'.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

bool sequence::contents_symbols_match(sequence *pattern1)
{
        symbol *symbol1, *symbol2 ;

        symbol2 = (symbol *)this->get_first_child() ;
        for (symbol1 = (symbol *)pattern1->get_first_child();
        symbol1 != NIL; symbol1 = (symbol *)pattern1->get_next_child())
        {
                if (symbol1->get_status() != CONTENTS) continue ;
                if (symbol2 == NIL) return(false) ;

                while (symbol2->get_status() != CONTENTS)
                {
                        symbol2 = (symbol *)this->get_next_child() ;
                        if (symbol2 == NIL) return(false) ;
                }

                if (symbol1->name_matches(symbol2))
                {
                        symbol2 = (symbol *)this->get_next_child() ;
                        continue ;
                }
                else return(false) ;
        }

        // pattern1 has been completely stepped through. Check to see
        // whether or not there are any CONTENTS symbols left in 'this'.

        if (symbol2 != NIL)
        {
                if (symbol2->get_status() == CONTENTS) return(false) ;
                while (symbol2 = (symbol *)this->get_next_child())
                        if (symbol2->get_status() == CONTENTS) return(false) ;
        }
```

```
        return(true) ;
} // sequence::contents_symbols_match

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks 'this' pattern against pre-existing patterns
**        (in old_patterns) to see whether there is one
**        that has all and only the same CONTENTS symbols in the
**        same order. If there is, that sequence is returned.
**        Otherwise, the method returns NIL.
**
** CALLING SEQUENCE:
**
**        sequence *sequence::check_patterns()
**
** FORMAL ARGUMENTS:
**
**        Return value:        A pre-existing pattern with all and only the
**                             same CONTENTS symbols, or NIL if no such pattern
**                             exists.
**
** IMPLICIT INPUTS:
**
**        patterns in old_patterns.
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

sequence *sequence::check_patterns()
{
        sequence *pattern1 ;
        list_element *el_pos1 = NIL ;
        bool pattern_found = false ;

        for (pattern1 = (sequence *)old_patterns->
                get_next_child_by_el_pos(&el_pos1);
                pattern1 != NIL ; pattern1 = (sequence *)old_patterns->
                        get_next_child_by_el_pos(&el_pos1))
        {
                if (this->contents_symbols_match(pattern1))
                {
                        pattern_found = true ;
                        break ;
                }
        }

        if (pattern_found)
        {
                fprintf(output_file, "Contents-symbol match found between ") ;
                this->write_pattern(false, true) ;
                fprintf(output_file, " and ") ;
                pattern1->write_pattern(true, true) ;
                return(pattern1) ;
        }

        return(NIL) ;
} // sequence::check_patterns

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Calculates the length of an alignment.
**
** CALLING SEQUENCE:
**
**        int sequence::compute_character_length()
**
** FORMAL ARGUMENTS:
**
**        Return value:        The length of the alignment in characters, including
**                             The white spaces between symbols.
```

```
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

int sequence::compute_character_length()
{
        int char_length = 0 ;
        symbol *symbol1 ;
        list_element *el_pos1 = NIL ;
        char *symbol_name ;

        while (symbol1 = (symbol *)this->get_next_child_by_el_pos(&el_pos1))
        {
                if (use_abbreviations == LETTERS || use_abbreviations == DIGITS)
                        symbol_name = find_symbol_abbreviation(symbol1->
                                get_name()) ;
                else symbol_name = symbol1->get_name() ;

                char_length += strlen(symbol_name) + 1 ; // Add 1
                // to the length of the symbol name for the
                // blank space between one column and the next.
        }

        return(char_length - 1) ; // Subtract 1 from char_length because
                                  // the blank character after the last symbol is not needed.
} // sequence::compute_character_length

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Writes a textual description of the sequence specified to
**        the 'write file'. This is called from write_tree_object()
**        and wto().
**
** CALLING SEQUENCE:
**
**        void sequence::wto(int print_code) ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        None.
**
**        print_code:               PRINT_ALL_FREQUENCIES if the frequencies
**                                       of the object and all its constituents
**                                       are to be printed,
**        print_code:               PRINT_ALL_FREQUENCIES if the frequencies
**                                       of the object and all its constituents
**                                       are to be printed,
**                                       PRINT_SEQUENCE_FREQUENCY if symbol
**                                       frequencies are to be suppressed.
**                                       PRINT_SEQUENCE_FREQUENCY otherwise.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        output_file
**
** SIDE EFFECTS:
**
**        NONE
*/

void sequence::wto(int print_code)
{
        base_object *child;
        char buffer[STRING_LENGTH] ;
```

```
        write_lines(output_file, "(", CONTINUE) ;

        this->initialise() ;
        while (child = this->get_next_child())
        {
                child->wto(print_code) ;
                if (current_el_pos->get_next() != NIL)
                        write_lines(output_file, " ", CONTINUE) ;
        }
        write_lines(output_file, ")", CONTINUE) ;
        // if (print_code == PRINT_SEQUENCE_FREQUENCY && frequency != 1)
        if (print_code == PRINT_SEQUENCE_FREQUENCY)
        {
                sprintf(buffer, "*%d", frequency) ;
                write_lines(output_file, buffer, CONTINUE) ;
                return ;
        }
        // if (print_code == PRINT_ALL_FREQUENCIES && frequency != 1)
        if (print_code == PRINT_ALL_FREQUENCIES)
        {
                sprintf(buffer, "*%d", frequency) ;
                write_lines(output_file, buffer, CONTINUE) ;
        }
} /* sequence::wto */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Determines whether or not 'this' ID-symbol has any function within
**        old_patterns, either as a reference from another pattern, or
**        as a discrimination symbol that distinguishes a member of a
**        context class or as a top-level identifier for a pattern.
**
** CALLING SEQUENCE:
**
**        bool symbol::has_no_function()
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if 'this' symbol has no function, false otherwise.
**
** IMPLICIT INPUTS:
**
**        old_patterns
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

bool symbol::has_no_function()
{
        // Look to see whether 'this' symbol is referenced from within
        // the CONTENTS symbols of any Old pattern.

        symbol *symbol1 ;
        sequence *pattern1 ;
        list_element *el_pos1, *el_pos2 ;
        int symbol_type, status, counter ;
        bool this_symbol_or_copy_found ;

        // Look to see whether 'this' symbol is the only ID-symbol
        // for any one pattern. If so, it is needed as the top-level
        // identifier for the pattern and should not be removed.

        // At the same time, look to see whether 'this' symbol has
        // been referenced from any pattern. If it has, it is needed
        // should not be removed.

        el_pos1 = NIL ;
        list_for_el_pos(pattern1, sequence, old_patterns, el_pos1)
        {
                el_pos2 = NIL ;
                counter = 0 ;
                this_symbol_or_copy_found = false ;
                list_for_el_pos(symbol1, symbol, pattern1, el_pos2)
                {
```

```
                        status = symbol1->get_status() ;
                        if (status == IDENTIFICATION)
                        {
                                if (symbol1 == this || name_matches(symbol1))
                                        this_symbol_or_copy_found = true ;
                                counter++ ;
                        }

                        if (status != CONTENTS) continue ;
                        symbol_type = symbol1->get_type() ;
                        if (symbol_type == LEFT_BRACKET
                                || symbol_type == RIGHT_BRACKET)
                                        continue ;

                        // symbol1 is a 'reference' symbol.

                        if (name_matches(symbol1))
                                return(false) ; // 'this' symbol has been referenced
                                        // so this->has_no_function() is false.
                }

                if (this_symbol_or_copy_found
                                && counter == 1) // 'this' symbol, or a copy of it,
                                        // is the only ID-symbol for pattern1
                                        // so this->has_no_function() is false.
                        return(false) ;
        }

        return(true) ;

} // symbol::has_no_function

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        This is like add_child but checks that the child to be added
**        does not duplicate any child already added (meaning the identical
**        object, not merely one with a matching name). If the proposed
**        addition is the same as a child already added, the proposed addition
**        is not added.
**
** CALLING SEQUENCE:
**
**        bool group::add_pattern_without_duplicates(sequence *seq1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if child added, false otherwise.
**
**        child:                        The new base_object to be added.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE. Notice, in particular, that the position of current_el_pos
**        is not changed by adding a new child to an object.
*/

bool group::add_pattern_without_duplicates(sequence *seq1)
{
        list_element *temp1, *temp2, *next ;
        base_object *existing_sequence ;

        if (seq1 == NIL)
                abort_run("NIL seq1 in \
                        group::add_pattern_without_duplicates()") ;
        temp2 = first_element ;
        if (temp2 == NIL)
        {
                temp1 = new list_element(seq1) ;
                first_element = temp1 ;
                this->mark_parent_and_int_positions_non_recursive() ; // take
                // care! This alters the value of the 'parent' field.
                return(true) ;
```

```
        }
        else
        {
                existing_sequence = temp2->get_el_obj() ;
                if (existing_sequence == seq1) return(false) ;
        }

        while (next = temp2->get_next())
        {
                existing_sequence = next->get_el_obj() ;
                if (existing_sequence == seq1) return(false) ;
                temp2 = next ;
        }

        temp1 = new list_element(seq1) ;
        temp2->set_next(temp1) ;
        this->mark_parent_and_int_positions_non_recursive() ; // take
                // care! This alters the value of the 'parent' field.
        return(true) ;
} /* group::add_pattern_without_duplicates */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        This is like add_child but checks that the child to be added
**        is not a copy of any child already added. If the proposed
**        addition is a copy of a child already added, the proposed addition
**        is not added. If the proposed addition is identical to a child already
**        added, the proposed addition is not added.
**
** CALLING SEQUENCE:
**
**        bool group::add_pattern_without_copies_or_duplicates(sequence *seq1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if child is added, false otherwise.
**
**        child:                        The new base_object to be added.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE. Notice, in particular, that the position of current_el_pos
**        is not changed by adding a new child to an object.
*/

bool group::add_pattern_without_copies_or_duplicates(sequence *seq1)
{
        list_element *temp1, *temp2, *next ;
        sequence *existing_sequence ;

        if (seq1 == NIL)
                abort_run("NIL seq1 in \
                        group::add_pattern_without_copies_or_duplicates().") ;
        temp2 = first_element ;
        if (temp2 == NIL)
        {
                temp1 = new list_element(seq1) ;
                first_element = temp1 ;
                this->mark_parent_and_int_positions_non_recursive() ; // take
                // care! This alters the value of the 'parent' field.
                return(true) ;
        }
        else
        {
                existing_sequence = (sequence *)temp2->get_el_obj() ;
                if (existing_sequence == seq1) return(false) ;
                if (existing_sequence->is_copy_of(seq1)) return(false) ; // Notice that this
                        // test must be done after the previous one because is_copy_of()
                        // flags an error if the two things being compared are the
                        // same entity.
        }
```

```
        while (next = temp2->get_next())
        {
                existing_sequence = (sequence *)next->get_el_obj() ;
                if (existing_sequence == seq1) return(false) ;
                if (existing_sequence->is_copy_of(seq1)) return(false) ; // Notice that this
                        // test must be done after the previous one because is_copy_of()
                        // flags an error if the two things being compared are the
                        // same entity.
                temp2 = next ;
        }

        temp1 = new list_element(seq1) ;
        temp2->set_next(temp1) ;
        this->mark_parent_and_int_positions_non_recursive() ; // take
                // care! This alters the value of the 'parent' field.
        return(true) ;
} /* group::add_pattern_without_copies_or_duplicates */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        This is like add_child but checks that the child to be added
**        is not a copy of any child already added. If the proposed
**        addition is a copy of a child already added, the proposed addition
**        is not added. If the proposed addition is identical to a child already
**        added, the proposed addition is not added.
**
** CALLING SEQUENCE:
**
**        bool group::add_symbol_without_copies_or_duplicates(symbol *symbol1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if symbol1 is added, false otherwise.
**
**        symbol1:                The new symbol to be added.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE. Notice, in particular, that the position of current_el_pos
**        is not changed by adding a new child to an object.
*/

bool group::add_symbol_without_copies_or_duplicates(symbol *symbol1)
{
        list_element *temp1, *temp2, *next ;
        symbol *existing_symbol ;

        if (symbol1 == NIL)
                abort_run("NIL symbol1 in \
                        group::add_symbol_without_copies_or_duplicates().") ;
        temp2 = first_element ;
        if (temp2 == NIL)
        {
                temp1 = new list_element(symbol1) ;
                first_element = temp1 ;
                this->mark_parent_and_int_positions_non_recursive() ; // take
                // care! This alters the value of the 'parent' field.
                return(true) ;
        }
        else
        {
                existing_symbol = (symbol *)temp2->get_el_obj() ;
                if (existing_symbol == symbol1) return(false) ;
                if (existing_symbol->name_matches(symbol1)) return(false) ;
        }

        while (next = temp2->get_next())
        {
                existing_symbol = (symbol *)next->get_el_obj() ;
                if (existing_symbol == symbol1) return(false) ;
                if (existing_symbol->name_matches(symbol1)) return(false) ;
                temp2 = next ;
```

```
        }

        temp1 = new list_element(symbol1) ;
        temp2->set_next(temp1) ;
        this->mark_parent_and_int_positions_non_recursive() ; // take
                // care! This alters the value of the 'parent' field.
        return(true) ;
} /* group::add_symbol_without_copies_or_duplicates */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks to see whether the group contains any duplicated references
**        to any object. It is assumed that the objects are sequences.
**
** CALLING SEQUENCE:
**
**        bool group::contains_duplicates()
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if the group contains at least one duplicate,
**                                    false otherwise.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

bool group::contains_duplicates()
{
        list_element *el_pos1 = NIL, *el_pos2 ;
        sequence *child1, *child2 ;

        list_for_el_pos(child1, sequence, this, el_pos1)
        {
                el_pos2 = el_pos1 ;
                list_for_el_pos(child2, sequence, this, el_pos2)
                {
                        if (child1 == child2)
                        {
                                fprintf(output_file, "%s%d%s%d%s",
                                        "In group ID",
                                        ID,
                                        " sequence ID",
                                        child1->get_ID(),
                                        " is duplicated\n\n") ;
                                return(true) ;
                        }
                }
        }

        return(false) ;
} // group::contains_duplicates

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Inserts an sequence into group_of_patterns in so that
**        sequences are in order of their CD.
**
** CALLING SEQUENCE:
**
**        void group::insert_sequence_in_order_of_CD(sequence *al1)
**
** FORMAL ARGUMENTS:
**
**        Return value:                void
**
**        al1:                                The sequence to be inserted.
**
```

```
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        'this' with addition of al1.
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void group::insert_sequence_in_order_of_CD(sequence *al1)
{
        sequence *existing_sequence ;
        double al_CD = al1->get_compression_difference() ;

        list_for(existing_sequence, sequence, this)
        {
                if (al_CD < existing_sequence->get_compression_difference())
                        continue ;

                this->precede(al1, existing_sequence) ;

                return ;
        }

        /* If the program gets this far, the al1 has a larger
        CR than any of the existing sequences so it should be added
        at the end of the set. */

        this->add_child(al1) ;

} /* group::insert_sequence_in_order_of_CD */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Writes out IDs of members. If a sequence is 'new_this_cycle', it
**        is marked with an asterisk ('*').
**
** CALLING SEQUENCE:
**
**        void group::write_IDs(int selection, sequence *cnp)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        selection:                Used to choose a subset of the group. NULL_VALUE
**                                        means that all alignments are selected. Other valid
**                                        values are FULL_A, FULL_B, FULL_C and PARTIAL.
**        cnp:                        The New pattern being processed
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void group::write_IDs(int selection, sequence *cnp)
{
        int counter1 = 0, counter2 = 0 ;
        sequence *pattern1 ;

        // Count the number of IDs to be printed.

        if (selection == NULL_VALUE)
                counter2 = count_number_of_children() ;
        else
        {
                list_for(pattern1, sequence, this)
```

```
                {
                        if (selection == pattern1->get_degree_of_matching())
                                counter2++ ;
                }
        }

        // Now print the IDs.

        int counter3 = 0 ;
        list_for(pattern1, sequence, this)
        {
                if (selection != pattern1->get_degree_of_matching()) continue ;
                counter3++ ;
                fprintf(output_file, "%s%d",
                        "ID",
                        pattern1->get_ID()) ;

                if (pattern1->is_new_this_cycle())
                        fprintf(output_file, "*") ;

                if (counter3 == counter2)
                {
                        fprintf(output_file, "\n\n") ;
                        break ;
                }

                if (++counter1 == 10)
                {
                        fprintf(output_file, ",\n") ;
                        counter1 = 0 ;
                }
                else fprintf(output_file, ", ") ;
        }
} // group::write_IDs

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks whether two groups contain the same objects, not
**        necessarily in the same order.
**
** CALLING SEQUENCE:
**
**        bool group::group_matches(group gr1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if 'this' contains the same objects as gr1,
**                                    otherwise false.
**
**        gr1:                        The group to be compared with 'this'.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

bool group::group_matches(group *gr1)
{
        list_element *pos1 = NIL, *pos2 ;
        base_object *obj1, *obj2 ;
        bool match_found ;

        if (this->count_number_of_children() !=
                gr1->count_number_of_children())
                        return(false) ;

        list_for_el_pos(obj1, base_object, this, pos1)
        {
                pos2 = NIL ;
                match_found = false ;
                list_for_el_pos(obj2, base_object, gr1, pos2)
                        if (obj1 == obj2)
```

```
                        {
                                match_found = true ;
                                break ;
                        }
                if (match_found == false) return(false) ;
        }
        return(true) ;
} // bool group::group_matches

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Writes out the patterns in a set of patterns, together with
**        an introductory heading.
**
** CALLING SEQUENCE:
**
**        void group::write_patterns(char *heading, int origin, int selection)
**
** FORMAL ARGUMENTS:
**
**        Return value:                void
**
**        heading:                        The heading for the set. If the value is NIL,
**                                               no heading is printed.
**        origin:                                Only patterns with the specified origin
**                                                are to be printed. If origin is NULL_VALUE,
**                                                all the patterns in the set_of_patterns
**                                                are to be written.
**        selection:                       If NULL_VALUE, all patterns are written, otherwise
**                                                Other valid values are FULL_A, FULL_B,
**                                                FULL_C or PARTIAL.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        Output to output_file.
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void group::write_patterns(char *heading, int origin, int selection)
{
        sequence *pattern1 ;
        list_element *el_pos1 = NIL ;

        if (heading != NIL) fprintf(output_file, "%s%s", heading, "\n\n") ;

        list_for_el_pos(pattern1, sequence, this, el_pos1)
        {
                if (origin != NULL_VALUE && origin != pattern1->get_origin())
                        continue ;
                if (selection != NULL_VALUE)
                {
                        if (selection != pattern1->get_degree_of_matching())
                                continue ;
                }

                pattern1->print_ID() ;
                fprintf(output_file, ": ") ;
                pattern1->write_tree_object(PRINT_SEQUENCE_FREQUENCY) ;
                fflush(output_file) ;
        }
} // group::write_patterns

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Write New or Old. Called from write_new_and_old.
**
** CALLING SEQUENCE:
**
**        void group::write_patterns_with_details()
```

```
**
** FORMAL ARGUMENTS:
**
**        Return value:               void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void group::write_patterns_with_details()
{
        sequence *pattern1 ;

        this->initialise() ;
        while (pattern1 = (sequence *)this->get_next_child())
        {
                pattern1->print_ID() ;

                if (this == old_patterns)
                        fprintf(output_file, "%s%1.2f%s",
                        " Coding cost: ",
                        pattern1->get_encoding_cost(),
                        ", ") ;
                else if (this == new_patterns)
                        fprintf(output_file, " No coding cost ") ;
                pattern1->write_tree_object(PRINT_ALL_FREQUENCIES) ;
                pattern1->write_with_details(true) ;
        }
} // group::write_patterns_with_details

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Sorts a group of patterns in descending order of their
**        compression_difference scores.
**
** CALLING SEQUENCE:
**
**        void group::sort_by_compression_difference()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void group::sort_by_compression_difference()
{
        group *temp_group = new group ;
        sequence *pattern1, *best_pattern ;
        double highest_compression_difference,
                pattern_compression_difference ;

        while (pattern1 = (sequence *)this->extract_first_child())
                temp_group->add_child(pattern1) ;

        while(temp_group->is_empty() == false)
        {
                highest_compression_difference = LOW_VALUE ;
```

```
                list_for(pattern1, sequence, temp_group)
                {
                        pattern_compression_difference =
                                pattern1->get_compression_difference() ;
                        if (pattern_compression_difference >
                                highest_compression_difference)
                        {
                                highest_compression_difference =
                                        pattern_compression_difference ;
                                best_pattern = pattern1 ;
                        }
                }
                temp_group->extract_child(best_pattern) ;
                this->add_child(best_pattern) ;
        }

        delete temp_group ;
} // group::sort_by_compression_difference

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Sorts a group of patterns in ascending order of ID number.
**
** CALLING SEQUENCE:
**
**        void group::sort_by_ID()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        The sorted group
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void group::sort_by_ID()
{
        group *temp_group = new group ;
        sequence *pattern1, *best_pattern ;
        int lowest_ID, pattern_ID ;

        while (pattern1 = (sequence *)this->extract_first_child())
                temp_group->add_child(pattern1) ;

        while(temp_group->is_empty() == false)
        {
                lowest_ID = HIGH_VALUE ;
                list_for(pattern1, sequence, temp_group)
                {
                        pattern_ID = pattern1->get_ID() ;
                        if (pattern_ID < lowest_ID)
                        {
                                lowest_ID = pattern_ID ;
                                best_pattern = pattern1 ;
                        }
                }
                temp_group->extract_child(best_pattern) ;
                this->add_child(best_pattern) ;
        }

        delete temp_group ;
} // group::sort_by_ID

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Compiles a list of the different symbol types in the patterns
**        in the group 'this' and measures the size of this alphabet.
```

```
**
** CALLING SEQUENCE:
**
**        group *group::compile_alphabet(int *alphabet_size)
**
** FORMAL ARGUMENTS:
**
**        Return value:        A group containing the list of symbols.
**
**        alphabet_size:        A pointer to a variable to hold the size of
**                                    the alphabet.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

group *group::compile_alphabet(int *alphabet_size)
{
        group *symbol_set = this->symbol_set() ;
        symbol_set->sort_by_name() ; // Puts symbols
                // in symbol_set in alpha-numeric order of their
                // name fields.
        *alphabet_size = symbol_set->count_number_of_children() ;
        return(symbol_set) ;
} // compile_alphabet

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Scans a list of patterns in 'this' and, for each symbol in each
**        pattern, adds the frequency of the pattern to the frequency of
**        the symbol type, stored in symbol_set.
**
** CALLING SEQUENCE:
**
**        void group::compile_frequencies(group *symbol_set,
**                sequence *last_pattern)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        symbol_set:               The alphabet of symbol types in the group.
**        last_pattern:        The last pattern in the group to be processed. If the
**                                    value is NIL, all patterns are to be processed.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void group::compile_frequencies(group *symbol_set,
        sequence *last_pattern)
{
        sequence *pattern1 ;
        symbol *symbol1, *symbol_type ;

        list_for(pattern1, sequence, this)
        {
                list_for(symbol1, symbol, pattern1)
                {
                        /* Scan through symbol_set to find the
```

```
                        symbol there (representing a symbol type)
                        which matches the current symbol. When it
                        is found, update the frequency. */

                        symbol_set->initialise() ;
                        while (symbol_type = (symbol *)
                                symbol_set->get_next_child())
                        {
                                if (strcmp(symbol1->get_name(),
                                        symbol_type->get_name()) == 0)
                                {
                                        symbol_type->
                                                add_to_frequency(pattern1->get_frequency()) ;
                                        break ;
                                }
                        }
                        if (symbol_type == NIL)
                                abort_run("symbol_type not found in \
                                        group::compile_frequencies") ;
                }
                if (pattern1 == last_pattern) break ;
        }
} // group::compile_frequencies

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Writes a textual description of the group specified to
**        the 'write file'. This is called from write_tree_object()
**        and wto().
**
** CALLING SEQUENCE:
**
**        void group::wto(int print_code) ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        None.
**
**        print_code:                PRINT_ALL_FREQUENCIES if the frequencies
**                                        of the object and all its constituents
**                                        are to be printed,
**                                        PRINT_SEQUENCE_FREQUENCY if symbol
**                                        frequencies are to be suppressed.
**                                        PRINT_SEQUENCE_FREQUENCY otherwise.
**
** IMPLICIT INPUTS:
**
** IMPLICIT OUTPUTS:
**
**        output_file
**
** SIDE EFFECTS:
**
**        NONE
*/

void group::wto(int print_code)
{
        base_object *child ;
        char buffer[STRING_LENGTH] ;

        write_lines(output_file, "[", CONTINUE) ;

        this->initialise() ;
        while (child = this->get_next_child())
        {
                child->wto(print_code) ;
                if (current_el_pos != NIL)
                        write_lines(output_file, " ", CONTINUE) ;
        }
        write_lines(output_file, "]", CONTINUE) ;
        if (print_code == PRINT_SEQUENCE_FREQUENCY) return ;
        if (print_code == PRINT_GROUP_FREQUENCIES && frequency != 1)
        {
                sprintf(buffer, "*%d", frequency) ;
                write_lines(output_file, buffer, CONTINUE) ;
                return ;
        }
        if (print_code == PRINT_ALL_FREQUENCIES && frequency != 1)
        {
```

```
                sprintf(buffer, "*%d", frequency) ;
                write_lines(output_file, buffer, CONTINUE) ;
        }
} /* group::wto */

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Writes a textual description of the group specified to
**        the 'write file'. This is called from write_tree_object()
**        and wto().
**
** CALLING SEQUENCE:
**
**        void symbol::wto(int print_code) ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        None.
**
**        print_code:                PRINT_ALL_FREQUENCIES if the frequencies
**                                        of the object and all its constituents
**                                        are to be printed,
**                                        PRINT_SEQUENCE_FREQUENCY if symbol
**                                        frequencies are to be suppressed.
**                                        PRINT_SEQUENCE_FREQUENCY otherwise.
**
** IMPLICIT INPUTS:
**
** IMPLICIT OUTPUTS:
**
**        output_file
**
** SIDE EFFECTS:
**
**        NONE
*/

void symbol::wto(int print_code)
{
        char buffer[STRING_LENGTH] ;

        write_lines(output_file, name, CONTINUE) ;

        if (print_code == PRINT_SEQUENCE_FREQUENCY) return ;

        if (print_code == PRINT_ALL_FREQUENCIES && frequency != 1)
        {
                sprintf(buffer, "*%d", frequency) ;
                write_lines(output_file, buffer, CONTINUE) ;
        }
} /* symbol::wto */

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Writes a base_object to a file. Using wto(),
**        it keeps track of the numbers of characters in a line and
**        takes a new line whenever that is needed to avoid splitting
**        a symbol or a frequency value.
**
** CALLING SEQUENCE:
**
**        void tree_object::write_tree_object(int print_code)
**
** FORMAL ARGUMENTS:
**
**        Return value:        NONE
**
**        print_code:                PRINT_ALL_FREQUENCIES if the frequencies
**                                        of the object and all its constituents
**                                        are to be printed,
**                                        PRINT_SEQUENCE_FREQUENCY if SYMBOL frequencies are
**                                        to be suppressed.
**                                        PRINT_SEQUENCE_FREQUENCY otherwise.
**
** IMPLICIT INPUTS:
**
**        NONE
```

```
**
** IMPLICIT OUTPUTS:
**
**        Text in the write file.
**
** SIDE EFFECTS:
**
**        NONE
*/

void tree_object::write_tree_object(int print_code)
{
        write_lines(output_file, "", START) ;
        this->wto(print_code) ;
        fprintf(output_file, "\n\n") ;
        fflush(output_file) ;
} /* tree_object::write_tree_object */

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks whether a set contains a symbol which
**        matches a parameter symbol.
**
** CALLING SEQUENCE:
**
**        symbol *tree_object::contains_copy_of(symbol *symbol1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        The first item in the set which matches symbol1 or
**                                    NIL if no matching item is found.
**
**        symbol1:                The data node to be matched to items in the set.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

symbol *tree_object::contains_copy_of(symbol *symbol1)
{
        symbol *symbol_in_set ;

        /* Run over set until symbol1 is found or end of set reached */

        this->initialise() ;
        while (symbol_in_set = (symbol *)this->get_next_child())
                if (symbol1->name_matches(symbol_in_set))
                        return(symbol_in_set) ;

        return(NIL) ;
} /* tree_object::contains_copy_of */

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Returns the number of child nodes in a composite data node and
**        set that value in number_of_children.
**
** CALLING SEQUENCE:
**
**        int tree_object::count_number_of_children() ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        The number of child nodes in the given object.
**
** IMPLICIT INPUTS:
**
```

```
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

int tree_object::count_number_of_children()
{
        register int no_of_children = 0 ;
        base_object *child ;
        list_element *pos1 = NIL ;

        list_for_el_pos(child, base_object, this, pos1)
                no_of_children++ ;

        number_of_children = no_of_children ;
        return(no_of_children) ;
} /* tree_object::count_count_number_of_children */

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        In an symbol, finds an al_el by its position.
**
** CALLING SEQUENCE:
**
**        list_element *tree_objectect::get_el_pos_by_int_pos(int position)
**
** FORMAL ARGUMENTS:
**
**        Return value:        The el corresponding to position.
**
**        position:                The position of the el to be found.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

list_element *tree_object::get_el_pos_by_int_pos(int position)
{
        list_element *el1 ;

        if (position < -1)
                abort_run("Invalid position in get_el_pos_by_int_pos") ;

        if (position == -1) return(NIL) ; // The position -1 corresponds
        // to the position just before the first actual position.

        for (el1 = this->get_first_element() ; el1 != NIL;
                el1 = el1->get_next())
                        if (el1->get_position() == position) break ;

        if (el1 == NIL) abort_run("el1 not found in \
                        get_el_pos_by_int_pos") ;

        return(el1) ;
} // tree_object::get_el_pos_by_int_pos

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        In an symbol, finds an al_el by its symbol.
**
** CALLING SEQUENCE:
```

```
**
**        list_element *tree_object::get_el_pos_by_child(base_object *obj1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        The el corresponding to position.
**
**        obj1:                        The object in the el to be found.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

list_element *tree_object::get_el_pos_by_child(base_object *obj1)
{
        list_element *el1 ;

        for (el1 = this->get_first_element() ; el1 != NIL;
                el1 = el1->get_next())
                        if (el1->get_el_obj() == obj1) break ;

        if (el1 == NIL) abort_run("el1 not found in \
                        get_el_pos_by_child") ;

        return(el1) ;
} // tree_object::get_el_pos_by_child

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        A method to create a set of (copies of) the symbols in an
**        object, without any duplicates.
**
** CALLING SEQUENCE:
**
**        group *tree_object::symbol_set()
**
** FORMAL ARGUMENTS:
**
**        Return value:        A group containing copies of symbols in an object,
**                                        without duplicates.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

group *tree_object::symbol_set()
{
        group *set ;

        set = new group ;
        this->symbol_set_recursion(set) ;
        return(set) ;
} // tree_object::symbol_set

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Calculates values for EC and total_cost for any pattern. EC is
```

```
**        the sum of the bit_cost values of the symbols in the pattern that are
**        marked as IDENTIFICATION symbols. total_cost is the sum of the
**        bit_costs of all the symbols in the pattern.
**
**        The computed value is stored in the pattern.
**
** CALLING SEQUENCE:
**
**        void sequence::compute_costs()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        The computed value is stored in the pattern.
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void sequence::compute_costs()
{
        symbol *symbol1 ;
        double cumulative_EC = 0, cumulative_cost = 0, symbol_bit_cost ;

        list_for(symbol1, symbol, this)
        {
                symbol_bit_cost = symbol1->get_bit_cost() ;
                cumulative_cost += symbol_bit_cost ;
                if (symbol1->get_status() != IDENTIFICATION) continue ;
                cumulative_EC += symbol_bit_cost ;
        }

        this->set_encoding_cost(cumulative_EC) ;
        this->set_total_cost(cumulative_cost) ;
} // sequence::compute_costs

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Makes a code from an alignment ('this'), assuming that patterns
**        in the alignment have code symbols. If they do not have code
**        symbols, then this method does not make a code.
**
**        The method is simply to traverse the alignment, making copies of
**        non-bracket IDENTIFICATION symbols that appear in a column by
**        themselves. The copies exclude the row information in the originals.
**
**        At the same time as making the code, this method computes values
**        for the 'actual cost' of the New hit symbols in the alignment
**        (NSC), the 'minimal cost' of the code symbols in the code (EC),
**        and the 'compression difference' (CD), the 'compression ratio'
**        (CR) and the abs_P of the alignment.
**
**        The code and the calculated values are not printed out - so that
**        the arrangements and formats of these values can be tailored
**        to specific contexts.
**
**        In calculating values for NSC, EC, CR and CD, no attempt is
**        made to correct for gaps in matching.
**
** CALLING SEQUENCE:
**
**        sequence *sequence::make_code(bool return_code)
**
** FORMAL ARGUMENTS:
**
**        Return value:        The new code.
**
**        return_code:        If true, the function returns an explicit code sequence.
**                                    Otherwise it returns NIL.
**
** IMPLICIT INPUTS:
```

```
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NSC, EC, CD, CR and abs_P for the 'this'.
**
** SIDE EFFECTS:
**
**        NONE
**
*/

sequence *sequence::make_code(bool return_code)
{
        symbol *symbol1, *symbol2 ;
        int i ;
        double NSC, EC ;

        struct row_record
        {
                sequence *row_pattern ; // The pattern in this row.
                int previous_new_int_pos ; // Records the position in New
                        // of the previous hit with row_pattern. If there has
                        // not yet been a hit with New, this variable has a
                        // NULL_VALUE.
                int previous_old_int_pos ; // Records the position in
                        // row_pattern of the previous hit between another pattern
                        // and row_pattern. If there has not yet been a hit on
                        // row_pattern, this variable has a NULL_VALUE.
                bool row_pattern_fully_recognised ; // true if all the
                        // CONTENTS symbols in this row have hits with New and if
                        // the corresponding hit symbols in New form a coherent
                        // sequence.
        } ;

        struct row_record *rows_array ;

        rows_array = new row_record[sequence_depth] ;

        // Initialise variables in rows_array.

        for (i = 0; i < sequence_depth; i++)
        {
                rows_array[i].row_pattern = NIL ;
                rows_array[i].previous_new_int_pos = NULL_VALUE ;
                rows_array[i].previous_old_int_pos = NULL_VALUE ;
                rows_array[i].row_pattern_fully_recognised = true ;
        }

        NSC = EC = 0 ;
        symbol *symbol_new ;
        list_element *el_pos1 = NIL ;

        sequence *new_code ;
        if (return_code) new_code = new sequence(CODE_PATTERN) ;

        list_for_el_pos(symbol1, symbol, this, el_pos1)
        {
                // Identify columns in the alignment that contain hit
                // symbols from cnp. Add up the
                // 'actual costs' of these symbols to arrive at the
                // new_symbols_cost for the alignment.

                if (symbol1->is_a_hit())
                {
                        symbol_new = symbol1->get_row_symbol(0) ;
                        if (symbol_new != NIL)
                        {
                                NSC += symbol_new->get_bit_cost() ;
                        }
                        continue ; // We are now only interested in non-hit
                                // CONTEXT_SYMBOLs with status IDENTIFICATION.
                }

                if (symbol1->get_status() != IDENTIFICATION) continue ;

                // Make a 'shallow' copy of symbol1 (excluding information about
                // rows ) and add it to the code. At the same time, add the
                // bit_cost of symbol1 to EC.

                if (return_code)
                {
```

```
                        symbol2 = symbol1->shallow_copy() ;
                        symbol2->set_status(CONTENTS) ;
                        new_code->add_child(symbol2) ;
                }

                EC += symbol1->get_bit_cost() ;
        }

        this->set_new_symbols_cost(NSC) ;
        this->set_encoding_cost(EC) ;
        this->set_compression_difference(NSC - EC) ;
        this->set_compression_ratio(NSC / EC) ;

        double two_power_s = pow(2, EC) ;
        this->set_abs_P(1 / two_power_s) ;

        delete[] rows_array ;

        if (return_code)
        {
                new_code->set_new_symbols_cost(NSC) ;
                new_code->set_encoding_cost(EC) ;
                new_code->set_compression_difference(NSC - EC) ;
                new_code->set_compression_ratio(NSC / EC) ;
                new_code->set_derived_from_parsing(ID) ;
                return(new_code) ;
        }
        else return(NIL) ;

} // sequence::make_code

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Recursive part of symbol_set.  Recursively adds all the different
**        symbols in 'object' to the object set 'set'.  If a SYMBOL is already
**        in 'set' then it's frequency is updated every time it is encountered
**        in 'object' rather than adding it again.
**
**        In the initial call the 'symbol_set_recursion', 'set' should be an empty
**        group.
**
** CALLING SEQUENCE:
**
**        void symbol::symbol_set_recursion(group *set)
**
** FORMAL ARGUMENTS:
**
**        Return value:                        None.
**
**        set:                                        The object to receive the set which is
**                                                        built up.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        The symbols in 'set' become altered by this function.
**
** SIDE EFFECTS:
**
**        NONE
*/

void symbol::symbol_set_recursion(group *set)
{
        symbol *item_in_set, *new_set_item ;

        // Check whether set contains a symbol that matches 'this'.

        item_in_set = (symbol *)set->contains_copy_of(this) ;
        if (item_in_set == NIL)
        {
                // Item is not in the set - add a copy of it.

                new_set_item = new symbol(*this) ;
                // new_set_item->set_frequency(1) ;
                set->add_child(new_set_item) ;
        }
```

```
        else
        {
                // The symbol is already in the set:
                // update it's frequency in the set.

                // item_in_set->increment_frequency() ;
        }
} /* symbol::symbol_set_recursion */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Checks a hit sequence to see whether the CONTENTS symbols in the
**        target pattern are all matched. Here, 'this' hit
**        node is the last node in the sequence.
**
** CALLING SEQUENCE:
**
**        bool hit_node::all_contents_symbols_are_matched()
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if all CONTENTS symbols are matched,
**                                   false otherwise.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

bool hit_node::all_contents_symbols_are_matched()
{
        hit_node *node1 = this ;
        symbol *target_symbol ;
        sequence *driving_pattern = node1->get_driving_pattern(),
                *target_pattern = node1->get_target_pattern() ;
        hit_node *hit_node_array[MEDIUM_SCRATCH_ARRAY_SIZE] ;
        int counter = 0, fe_hit_node_array, i ;

        // Recast hit sequence for easy left-to-right access. First,
        // find the length of the hit sequence.

        while (node1 != hit_root)
        {
                counter++ ;
                node1 = node1->get_hn_parent() ;
        }

        if (counter >= MEDIUM_SCRATCH_ARRAY_SIZE)
                abort_run("Array overflow in \
                        hit_node::all_data_symbols_are_matched()") ;

        fe_hit_node_array = counter ;

        // Now put the nodes into hit_node_array[].

        node1 = this ;
        while (node1 != hit_root)
        {
                hit_node_array[--counter] = node1 ;
                node1 = node1->get_hn_parent() ;
        }

        // Work left to right, checking symbols in the hits recorded
        // in hit_node_array[] and symbols between the hits in
        // the target_pattern.

        target_pattern->initialise() ;

        symbol *symbol1 ;
        int dr_int_pos2 = NULL_VALUE ;

        for (i = 0; i < fe_hit_node_array; i++)
```

```
        {
                node1 = hit_node_array[i] ;
                target_symbol = node1->get_target_symbol() ;

                // Check the symbols in the target_pattern up to but not
                // including the target_symbol.

                while (symbol1 = (symbol *)target_pattern->get_next_child())
                {
                        if (symbol1 == target_symbol) break ;
                        if (symbol1->get_status() == CONTENTS)
                                return(false) ; // We have a CONTENTS symbol that is
                                        // not part of a hit on this cycle.
                }
        }

        // Now do the trailing parts of the target_pattern, if any.

        while (symbol1 = (symbol *)target_pattern->get_next_child())
        {
                if (symbol1->get_status() == CONTENTS)
                        return(false) ; // We have a CONTENTS symbol that is
                                // not part of a hit on this cycle.
        }

        return(true) ;

} // hit_node::all_contents_symbols_are_matched

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Consructor for the class hit_node.
**
** CALLING SEQUENCE:
**
**        hit_node::hit_node() : tree_object()
**
** FORMAL ARGUMENTS:
**
**        Return value:        None
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

hit_node::hit_node() : cost_tree_object()
{
        ID = hit_node_ID_number++ ;
        hn_parent = NIL ;
        driving_pattern = NIL ;
        driving_symbol = NIL ;
        driving_int_pos = NULL_VALUE ;
        target_pattern = NIL ;
        target_symbol = NIL ;
        target_int_pos = NULL_VALUE ;
        leaf_entry = NIL ;
} // hit_node::hit_node

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Copy constructor for the class hit_node.
**
** CALLING SEQUENCE:
**
**        hit_node::hit_node(hit_node &hn) : tree_object(hn)
**
** FORMAL ARGUMENTS:
```

```
**
**        Return value:        None
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

hit_node::hit_node(hit_node &hn) : cost_tree_object(hn)
{
        ID = hit_node_ID_number++ ;
        hn_parent = hn.get_hn_parent() ;
        driving_pattern = hn.get_driving_pattern() ;
        driving_symbol = hn.get_driving_symbol() ;
        driving_int_pos = hn.get_driving_int_pos() ;
        target_pattern = hn.get_target_pattern() ;
        target_symbol = hn.get_target_symbol() ;
        target_int_pos = hn.get_target_int_pos() ;
        leaf_entry = hn.get_leaf_entry() ;
} // hit_node::hit_node

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Compares two alignments to see whether they match or not.
**
** CALLING SEQUENCE:
**
**        bool sequence::alignment_matches(sequence *al1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if the two alignments match, false otherwise.
**
**        al1:                        The second of the two alignments to be compared.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

bool sequence::alignment_matches(sequence *al1)
{
        symbol *al_col1, *al_col2, *symbol1, *symbol2 ;
        alignment_element *al_el1, *al_el2 ;
        sequence *orig_patt1, *orig_patt2 ;
        int al1_length, al1_depth, row1, row2 ;
        bool match_found, al1_rows[MEDIUM_SCRATCH_ARRAY_SIZE] ;

        if (al1 == NIL) abort_run("NIL value for al1 in \
                sequence::alignment_matches") ;

        /* Compare the lengths and depths of the sequences. */

        al1_length = al1->get_number_of_children() ;

        if (number_of_children != al1_length) return(false) ;

        al1_depth = al1->get_sequence_depth() ;

        if (sequence_depth != al1_depth) return(false) ;

        // Check for a match in the basic sequence of columns.
```

```
        symbol2 = (symbol *)al1->get_first_child() ;
        for (symbol1 = (symbol *)this->get_first_child();symbol1 != NIL ;
                symbol1 = (symbol *)this->get_next_child())
        {
                if (symbol1->name_matches(symbol2) == false) return(false) ;
                symbol2 = (symbol *)al1->get_next_child() ;
                if (symbol2 == NIL)
                {
                        symbol1 = (symbol *)this->get_next_child() ;
                        break ;
                }
        }

        if (symbol1 != NIL || symbol2 != NIL)
                return(false) ;

        /* Now check if actual sequences match. Two sequences can
        be the same even though the rows are in a different order.
        However, if they are to be recognised as the same, for every
        row in one sequence, there must be a row in the other sequence
        which matches it exactly. */

        /* Initialise al1_rows[] which is used to avoid unnecessary
        re-checking of rows in al1 which have already been found to
        match a row in *this. */

        for (row2 = 0; row2 < al1_depth; row2++) al1_rows[row2] = false ;

        for (row1 = 0; row1 < sequence_depth; row1++)
        {
                match_found = false ;
                for (row2 = 0; row2 < al1_depth; row2++)
                {
                        if (al1_rows[row2] == true) continue ;

                        al_col1 = (symbol *)this->get_first_child() ;
                        this->set_current_el_pos(this->get_first_el_pos()) ;
                        al_el1 = al_col1->get_al_el(row1) ;
                        orig_patt1 = al_el1->get_original_pattern() ;

                        al_col2 = (symbol *)al1->get_first_child() ;
                        al1->set_current_el_pos(al1->get_first_el_pos()) ;
                        al_el2 = al_col2->get_al_el(row2) ;
                        orig_patt2 = al_el2->get_original_pattern() ;

                        if (orig_patt2 != orig_patt1) continue ;

                        symbol1 = (symbol *)al_el1->get_el_obj() ;
                        symbol2 = (symbol *)al_el2->get_el_obj() ;
                        if (symbol1 != symbol2) continue ;

                        while (al_col1 = (symbol *)this->get_next_child())
                        {
                                symbol1 = al_col1->get_row_symbol(row1) ;
                                al_col2 = (symbol *)al1->get_next_child() ;
                                symbol2 = al_col2->get_row_symbol(row2) ;
                                if (symbol1 != symbol2) goto L1 ;
                        }
                        match_found = true ;
                        al1_rows[row2] = true ;
                        break ;
                        L1: ;
                }
                if (!match_found) return(false) ;
        }

        return(true) ;
} /* sequence::alignment_matches */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        For diagnostic purposes.
**
** CALLING SEQUENCE:
**
**        void sequence::show_al_structure(hit_node *h_node)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
```

```
**
**        h_node:                       The leaf node for the given alignment. The
**                                       value may be NIL, in which case, information
**                                       about driving pattern, target_pattern and
**                                       h_node itself are not printed out.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        Printed output
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void sequence::show_al_structure(hit_node *h_node)
{
        symbol *al_col1 ;
        int i ;

        fprintf(output_file, "Alignment ") ;
        this->print_ID() ;
        fprintf(output_file, " : ") ;

        if (h_node != NIL)
        {
                (h_node->get_driving_pattern())->print_ID() ;
                fprintf(output_file, " : ") ;
                (h_node->get_target_pattern())->print_ID() ;
                fprintf(output_file, " : ") ;
                h_node->print_ID() ;
        }
        else fprintf(output_file, "NIL hit_node") ;

        fprintf(output_file, "%s%1.2f%s%1.2f%s%1.2f%s%d%s%d%s",
                ", NSC = ", new_symbols_cost,
                ", EC = ", encoding_cost,
                ", CR = ", compression_ratio,
                ",\nsequence_depth = ", sequence_depth,
                ", number of columns = ", number_of_children,
                "\n\n") ;
        this->initialise();
        i = 0 ;
        while (al_col1 = (symbol *)this->get_next_child())
        {
                fprintf(output_file, "%s%d%s", "Column ", i, ":\n") ;
                al_col1->show_al_structure() ;
                i++ ;
        }
        fflush(output_file) ;

        fprintf(output_file, "Alignment ") ;
        this->print_ID() ;
        fprintf(output_file, "\n\n") ;
        this->write_alignment(output_file, write_section_chars_length,
                NULL_VALUE, alignment_format) ;
        fprintf(output_file, "End of alignment ") ;
        this->print_ID() ;
        fprintf(output_file, "\n\n") ;
        fflush(output_file) ;
} // sequence::show_al_structure

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Fetches terminal nodes of the hit structure in sorted order and
**        sets the value of an external pointer to the corresponding sequence.
**
** CALLING SEQUENCE:
**
**        hit_node *get_leaf_nodes_in_order(int *row, sequence **alignm,
**                int *leaf_index)
**
** FORMAL ARGUMENTS:
**
**        Return value:        The next node to be fetched. NIL when there are
```

```
**                                      no more nodes.
**
**        row:                        A pointer to an integer referencing the relevant row
**                                        in sort_array[]. Start *row with 0.
**        alignm:                 A pointer to an external pointer to the sequence
**                                        corresponding to row.
**        leaf_index:                A pointer to an external integer indexing leaf_array[]
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

hit_node *get_leaf_nodes_in_order(int *row, sequence **alignm, int *leaf_index)
{
        if (*row >= fe_sort) return(NIL) ;
        *leaf_index = sort_array[*row] ;
        (*row)++ ;
        *alignm = alignments_array[*leaf_index] ;
        return(leaf_array[*leaf_index]) ;
} // get_leaf_nodes_in_order

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Cleans up memory allocated by new_patterns and
**        then exits. If this memory is not cleaned up it may clog
**        up the machine.
**
** CALLING SEQUENCE:
**
**        void exit_routine(int status)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        status:                 0 for successful termination, 1 otherwise.
**
** IMPLICIT INPUTS:
**
**        corpus, original_symbols_in_corpus, boundary_symbol,
**        duplicate_unifications, hn_master, hit_root
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void exit_routine(int status)
{
        delete created_patterns ;
        parsing_alignments->delete_children() ;
        parsing_alignments->release_children() ;
        delete parsing_alignments ;
        selected_cumulative_parsing_alignments->release_children() ;
        delete selected_cumulative_parsing_alignments ;
        delete cumulative_parsing_alignments ;
        delete set_of_grammars ;
        delete naive_grammar ;
        set_of_combinations->delete_children() ;
        delete set_of_combinations ;
        delete best_combinations ;
        delete generalisation_list1 ;
        delete generalisation_list2 ;

        delete hn_master ;
```

```
        delete brackets_list ;

        // delete corpus ;

        delete symbols_in_new ;
        delete original_symbols_in_corpus ;
        delete symbol_types_in_old ;
        delete new_patterns ;
        delete old_patterns ;

        delete[] write_al_chars ;
        delete[] write_al_vacant_slots ;

        fclose(output_file) ;
        fclose(input_file) ;
        fclose(parameters_file) ;
        fclose(latex_file) ;
        exit(status) ;
} /* exit_routine */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**               Aborts the program after displaying an error message.
**
** CALLING SEQUENCE:
**
**        void abort_run(char *message)
**
** FORMAL ARGUMENTS:
**
**        Return value:        None; this function will never return.
**
**        message:               The string containing the error message.
**
** IMPLICIT INPUTS:
**
**        Functions: write_message
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void abort_run(char *message)
{
        write_message(message) ;
        exit_routine(1) ;
} /* abort_run */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Writes a string to the logfile.
**
** CALLING SEQUENCE:
**
**        void write_message(char *s)
**
** FORMAL ARGUMENTS:
**
**        Return value:        None
**
**        s:                                The string to be written.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        logfile: Written to.
**
** SIDE EFFECTS:
```

```
**
**        Logfile is updated (written to).  No checks are made
**        as to whether an error occurred during writing.
*/

void write_message(char *s)
{
        fprintf(output_file, "%s%s", s, "\n\n") ;
} /* write_message */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Writes to a given output stream, counting the number of characters
**        in each line and taking a new line when necessary.
**
** CALLING SEQUENCE:
**
**        void void write_lines(FILE *stream, char *string, int status)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        stream:                        The file to be written to.
**        string:                  The string to be written.
**        status:                  START for starting, CONTINUE for carrying on.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void write_lines(FILE *stream, char *string, int status)
{
        static column_counter = 0 ;

        if (status == START) column_counter = 0 ;

        column_counter += strlen(string) ;
        if (column_counter > MAX_WRITE_LINE)
        {
                fprintf(stream, "\n") ;
                column_counter = 0 ;
        }

        fprintf(stream, "%s", string) ;
} /* write_lines */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Scans over spaces and commas in the ‘syntax buffer’.
**
** CALLING SEQUENCE:
**
**        void scan(void) ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        None.
**
** IMPLICIT INPUTS:
**
**        Variables:        syn_buf_ptr
**
** IMPLICIT OUTPUTS:
**
**        syn_buf_ptr
**
** SIDE EFFECTS:
```

```
**
**        None, but see IMPLICIT OUTPUTS above.
*/

void scan() /* Scans over spaces, commas etc between units in the input */
{
        char c;

        c = *syn_buf_ptr ;
        while (c == ' ' || c == ',')
                c = *(++syn_buf_ptr) ;
} /* scan */

/*****************************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Reads a frequency value for an object (if any) recorded as
**        '*<integer_string>' immediately after the object in the
**        input file. If there is a value, this overwrites the default
**        value (1) in internal representation of the object.
**
** CALLING SEQUENCE:
**
**        void read_frequency(base_object *obj) ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        obj:                          The object whose frequency value is to be read.
**
** IMPLICIT INPUTS:
**
**        syn_buf_ptr
**
** IMPLICIT OUTPUTS:
**
**        syn_buf_ptr
**
** SIDE EFFECTS:
**
**        None, but see IMPLICIT OUTPUTS above.
*/

void read_frequency(base_object *obj)
{
        char *cpx, digit_array[STRING_LENGTH] ;
        int i ;

        cpx = syn_buf_ptr ;
        if (*syn_buf_ptr == '*') syn_buf_ptr++ ;
        else return ;
        if (!isdigit(*syn_buf_ptr))
        {
                syn_buf_ptr = cpx ;
                return ;
        }
        i = 0 ;
        while (isdigit(*syn_buf_ptr))
        {
                digit_array[i] = *syn_buf_ptr ;
                if (++i >= STRING_LENGTH)
                        abort_run("Frequency digits in input too long") ;
                syn_buf_ptr++ ;
        }
        digit_array[i] = '\0' ;

        obj->set_frequency(atoi(digit_array)) ;

        scan() ;
} /* read_frequency */

/*****************************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Make a symbol out of a string of characters in the 'syntax buffer'.
**        true is returned if a valid symbol was constructed, false otherwise.
**        A valid symbol includes "". If the
**        symbol is too long, an error is raised and the program aborts.
```

```
**
** CALLING SEQUENCE:
**
**        bool rec_symbol(char *nme, bool *symbol_is_identifier) ;
**
** FORMAL ARGUMENTS:
**
**        Return value:                        true if 'nme' is valid, false otherwise.
**
**        nme:                                        String to receive the symbol.
**                                                        NB: Call by reference - symbol value is
**                                                        passed back through this parameter.
**        symbol_is_identifier:        Pointer to an external variable which is set in
**                                                        the function to true if the given
**                                                        symbol is IDENTIFICATION, false otherwise.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        syn_buf_ptr
**
** SIDE EFFECTS:
**
**        The string 'nme' must be declared such that it is of sufficient size
**        to accept the symbol created.  Also note that 'syn_buf_ptr' is
**        updated.
*/

bool rec_symbol(char *nme, bool *symbol_is_identifier) /* Makes a
        symbol out of a string of characters */
{
        char c;
        int i = 0;

        c = *syn_buf_ptr;
        if (c == '\"') /* A quoted string in the input */
        {
                c = *(++syn_buf_ptr) ;
                while (c != '\"')
                {
                        if (i >= (LONG_STRING_LENGTH - 2))
                                abort_run("Symbol too long in 'symbol'") ;
                        *(nme + i++) = c;
                        c = *(++syn_buf_ptr) ;
                }
                syn_buf_ptr++;
        }
        else
        {
                if (c == '!')
                {
                        identification_symbols_marked = true ;
                        *symbol_is_identifier = true ;
                        c = *(++syn_buf_ptr) ;
                }

                while (c != ' ' && c != ',' && c != '('
                        && c != ')' && c != '[' && c != ']')
                {
                        if (i >= (LONG_STRING_LENGTH - 2))
                                abort_run("Symbol too long in 'symbol'") ;
                        nme[i++] = c ;
                        c = *(++syn_buf_ptr) ;
                }
        }
        if (i == 0) return false ;
        nme[i] = '\0' ;
        scan() ;
        return(true) ;
} /* rec_symbol */

/*****************************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Returns true if the string given matches the
**        'syntax buffer'.  The current place in the syntax buffer is advanced
**        past the matched string if the string matches and is left unchanged
**        if it doesn't.
```

```
**
** CALLING SEQUENCE:
**
**        bool characters(char *charsx)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if a match occurred, false otherwise.
**
**        charsx:                 String to match 'syntax buffer' with.
**
** IMPLICIT INPUTS:
**
**        Functions:        scan
**
**        Variables:        syn_buf_ptr
**
** IMPLICIT OUTPUTS:
**
**        syn_buf_ptr
**
** SIDE EFFECTS:
**
**        None, but see IMPLICIT OUTPUTS above.
*/

bool characters(char *charsx) /* Recognises a string of characters
in the syntax buffer */
{
        char *cpx;

        cpx = syn_buf_ptr;
        while (*charsx != '\0')
                if (*syn_buf_ptr++ != *charsx++)
                {
                        syn_buf_ptr = cpx;
                        return(false) ;
                }
        scan() ;
        return(true) ;
} /* characters */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Returns false if the next character in the 'syntax buffer' isn't an SP
**        opening bracket, otherwise the object type associated with the bracket
**        is returned.
**
** CALLING SEQUENCE:
**
**        int left_bracket(void) ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        false if not an SP opening bracket, corresponding object
**                                    type (GROUP, SEQUENCE) otherwise.
**
** IMPLICIT INPUTS:
**
**        Functions: characters
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

int left_bracket()
{
        int b1 = false;

        if (characters("(")) b1 = SEQUENCE ;
        else if (characters("[")) b1 = GROUP ;
        return(b1) ;
} /* left_bracket */

/*****************************************************************/
```

```
/*
** FUNCTIONAL DESCRIPTION:
**
**        Returns true if the character in the ‘syntax buffer’ is the closing
**        bracket of the object type specified, false otherwise.
**
** CALLING SEQUENCE:
**
**        bool right_bracket(int b1)
**
** FORMAL ARGUMENTS:
**
**        Return value:        true if corresponding closing bracket, false otherwise.
**
**        b1:                  Object type.
**
** IMPLICIT INPUTS:
**
**        Functions: characters
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

bool right_bracket(int b1)
{
        if (b1 == SEQUENCE && characters(")")) return(true) ;
        else if (b1 == GROUP && characters("]")) return(true) ;
        return(false) ;
} /* right_bracket */

/*********************************************************************/

tree_object *comp_object() ;

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Recognises an ‘object’.
**
** CALLING SEQUENCE:
**
**        base_object *find_node() ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        A new object if object recognised, NIL otherwise.
**
** IMPLICIT INPUTS:
**
**        Functions:        comp_object, symbol, scan
**
**        Variables:        syn_buf_ptr
**
** IMPLICIT OUTPUTS:
**
**        syn_buf_ptr
**
** SIDE EFFECTS:
**
**        NONE?
*/

/* object

  syntax -> object | bad_syntax ;
  object -> comp_object | simple_object ;
  comp_object -> SEQUENCE | GROUP ;
  SEQUENCE -> ’(’ , body , ’)’ ;
  GROUP -> ’[’ , body , ’]’ ;
  body -> b | NIL ;
  b -> object , body ;
  simple_object -> SYMBOL ;
  SYMBOL -> alphanumeric(+) ;
```

```
*/

base_object *find_node()
{
        char nme[LONG_STRING_LENGTH] ;
        base_object *node ;
        symbol *symbol1 ;
        bool symbol_is_identifier = false ;

        if (node = comp_object())
        {
                read_frequency(node) ;
                return(node) ;
        }
        else // Object is a symbol
        if (rec_symbol(nme, &symbol_is_identifier))
        {
                symbol1 = new symbol(nme, 1, NULL_VALUE) ;
                if (symbol_is_identifier) symbol1->set_status(IDENTIFICATION) ;
                else symbol1->set_status(CONTENTS) ;

                read_frequency(symbol1) ;
                return(symbol1) ;
    }
        else return(NIL) ;

} /* find_node */

/*********************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Recognises a tree_object.
**
** CALLING SEQUENCE:
**
**        tree_object *comp_object() ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        A new node if composite object, NIL otherwise.
**
** IMPLICIT INPUTS:
**
**        syntax_buffer
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

tree_object *comp_object()
{
        int b1 ;
        tree_object *new_node ;
        base_object *child ;

        if (b1 = left_bracket())
        {
                if (b1 == SEQUENCE) new_node = new sequence(BASIC_PATTERN) ;
                else new_node = new group ;
                while (child = find_node())
                        new_node->add_child(child) ;
                if (!right_bracket(b1))
                {
                        delete new_node ;
                        return(NIL) ;
                }
                new_node->mark_parent_and_int_positions_non_recursive();
                // take care! This alters the value of
                // the 'parent' field.
                new_node->set_number_of_children(new_node->
                        count_number_of_children()) ;
                return(new_node) ;
        }
        else return(NIL) ;
} /* comp_object */
```

```
/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Input the SP corpus.
**
** CALLING SEQUENCE:
**
**        tree_object *top_object() ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        node if input is correct, false otherwise.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        syn_buf_ptr
**
** SIDE EFFECTS:
**
**        None, except see IMPLICIT OUTPUTS above.
*/

tree_object *top_object()
{
        tree_object *node ;

        syn_buf_ptr = syntax_buffer;
        scan() ;
        if (node = (tree_object *)find_node()) return(node) ;
        else return(NIL) ;
} /* top_object */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Error non-terminal for SP parser.  Writes 'informative' error message
**        and returns.
**
** CALLING SEQUENCE:
**
**        void bad_syntax(void) ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        None.
**
** IMPLICIT INPUTS:
**
**        Functions:        write_message
**
**        Variables:        syn_buf_ptr
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
*/

void bad_syntax()
{
        char *cpx;
        cpx = syn_buf_ptr - 20;

        if (cpx < syntax_buffer) cpx = syntax_buffer;

        write_message("Error_in_syntax:") ;
        while (cpx <= syn_buf_ptr) write_message(cpx++) ;
        write_message("\n") ;
} /* bad_syntax */

/*****************************************************************/
```

```
/*
** FUNCTIONAL DESCRIPTION:
**
**        Opens file specified using buffer size specified and reads
**        the entire file into the 'syntax buffer'. The file is
**        closed after reading.
**
** CALLING SEQUENCE:
**
**        void in_syntax() ;
**
** FORMAL ARGUMENTS:
**
**        Return value:        None.
**
** IMPLICIT INPUTS:
**
**        Functions:        abort_run
**
**        Variables:        syntax_buffer
**
** IMPLICIT OUTPUTS:
**
**        syntax_buffer
**
** SIDE EFFECTS:
**
**        NONE
*/

void in_syntax()
{
        int c;
        int i = 0;

        /* range check required */
        while (c = getc(input_file))
        {
                if (c == EOF || c == '~') break ;
                if (i >= BUFSIZE)
                        abort_run("Array overflow in 'in_syntax'") ;
                if (c == '\n' || c == '\t')
                        syntax_buffer[i++] = ' ';
                else syntax_buffer[i++] = c;
        }
} /* in_syntax */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Writes out the key to the abbreviations of symbols used in
**        the corpus.
**
** CALLING SEQUENCE:
**
**        void write_key()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        key_array[]
**
** IMPLICIT OUTPUTS:
**
**        Output file
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void write_key()
{
        int key_index ;

        fprintf(output_file, "KEY\n\n") ;
        for (key_index = 0; key_index < fe_key_array; key_index++)
```

```
        {
                fprintf(output_file, "%s%c",
                        key_array[key_index].abbreviated_symbol, ' ') ;
                fprintf(output_file, "%s%c",
                        key_array[key_index].complete_symbol, '\n') ;
        }
        fprintf(output_file, "\nEND KEY\n\n") ;
} // write_key

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Finds a unique abbreviation for a symbol. Called from
**        make_letter_abbreviations.
**
** CALLING SEQUENCE:
**
**        void find_abbreviation(int i)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
**        i:                                The last position in key_array[] that needs to be
**                                        checked in checking for the uniqueness of
**                                        an abbreviation.
**
** IMPLICIT INPUTS:
**
**        complete_symbol[]
**
** IMPLICIT OUTPUTS:
**
**        new_abbreviation[]
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void find_abbreviation(int i)
{
        int early_char_pos, late_char_pos, last_char_pos, j, m ;

        early_char_pos = late_char_pos = 0 ;
        while (complete_symbol[late_char_pos + 1] != '\0')
                late_char_pos++ ;
        last_char_pos = late_char_pos ;
        int k = 0; // Index into new_abbreviation[].
        new_abbreviation[k] = complete_symbol[early_char_pos] ;
        if (late_char_pos > early_char_pos)
        {
                new_abbreviation[++k] = complete_symbol[late_char_pos] ;
                new_abbreviation[++k] = '\0' ;
        }
        else new_abbreviation[++k] = '\0' ;

        // Check whether the newly-constructed abbreviation is
        // unique.

        bool abbreviation_is_unique = true ;
        for (j = 0; j < i; j++)
        {
                if (key_array[j].abbreviated_symbol == NIL) continue ;
                if (strcmp(key_array[j].abbreviated_symbol,
                        new_abbreviation) == 0)
                {
                        abbreviation_is_unique = false ;
                        break ;
                }
        }

        // If it is not unique, keep adding characters until it is.

        bool next_char_is_early = true ;
        char c ;
        while (!abbreviation_is_unique)
        {
                // Add the next character.
```

```
                if (next_char_is_early)
                {
                        next_char_is_early = false ;
                        c = complete_symbol[++early_char_pos] ;
                        while (c == 'a' || c == 'e' || c == 'i' ||
                                c == 'o' || c == 'u')
                                c = complete_symbol[++early_char_pos] ;
                        if (early_char_pos >= late_char_pos)
                        {
                                fprintf(output_file, "%s%s",
                                        complete_symbol,
                                        "\n\n") ;
                                abort_run("Non-unique abbreviation") ;
                        }
                }
                else
                {
                        next_char_is_early = true ;
                        c = complete_symbol[--late_char_pos] ;
                        while (c == 'a' || c == 'e' || c == 'i' ||
                                c == 'o' || c == 'u')
                                c = complete_symbol[--late_char_pos] ;
                        if (early_char_pos >= late_char_pos)
                        {
                                fprintf(output_file, "%s%s",
                                        complete_symbol,
                                        "\n\n") ;
                                abort_run("Non-unique abbreviation") ;
                        }
                }

                // Make the new abbreviation. Do the leading
                // characters first.

                k = 0 ; // Index into new_abbreviation.
                new_abbreviation[k++] = complete_symbol[0] ; // The
                // first character can be a vowel.
                for (m = 1; m <= early_char_pos; m++)
                {
                        c = complete_symbol[m] ;
                        while (c == 'a' || c == 'e' || c == 'i' ||
                                c == 'o' || c == 'u')
                                c = complete_symbol[++m] ;
                        new_abbreviation[k++] = c ;
                }

                // Now do the trailing characters.

                new_abbreviation[k++] = complete_symbol[late_char_pos] ; // The
                // last character can be a vowel.
                for (m = late_char_pos+1; m < last_char_pos; m++)
                {
                        c = complete_symbol[m] ;
                        while (c == 'a' || c == 'e' || c == 'i' ||
                                c == 'o' || c == 'u')
                                c = complete_symbol[++m] ;
                        new_abbreviation[k++] = c ;
                }

                // The last character may be a consonant or a vowel.

                new_abbreviation[k++] = complete_symbol[m] ;
                new_abbreviation[k] = '\0' ;

                // Check whether the newly-constructed abbreviation is
                // unique.

                abbreviation_is_unique = true ;
                for (j = 0; j < i; j++)
                {
                        if (key_array[j].abbreviated_symbol == NIL)
                                continue ;
                        if (strcmp(key_array[j].abbreviated_symbol,
                                new_abbreviation) == 0)
                        {
                                abbreviation_is_unique = false ;
                                break ;
                        }
                }
        }
} // find_abbreviation

/*****************************************************************/
```

```
/*
** FUNCTIONAL DESCRIPTION:
**
**        Creates a key to the abbreviations of symbols to be used in
**        writing out alignments.
**
**        Abbreviations are only made for symbols which do not begin with
**        a digit. Apart from the first and last characters (which can be
**        a consonant or a vowel) the abbreviation for a symbol is made
**        from consonants in the symbol, starting with the first, followed
**        by the last, followed by the consonant after the first, followed
**        by the consonant before the last, and so on. Addition of
**        consonants stops when the abbreviation is unique in the set
**        of abbreviations.
**
** CALLING SEQUENCE:
**
**        void make_letter_abbreviations()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        original_symbols_in_corpus
**
** IMPLICIT OUTPUTS:
**
**        Values for key_array[]
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void make_letter_abbreviations()
{
        symbol *symbol1 ;
        int i, j, symbol_length ;

        // Copy the name of the symbols in the corpus into
        // the 'complete_symbol' field of key_array[] and initialise
        // the abbreviation fields at the same time.

        fe_key_array = 0 ;
        original_symbols_in_corpus->initialise() ;
        while (symbol1 = (symbol *)original_symbols_in_corpus->get_next_child())
        {
                symbol_length = strlen(symbol1->get_name()) + 1 ;
                if (symbol_length > MEDIUM_SCRATCH_ARRAY_SIZE)
                abort_run("Symbol is too long in \
                make_letter_abbreviations") ;
                key_array[fe_key_array].complete_symbol =
                        new char[symbol_length] ;
                strcpy(key_array[fe_key_array].complete_symbol,
                        symbol1->get_name()) ;
                key_array[fe_key_array].abbreviated_symbol = NIL ;
                plus_one(&fe_key_array, KEY_ARRAY_SIZE,
                        "Overflow of key_array[] in read_key") ;
        }

        // Now step through key_array[] finding an abbreviation for each
        // entry that does not begin with '#'.

        for (i = 0; i < fe_key_array; i++)
        {
                strcpy(complete_symbol, key_array[i].complete_symbol) ;
                if (complete_symbol[0] == '#') continue ;

                if (isdigit(complete_symbol[0]))
                {
                        key_array[i].abbreviated_symbol =
                                new char[strlen(complete_symbol) + 1] ;
                        strcpy(key_array[i].abbreviated_symbol,
                                complete_symbol) ;
                        continue ;
                }

                find_abbreviation(i) ;
```

```
                // Add the newly-constructed abbreviation to the key_array[].

                key_array[i].abbreviated_symbol =
                        new char[strlen(new_abbreviation) + 1] ;
                strcpy(key_array[i].abbreviated_symbol, new_abbreviation) ;
        }

        // For each symbol in key_array[] that does begin with '#', try
        // to find the corresponding symbol which does not begin with '#'.
        // If one is found, create an abbreviation for the second symbol
        // by appending the abbreviation of the second symbol to '#'.
        // If one is not found, create an abbreviation for it.

        for (i = 0; i < fe_key_array; i++)
        {

                if (*(key_array[i].complete_symbol) != '#') continue ;
                strcpy(complete_symbol, key_array[i].complete_symbol) ;

                // Try to find a complete_symbol in key_array that matches
                // the current symbol after the '#'.

                for (j = 0; j < fe_key_array; j++)
                {
                        if (*(key_array[j].complete_symbol) == '#')
                                continue ;

                        if (strcmp((complete_symbol + 1),
                                key_array[j].complete_symbol) == 0)
                                break ;
                }

                if (j >= fe_key_array)
                {
                        // No complete symbol has been found which matches
                        // key_array[i].complete_symbol after
                        // the initial '#'.

                        find_abbreviation(fe_key_array) ;

                        key_array[i].abbreviated_symbol =
                                new char[strlen(new_abbreviation) + 1] ;
                        strcpy(key_array[i].abbreviated_symbol,
                                new_abbreviation) ;
                }
                else
                {
                        // Make an abbreviation by appending the abbreviation
                        // that has been found to '#'.

                        new_abbreviation[0] = '#' ;
                        strcpy((new_abbreviation + 1),
                                key_array[j].abbreviated_symbol) ;
                        key_array[i].abbreviated_symbol =
                                new char[strlen(new_abbreviation) + 1] ;
                        strcpy(key_array[i].abbreviated_symbol,
                                new_abbreviation) ;
                }
        }
} // make_letter_abbreviations

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Creates digit abbreviations of symbols in the corpus.
**
** CALLING SEQUENCE:
**
**        void make_digit_abbreviations()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
```

```
**
** SIDE EFFECTS:
**
**        NONE
**
*/

#if MAKE_DIGIT_ABBREVIATIONS

void make_digit_abbreviations()
{
        symbol *symbol1 ;
        int i, j, k, m, symbol_length,
                early_char_pos, late_char_pos,
                last_char_pos ;
        char complete_symbol[MEDIUM_SCRATCH_ARRAY_SIZE],
                new_abbreviation[MEDIUM_SCRATCH_ARRAY_SIZE], c ;
        bool abbreviation_is_unique ;
        bool next_char_is_early ;

        // First, copy the name of the symbols in the corpus into
        // the 'complete_symbol' field of key_array[].

        fe_key_array = 0 ;
        original_symbols_in_corpus->initialise() ;
        while (symbol1 = (symbol *)original_symbols_in_corpus->get_next_child())
        {
                symbol_length = strlen(symbol1->get_name()) + 1 ;
                if (symbol_length > MEDIUM_SCRATCH_ARRAY_SIZE)
                abort_run("Symbol is too long in \
                make_letter_abbreviations") ;
                key_array[fe_key_array].complete_symbol =
                        new char[symbol_length] ;
                strcpy(key_array[fe_key_array].complete_symbol,
                        symbol1->get_name()) ;
                plus_one(&fe_key_array, KEY_ARRAY_SIZE,
                        "Overflow of key_array[] in read_key") ;
        }

        // Now step through key_array[] finding an abbreviation for each
        // entry. To do this, symbols are processed to remove all characters
        // which are not alphanumeric and these symbols are replaced in
        // the same relative position in the digit abbreviation.
        // This means that '(symbol' and 'symbol)'
        // will both contain the same digits, eg '(23' and '23)'.

        for (i = 0; i < fe_key_array; i++)
        {
        }
} // make_digit_abbreviations

#endif

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Reads in the key to abbreviations of symbols used in the corpus.
**
** CALLING SEQUENCE:
**
**        void read_key()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        input file
**
** IMPLICIT OUTPUTS:
**
**        Values for the key stored in key_array[]
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void read_key()
```

```
{
        char input_string[LONG_STRING_LENGTH] ;
        int i ;

        fe_key_array = 0 ;
        fscanf(input_file, "%s", input_string) ;
        while (strcmp(input_string, "KEY") != 0)
                fscanf(input_file, "%s", input_string) ;
        fscanf(input_file, "%s", input_string) ;
        while (strcmp(input_string, "END_KEY") != 0)
        {
                key_array[fe_key_array].abbreviated_symbol =
                        new char[strlen(input_string) + 1] ;
                strcpy(key_array[fe_key_array].abbreviated_symbol,
                        input_string) ;

                fscanf(input_file, "%s", input_string) ;
                if (strcmp(input_string, "END_KEY") == 0) break ;

                // Get rid of any comma at end of string.

                for (i = 0; i < LONG_STRING_LENGTH; i++)
                {
                        if (input_string[i] == '\0')
                        {
                                fprintf(output_file, "%s%s",
                                        input_string, "\n\n") ;
                                abort_run("Error in input_string in read_key") ;
                        }
                        if (input_string[i] == ',' ||
                                input_string[i] == '.')
                        {
                                input_string[i] = '\0' ;
                                break ;
                        }
                }
                key_array[fe_key_array].complete_symbol =
                        new char[strlen(input_string) + 1] ;
                strcpy(key_array[fe_key_array].complete_symbol, input_string) ;

                plus_one(&fe_key_array, KEY_ARRAY_SIZE,
                        "Overflow of key_array[] in read_key") ;

                fscanf(input_file, "%s", input_string) ;
        }
} // read_key

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        Reads in parameters for the program.
**
** CALLING SEQUENCE:
**
**        void read_parameters()
**
** FORMAL ARGUMENTS:
**
**        Return value:        void
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void read_parameters()
{
        char input_string[LONG_STRING_LENGTH],
                *parameters[] =
        {
                "/*",
                "PROBABILITIES",
```

```
        "SHOW_AL_STRUCTURE",
        "SHOW_HIT_STRUCTURE",
        "SHOW_UNSELECTED_ALIGNMENTS",
        "VERBOSE",
        "USE_ABBREVIATIONS",
        "SHOW_ALL_PARSING_ALIGNMENTS",
        "ID_C_SYMBOL_CONSTRAINT",
        "ALIGNMENT_FORMAT",
        "HIT_STRUCTURE_ROWS",
        "FAIL_SCORE",
        "COST_FACTOR",
        "KEEP_ROWS",
        "DRIVING_KEEP_ROWS",
        "FULL_ALIGNMENT_LIMIT",
        "GRAMMAR_LIMIT",
        "EXTRACTION_LIMIT",
        "FIGURE_ID",
        "COMBINATION_LIMIT",
        "MAX_OLD_GAP",
        "MAX_DRIVING_GAP",
        "MAX_TARGET_GAP",
        "MAX_UNSUPPORTED_CYCLES",
        "MAX_ALIGNMENTS_IN_ONE_CYCLE",
        "ORIENTATION",
        "WRITE_AL_CHARS_LENGTH",
        "WRITE_SECTION_CHARS_LENGTH",
        "MINIMUM_FONT_HEIGHT",
        "MAXIMUM_FONT_HEIGHT",
        "N_GRAMMARS",
        "CREATE_PATTERNS_DIAGNOSTIC",
        "LAST_ITEM"
} ;
int i, j, parameter_index ;

fscanf(parameters_file, "PARAMETERS%s", input_string) ;
while (strcmp(input_string, "END_PARAMETERS") != 0)
{
        parameter_index = 0 ;
        while (true)
        {
                if (strcmp(parameters[parameter_index],
                        input_string) == 0)
                        break ;
                if (strcmp(parameters[parameter_index],
                        "LAST_ITEM") == 0)
                {
                        fprintf(output_file, "%s%s", input_string,
                                "\n\n") ;
                        abort_run("Parameter not identified") ;
                }
                parameter_index++ ;
        }

        switch (parameter_index)
        {
                case 0: /* Opening characters for a comment */
                {
                        fscanf(parameters_file, "%s", input_string) ;
                        while (strcmp(input_string, "*/") != 0)
                                fscanf(parameters_file, "%s",
                                input_string) ;
                        goto L1 ;
                }

                case 1: /* "PROBABILITIES" */
                {
                        fscanf(parameters_file, "%s", input_string) ;
                        if (strcmp(input_string, "ON") == 0)
                                probabilities = ON ;
                        else if (strcmp(input_string, "OFF") == 0)
                                probabilities = OFF ;
                        else
                        {
                                fprintf(output_file, "%s%s",
                                        input_string, "\n\n") ;
                                abort_run("Invalid parameter") ;
                        }
                        goto L1 ;
                }

                case 2: /* "SHOW_AL_STRUCTURE" */
                {
                        fscanf(parameters_file, "%s", input_string) ;
```

```
        if (strcmp(input_string, "ON") == 0)
                show_al_structure = ON ;
        else if (strcmp(input_string, "OFF") == 0)
                show_al_structure = OFF ;
        else
        {
                fprintf(output_file, "%s%s",
                        input_string, "\n\n") ;
                abort_run("Invalid parameter") ;
        }
        goto L1 ;
}

case 3: /* "SHOW_HIT_STRUCTURE" */
{
        fscanf(parameters_file, "%s", input_string) ;
        if (strcmp(input_string, "ON") == 0)
                show_hit_structure = ON ;
        else if (strcmp(input_string, "OFF") == 0)
                show_hit_structure = OFF ;
        else
        {
                fprintf(output_file, "%s%s",
                        input_string, "\n\n") ;
                abort_run("Invalid parameter") ;
        }
        goto L1 ;
}

case 4: /* "SHOW_UNSELECTED_ALIGNMENTS" */
{
        fscanf(parameters_file, "%s", input_string) ;
        if (strcmp(input_string, "ON") == 0)
                show_unselected_alignments = ON ;
        else if (strcmp(input_string, "OFF") == 0)
                show_unselected_alignments = OFF ;
        else
        {
                fprintf(output_file, "%s%s",
                        input_string, "\n\n") ;
                abort_run("Invalid parameter") ;
        }
        goto L1 ;
}

case 5: /* "VERBOSE" */
{
        fscanf(parameters_file, "%s", input_string) ;
        if (strcmp(input_string, "ON") == 0)
                verbose = ON ;
        else if (strcmp(input_string, "OFF") == 0)
                verbose = OFF ;
        else
        {
                fprintf(output_file, "%s%s",
                        input_string, "\n\n") ;
                abort_run("Invalid parameter") ;
        }
        goto L1 ;
}

case 6: /* "USE_ABBREVIATIONS" */
{
        fscanf(parameters_file, "%s", input_string) ;
        if (strcmp(input_string, "LETTERS") == 0)
                use_abbreviations = LETTERS ;
        else if (strcmp(input_string, "DIGITS") == 0)
                use_abbreviations = DIGITS ;
        else if (strcmp(input_string, "OFF") == 0)
                use_abbreviations = OFF ;
        else
        {
                fprintf(output_file, "%s%s",
                        input_string, "\n\n") ;
                abort_run("Invalid parameter") ;
        }
        goto L1 ;
}

case 7: /* "SHOW_ALL_PARSING_ALIGNMENTS" */
{
        fscanf(parameters_file, "%s", input_string) ;
        if (strcmp(input_string, "ON") == 0)
```

```
                show_all_parsing_alignments = ON ;
        else if (strcmp(input_string, "OFF") == 0)
                show_all_parsing_alignments = OFF ;
        else
        {
                fprintf(output_file, "%s%s",
                        input_string, "\n\n") ;
                abort_run("Invalid parameter") ;
        }
        goto L1 ;
}

case 8: /* "ID_C_SYMBOL_CONSTRAINT" */
{
        fscanf(parameters_file, "%s", input_string) ;
        if (strcmp(input_string, "ON") == 0)
                id_c_symbol_constraint = ON ;
        else if (strcmp(input_string, "OFF") == 0)
                id_c_symbol_constraint = OFF ;
        else
        {
                fprintf(output_file, "%s%s",
                        input_string, "\n\n") ;
                abort_run("Invalid parameter") ;
        }
        goto L1 ;
}

case 9: /* "ALIGNMENT_FORMAT" */
{
        fscanf(parameters_file, "%s", input_string) ;
        if (strcmp(input_string, "H") == 0)
                alignment_format = 'H' ;
        else if (strcmp(input_string, "V") == 0)
                alignment_format = 'V' ;
        else
        {
                fprintf(output_file, "%s%s",
                        input_string, "\n\n") ;
                abort_run("Invalid parameter") ;
        }
        goto L1 ;
}

case 10: /* "HIT_STRUCTURE_ROWS" */
{
        fscanf(parameters_file, "%d",
                &hit_structure_rows) ;
        goto L1 ;
}

case 11: /* "FAIL_SCORE" */
{
        fscanf(parameters_file, "%d", &fail_score) ;
        goto L1 ;
}

case 12: /* "COST_FACTOR" */
{
        fscanf(parameters_file, "%lg", &cost_factor) ;
        goto L1 ;
}

case 13: /* "KEEP_ROWS" */
{
        fscanf(parameters_file, "%d",
                &keep_rows) ;
        if (keep_rows > MAX_NEW_ARRAY_ROWS)
                abort_run("keep_rows too large") ;
        goto L1 ;
}

case 14: /* "DRIVING_KEEP_ROWS" */
{
        fscanf(parameters_file, "%d",
                &driving_keep_rows) ;
        goto L1 ;
}

case 15: /* "FULL_ALIGNMENT_LIMIT" */
{
        fscanf(parameters_file, "%d",
                &full_alignment_limit) ;
```

```
        goto L1 ;
}

case 16: /* "GRAMMAR_LIMIT" */
{
        fscanf(parameters_file, "%d",
                &grammar_limit) ;
        goto L1 ;
}

case 17: /* "EXTRACTION_LIMIT" */
{
        fscanf(parameters_file, "%d",
                &extraction_limit) ;
        goto L1 ;
}

case 18: /* "FIGURE_ID" */
{
        fscanf(parameters_file, "%s", figure_ID) ;
        goto L1 ;
}

case 19: /* "COMBINATION_LIMIT" */
{
        fscanf(parameters_file, "%d", &combination_limit) ;
        goto L1 ;
}

case 20: /* "MAX_OLD_GAP" */
{
        fscanf(parameters_file, "%d", &max_old_gap) ;
        goto L1 ;
}

case 21: /* "MAX_DRIVING_GAP" */
{
        fscanf(parameters_file, "%d",
                &max_driving_gap) ;
        goto L1 ;
}

case 22: /* "MAX_TARGET_GAP" */
{
        fscanf(parameters_file, "%d", &max_target_gap) ;
        goto L1 ;
}

case 23: /* "MAX_UNSUPPORTED_CYCLES" */
{
        fscanf(parameters_file, "%d",
                &max_unsupported_cycles) ;
        goto L1 ;
}

case 24: /* "MAX_ALIGNMENTS_IN_ONE_CYCLE" */
{
        fscanf(parameters_file, "%d",
                &max_alignments_in_one_cycle) ;
        goto L1 ;
}

case 25: /* "ORIENTATION" */
{
        fscanf(parameters_file, "%s", input_string) ;
        if (strcmp(input_string, "LANDSCAPE") == 0)
                orientation = LANDSCAPE ;
        else if (strcmp(input_string, "PORTRAIT") == 0)
                orientation = PORTRAIT ;
        else
        {
                fprintf(output_file, "%s%s",
                        input_string, "\n\n") ;
                abort_run("Invalid parameter") ;
        }
        goto L1 ;
}

case 26: /* "WRITE_AL_CHARS_LENGTH" */
{
        fscanf(parameters_file, "%d",
                &write_al_chars_length) ;
```

```
        goto L1 ;
}

case 27: /* "WRITE_SECTION_CHARS_LENGTH" */
{
        fscanf(parameters_file, "%d",
                &write_section_chars_length) ;

        goto L1 ;
}

case 28: /* "MINIMUM_FONT_HEIGHT" */
{
        fscanf(parameters_file, "%lg",
                &minimum_font_height) ;

        // Find the smallest available font size
        // which is larger than or equal to the
        // minimum_font_height and re-set the value
        // of minimum font height to this value.

        for (i = 0; i < FONT_SET_SIZE; i++)
        {
                if (font_heights[i] >=
                        minimum_font_height)
                {
                        j = i ;
                        break ;
                }
        }

        if (i >= FONT_SET_SIZE)
        abort_run("MINIMUM_FONT_HEIGHT is \
                                                                larger than the largest \
                                                                available font size") ;


        if (minimum_font_height == font_heights[j])
        {
                fprintf(output_file, "%s%1.2f%s",
                        "MINIMUM_FONT_HEIGHT is ",
                        minimum_font_height,
                        " pt\n\n") ;
        }
        else
        {
                fprintf(output_file, "%s%1.2f%s",
                        "MINIMUM_FONT_HEIGHT changed from ",
                        minimum_font_height,
                        " pt to ") ;
                minimum_font_height = font_heights[j] ;
                fprintf(output_file, "%1.2f%s",
                        minimum_font_height,
                        " pt\n\n") ;
        }

        goto L1 ;
}

case 29: /* "MAXIMUM_FONT_HEIGHT" */
{
        fscanf(parameters_file, "%lg",
                &maximum_font_height) ;

        // Find the largest available font size
        // which is smaller than or equal to the
        // maximum_font_height and re-set the value
        // of maximum font size to this value.

        for (i = FONT_SET_SIZE - 1; i >= 0; i--)
        {
                if (font_heights[i] <=
                        maximum_font_height)
                {
                        j = i ;
                        break ;
                }
        }

        if (i < 0) abort_run("MAXIMUM_FONT_HEIGHT is \
             smaller than the smallest \
                 available font size") ;

        if (maximum_font_height == font_heights[j])
```

```
                        {
                                fprintf(output_file, "%s%1.2f%s",
                                        "MAXIMUM_FONT_HEIGHT is ",
                                        maximum_font_height,
                                        " pt\n\n") ;
                        }
                        else
                        {
                                fprintf(output_file, "%s%1.2f%s",
                                        "MAXIMUM_FONT_HEIGHT changed from ",
                                        maximum_font_height,
                                        " pt to ") ;
                                maximum_font_height = font_heights[j] ;
                                fprintf(output_file, "%1.2f%s",
                                        maximum_font_height,
                                        " pt\n\n") ;
                        }

                        goto L1 ;
                }

                case 30: /* "N_GRAMMARS" */
                {
                        fscanf(parameters_file, "%d",
                                &n_grammars) ;

                        goto L1 ;
                }

                case 31: /* "CREATE_PATTERNS_DIAGNOSTIC" */
                {
                        fscanf(parameters_file, "%s", input_string) ;
                        if (strcmp(input_string, "ON") == 0)
                                create_patterns_diagnostic = ON ;
                        else if (strcmp(input_string, "OFF") == 0)
                                create_patterns_diagnostic = OFF ;
                        else
                        {
                                fprintf(output_file, "%s%s",
                                        input_string, "\n\n") ;
                                abort_run("Invalid parameter") ;
                        }
                        goto L1 ;
                }

                default:
                {
                        fprintf(output_file, "%d%s",
                                parameter_index, "\n\n") ;
                        abort_run("Default reached in read_parameters");
                }
        }
        L1: fscanf(parameters_file, "%s", input_string) ;
    }

    if (maximum_font_height < minimum_font_height)
            abort_run("MAXIMUM_FONT_HEIGHT is smaller \
            than MINIMUM_FONT_HEIGHT") ;
} /* read_parameters */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**               Load the SP corpus from the text file specified (using
**       the 'syntax buffer' size specified), converting it to
**       it's internal format during the process.
**
** CALLING SEQUENCE:
**
**       void load() ;
**
** FORMAL ARGUMENTS:
**
**       Return value:        None.
**
** IMPLICIT INPUTS:
**
**       Input to load() from parameters file.
**
** IMPLICIT OUTPUTS:
**
```

```
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void load()
{
        read_key() ;

        /* Read in syntax. */

        syntax_buffer = new char[BUFSIZE] ;
        if (syntax_buffer == NIL)
                abort_run("Out of memory in 'load'") ;

        in_syntax() ;

        identification_symbols_marked = false ;

        if (corpus = (group *)top_object())
                fprintf(output_file, "Corpus loaded successfully.\n\n") ;
        else
        {
                bad_syntax() ;
                abort_run("Error in syntax in 'load'") ;
        }

        delete[] syntax_buffer ;

} /* load */

/*****************************************************************/

/*
** FUNCTIONAL DESCRIPTION:
**
**        A function for incrementing an integer value by 1 and, at
**        the same time, checking whether it equals or exceeds a specified limit.
**        If it does exceed the limit, the program is aborted with a
**        specified message.
**
** CALLING SEQUENCE:
**
**        void plus_one(int *var1, int limit, char *message)
**
** FORMAL ARGUMENTS:
**
**        Return value:        void.
**
**        var1:                        A pointer to an external variable holding
**                                       the input value.
**        limit:                       The ceiling which must not be equalled or exceeded.
**        message:               The message to go to write file if the ceiling is
**                                       breached.
**
** IMPLICIT INPUTS:
**
**        NONE
**
** IMPLICIT OUTPUTS:
**
**        NONE
**
** SIDE EFFECTS:
**
**        NONE
**
*/

void plus_one(int *var1, int limit, char *message)
{
        (*var1)++ ;
        if (*var1 >= limit) abort_run(message) ;
} // plus_one

/*****************************************************************/
```

2.5 SP71_p.txt

```
PARAMETERS

/* Optional, ON or OFF. */

PROBABILITIES OFF /* Calculations of probabilities at the end
        of each program run. */

SHOW_AL_STRUCTURE OFF /* If ON, the structure of each column of
        each alignment is printed. If OFF, the structure is
        not printed. */

SHOW_HIT_STRUCTURE OFF /* If ON, the hit structure is displayed at one
        or more selected points in the program. If OFF, they are not
        displayed. */

SHOW_UNSELECTED_ALIGNMENTS OFF /* If ON, all the alignments which are
        formed are shown before selection (as well as showing the
        alignments which are selected). */

VERBOSE OFF /* When OFF, lots of messages are printed, otherwise
        minimal messages are printed. */

USE_ABBREVIATIONS OFF /* When LETTERS, alignments are printed
        using abbreviated names of symbols in alphabetic form
        (any symbol which starts with a digit is left as it is).
        When DIGITS, symbols are  abbreviated as digits (which
        may include any trailing or leading '#' symbol which
        appears in the original symbol). When OFF, no abbreviations
        are used. */

SHOW_ALL_PARSING_ALIGNMENTS ON /* If ON, all alignments produced by
        by compressing New against Old are shown. If OFF, only the best
        ones for any given pattern from New are shown. */

ID_C_SYMBOL_CONSTRAINT ON /* If this is ON, ID-symbols can only ever
        be matched with C-symbols and vice versa. If the parameter is OFF,
        the constraint is not applied. */

/* Other parameters. */

ALIGNMENT_FORMAT H /* The format of alignments: 'V' means
        'vertical' and 'H' means 'horizontal'. */

HIT_STRUCTURE_ROWS 20000 /* The number of rows that can be used in
        the leaf_array[]. */

FAIL_SCORE -100 /* Any compression difference which is equal to this or
        smaller counts as a fail. */

COST_FACTOR 100 /* The multiplier to convert min_cost for each
        symbol to the actual_cost (with rounding up). */

KEEP_ROWS 50 /* Controls the number of rows in the
        new_OAO_array[][] which are used to determine which alignments
        are *not* purged at the end of each cycle. */

DRIVING_KEEP_ROWS 5 /* Of the patterns selected by
        KEEP_ROWS, the best DRIVING_KEEP_ROWS are
        selected to be driving patterns for the next cycle. THIS PARAMETER
        IS NOT CURRENTLY USED. */

FULL_ALIGNMENT_LIMIT 30 /* This limits the number of alternative
        alignments (for a given pattern from New) that are considered
        during the compiling of alternative grammars. */

GRAMMAR_LIMIT 30 /* This limits the number of alternative grammars
        that may be developed at any one time (in
        compile_alternative_grammars()) */

EXTRACTION_LIMIT 10 /* This limits the number of alignments for
        a given driving pattern that are used for learning. */

FIGURE_ID 0 /* The figure number in the article about these
        parsings. */

COMBINATION_LIMIT 10 /* The number of combinations to be selected
        on each phase of combine_alignments(). */

MAX_OLD_GAP 100 /* The maximum gap between two hit symbols in an
        original pattern in Old (not an alignment formed during
        current processing). */
```

```
MAX_DRIVING_GAP 100 /* Sets the maximum permitted gap between
        hit symbols in any driving pattern. */

MAX_TARGET_GAP 100 /* Sets the maximum permitted gap between
        hit symbols in any target pattern. */

MAX_UNSUPPORTED_CYCLES 5 /* For any one pattern from New,
        this parameter sets the maximum number of cycles which
        build alignments without any gain in maximum compression
        score compared with the best cycle for the current window. */

MAX_ALIGNMENTS_IN_ONE_CYCLE 10000 /* This can be used to limit
        the number of alignments formed in one cycle. It is
        applied as alignments are formed, in order of the
        approximate compression scores calculated for
        the hit structure. If this limit is applied, there
        is a possibility of cutting short the alignments
        before the best one has been made. */

ORIENTATION LANDSCAPE /* Orientation of the output may be
        specified as LANDSCAPE or PORTRAIT. */

WRITE_AL_CHARS_LENGTH 1000 /* The maximum size of any
        alignment when written out. Any excess is truncated
        with ellipses. */

WRITE_SECTION_CHARS_LENGTH 80 /* The maximum size of a section of
        any alignment when written out. */

MINIMUM_FONT_HEIGHT 12 /* The smallest font to be used in printing
        alignments (in pt). */

MAXIMUM_FONT_HEIGHT 18 /* The largest font to be used in printing
        alignments (in pt). */

N_GRAMMARS 5 /* This is the number of grammars, over and obove the
        'naive' grammar, which provide a set of Old patterns after
        each New pattern has been processed. */

CREATE_PATTERNS_DIAGNOSTIC OFF /* When ON, prints out diagnostic
        information for sequence::create_patterns() methods. Otherwise,
        no diagnostic information is printed. */

/* FORCING OF CODE AND CONTENTS STATUS FOR SYMBOLS

If any symbols in the corpus have a '!' prefix, this
suppresses the normal mechanism for classifying symbols
as IDENTIFICATION or CONTENTS. In this case, any symbol prefixed
by '!' is IDENTIFICATION and all other symbols are CONTENTS. The
'!' character does not form part of the symbol. */

END_PARAMETERS
```

2.6 SP71_od.txt

```
OBJECTIVES AND DEBRIEFING FOR SP71

JGW

%1 18/6/03

SP71 is a successor to SP70, derived from that model, and designed to overcome its weaknesses.
Here are the main ideas for developing this model:

1 The process of building multiple alignments should be modified so that, instead of discarding
alignments that contain mismatches between Old patterns, the system uses those mismatches as an
opportunity to create new patterns (see sp70_od, %141).

2 With this modification, the best alignment created for any given PFN should be a 'full' parsing
of the PFN. So it should not be necessary to re-parse the PFNs in order to derive frequency values
needed for the process of compiling grammars. After a single parsing of the PFNs, it should be
possible to move straight on to compiling alternative grammars.

3 When the program builds new structures, it should be able to form 'discontinuous' patterns as well
as coherent patterns -- to avoid the kind of anomaly described at the beginning of sp70_od, %139.
Hopefully, this should enable the system to learn discontinuous dependencies in NL syntax automatically.
```

4 At some stage, the program may be modified so that it processes New patterns in batches and, between each batch, it compiles alternative grammars and then purges Old of all patterns that are not in the best grammar or in the best two or three grammars. This should speed up processing because there would be a lot less rubbish accumulating in Old and, for that reason, less matching to be done in the process of building multiple alignments.

5 With luck, these modifications should overcome the main weakness of SP70: that it is not good at finding intermediate levels of structure in grammars. As SP71 develops, this aspect of its capabilities needs to be checked.

%2 26/6/03

VERSION 1.0

The first version eliminates the need for reparsing of New before sifting_and_sorting() because it includes composite alignments, derived from parsing, in the set of full alignments from parsing.

If it is run on input like this:

```
[
    [
       (j o h n r u n s)
       (m a r y r u n s)
       (j o h n w a l k s)
       (m a r y w a l k s)
    ]
    [
    ]
]
```

it produces grammars like this:

Grammar GR20, G = 14271.28, E = 149.39, score = 14420.67:

ID523: (< %2 10 n >)

ID524: (< %3 1 j o h n r u >)

ID525: (< %4 2 r u n s >)

ID526: (< %4 3 s >)

ID527: (< 7 < %3 > < %2 > < %4 > >)

ID528: (< %5 4 j o h n >)

ID529: (< %5 5 m a r y >)

ID530: (< 6 < %5 > < %4 > >)

ID531: (< %1 8 w a l k >)

ID532: (< 9 < %5 > < %1 > < %4 > >)

Grammar GR21, G = 14278.23, E = 153.89, score = 14432.13:

ID533: (< %2 10 n >)

ID534: (< %3 1 j o h n r u >)

ID535: (< %4 2 r u n s >)

ID536: (< %4 3 s >)

ID537: (< 7 < %3 > < %2 > < %4 > >)

ID538: (< %5 4 j o h n >)

ID539: (< %5 %6 5 m a r y >)

ID540: (< 6 < %6 > < %4 > >)

ID541: (< %1 8 w a l k >)

ID542: (< 9 < %5 > < %1 > < %4 > >)

Grammar GR22, G = 14338.23, E = 158.70, score = 14496.92:

ID543: (< %3 12 n >)

ID544: (< %4 1 j o h n r u >)

ID545: (< %5 2 r u n s >)

```
ID546: (< %5 3 s >)

ID547: (< 7 < %4 > < %3 > < %5 > >)

ID548: (< %6 4 j o h n >)

ID549: (< %6 5 m a r y >)

ID550: (< 6 < %6 > < %5 > >)

ID551: (< %1 8 w a l k >)

ID552: (< 9 < %6 > < %1 > < %5 > >)

ID553: (< %2 10 < %6 > < %1 > >)

ID554: (< 11 < %2 > < %5 > >)
```

These are not very tidy, largely because the early parses are poor. This problem can be overcome by running the program on input like this:

```
[
    [
        (j o h n r u n s)
        (m a r y r u n s)
        (j o h n w a l k s)
        (m a r y w a l k s)
        (j o h n r u n s)
        (m a r y r u n s)
        (j o h n w a l k s)
        (m a r y w a l k s)
    ]
    [
    ]
]
```

and arranging that it compiles its grammars on a 'window' or subset of the New patterns which is always the most recently processed. If the window is set to 4, the best grammars found are these:

```
Grammar GR153, G = 9302.81, E = 165.10, score = 9467.90:

ID1627: (< %2 3 s >)

ID1628: (< %3 1 j o h n >)

ID1629: (< %3 2 m a r y >)

ID1630: (< %1 4 r u n >)

ID1631: (< %1 5 w a l k >)

ID1632: (< 6 < %3 > < %1 > < %2 > >)

Grammar GR154, G = 9367.69, E = 172.28, score = 9539.97:

ID1633: (< %3 3 s >)

ID1634: (< %4 1 j o h n >)

ID1635: (< %4 2 m a r y >)

ID1636: (< %1 4 r u n >)

ID1637: (< %1 5 w a l k >)

ID1638: (< 6 < %4 > < %1 > < %3 > >)

ID1639: (< %2 7 < %4 > < %1 > >)

ID1640: (< 8 < %2 > < %3 > >)

Grammar GR155, G = 9367.69, E = 172.28, score = 9539.97:

ID1641: (< %3 3 s >)

ID1642: (< %4 1 j o h n >)

ID1643: (< %4 2 m a r y >)

ID1644: (< %1 4 r u n >)
```

```
ID1645: (< %1 5 w a l k >)

ID1646: (< 6 < %4 > < %1 > < %3 > >)

ID1647: (< %2 7 < %4 > < %1 > >)

ID1648: (< 8 < %2 > < %3 > >)
```

The last two are identical, presumably produced by two different routes. There may be a need to check for identical grammars and eliminate duplicates. It is interesting that in grammars GR154 and GR155, there is an intermediate level of structure (ID1647), although it happens to be spurious in this case.

%3 26/6/03

VERSION 1.0

With a trailing 'window' of 8 and this input:

```
[
    [
       (t h a t b o y r u n s)
       (t h a t g i r l r u n s)
       (t h a t b o y w a l k s)
       (t h a t g i r l w a l k s)
       (s o m e b o y r u n s)
       (s o m e g i r l r u n s)
       (s o m e b o y w a l k s)
       (s o m e g i r l w a l k s)
       (t h a t b o y r u n s)
       (t h a t g i r l r u n s)
       (t h a t b o y w a l k s)
       (t h a t g i r l w a l k s)
       (s o m e b o y r u n s)
       (s o m e g i r l r u n s)
       (s o m e b o y w a l k s)
       (s o m e g i r l w a l k s)
    ]
    [
    ]
]
```

the best grammar found is:

Grammar GR508, G = 29209.49, E = 463.45, score = 29672.95:

```
ID3935: (< %4 26 b o y r u n s >)

ID3936: (< %29 %37 %41 %48 218 t h a t >)

ID3937: (< 702 t h a t g i r l w a l k s >)

ID3938: (< %29 %37 %41 %48 1497 s o m e >)

ID3939: (< 1502 < %29 > < %4 > >)

ID3940: (< %36 1855 g i r l r u n s >)

ID3941: (< 1864 < %37 > < %36 > >)

ID3942: (< %40 2150 b o y w a l k s >)

ID3943: (< 2159 < %41 > < %40 > >)

ID3944: (< %47 2388 g i r l w a l k s >)

ID3945: (< 2397 < %48 > < %47 > >)
```

There seem to be two main problems here:

* 'some' and 'that' have both been assigned to classes '%29', '%37', '%41', and '%48' whereas we would expect only one class to be necessary. This seems to be largely because of the second problem.

* Patterns like (< %4 26 b o y r u n s >), (< %36 1855 g i r l r u n s >), (< %40 2150 b o y w a l k s >) and (< %47 2388 g i r l w a l k s >) have not been broken down into words and other consitutents (and this means that the program cannot 'see' that they are all part of one structure).

%4 12/7/03

The main reason for the poor results for version 1.0 of SP71 seems to be that, as the program proceeds, patterns that are 'correct' (eg words) accumulate large numbers of ID symbols.

This leads to low compression scores for alignments that contain those correct patterns so that the alignments that score best are those that analyse each sentence into relatively few, relatively large chunks like '< %47 2388 g i r l w a l k s >'.

A possible answer is to select the best one or two grammars that have been created after each PFN has been processed, clean them up to remove unecessary ID symbols, perhaps rename the ID symbols, and then purge Old of all patterns are that are not in those best one or two grammars. All previous alignments must also be purged.

This will be tried in version 1.1.

%5 8/10/03

DETERMINATION OF FREQUENCY VALUES IN SP71

In SP70, frequency values for the compiling of grammars was determined from a reparsing of all the New patterns after phase 1 (which created the patterns) and only the best 'full' parsings were used.

In SP71, this step is eliminated and, instead, the full parsings created as composite alignments during phase 1 are used to determine the frequencies of patterns. This seems OK but it is important that only the best full parsing for any one pattern from New is used to determine the frequencies of patterns. Otherwise, there is a risk of double-counting and spuriously high frequency values from alternative parsings of the same pattern from New. It is also important to ensure that the initial frequencies of patterns derived in phase 1 are zero.

Actually, the risk of double counting is overcome by the system of looking for the maximum frequency of any given pattern in any *one* full parsing of a given pattern from New.

%6 20/10/03

The 'incremental' approach that avoids reparsing will probably best in the long run -- when large samples are used. But with small samples that are currently used for experimentation, it is not possible to derive accurate grammars in this way because early parsings are always inaccurate. With large samples, the inaccuracy of early parsings will tend to fade away.

So for the time being, there is a case for re-introducing the use of re-parsing in order to achieve accurate parsings of the whole sample and, from these, accurate frequency values for patterns.

This can be added to SP71 as a discrete module that can easily be removed later when larger samples are used. The program currently derives frequencies from the parsings in phase 1 and this coding will not be wasted -- it will be used later.

Version 1.2 reintroduces re-parsing.

%7 22/10/03

One of the problems with v 1.2 is that, in Phase 2, it produces many alternative parses like this:

ALIGNMENT

ID1799: NSC = 300.48, EC = 77.79, CR = 0.26, CD = 222.69,
Absolute P = 3.82052625877e-024

```
0                         m a r y                   w a l k s     0
                          | | | |                   | | | | |
1                         | | | |   < %24 %29 %4 439 w a l k s >   1
                          | | | |   |  |                        |
2 < %32 808 < %7          | | | | > < %24                    > > 2
            | |           | | | | |
3           < %7 %9 %11 152 m a r y >                              3
```

ALIGNMENT

ID1803: NSC = 300.48, EC = 79.68, CR = 0.27, CD = 220.80,
Absolute P = 1.03256250715e-024

```
0                         m a r y                   w a l k s     0
                          | | | |                   | | | | |
1                         | | | |   < %24 %29 %4 439 w a l k s >   1
                          | | | |   |  |                        |
2 < %25 443 <    %9       | | | | > < %24                    > > 2
            |    |        | | | | |
3           < %7 %9 %11 152 m a r y >                              3
```

ALIGNMENT

ID1805: NSC = 300.48, EC = 80.70, CR = 0.27, CD = 219.79,
Absolute P = 5.10701445814e-025

```
0                       m a r y                 w a l k s     0
```

```
                        | | | |                   | | | | |
1                       | | | |   < %24 %29 %4 439 w a l k s >   1
                        | | | |   |         |               |
2 < %8 157 < %7         | | | | > <         %4              > > 2
          | |           | | | | |
3         < %7 %9 %11 152 m a r y >                           3
```

ALIGNMENT

ID1808: NSC = 300.48, EC = 81.59, CR = 0.27, CD = 218.89,
Absolute P = 2.74512416997e-025

```
0                       m a r y                   w a l k s     0
                        | | | |                   | | | | |
1                       | | | |   < %24 %29 %4 439 w a l k s >   1
                        | | | |   |      |                |
2 < %30 470 <    %9     | | | | > <     %29               > > 2
           |    |       | | | | |
3          < %7 %9 %11 152 m a r y >                           3
```

ALIGNMENT

ID1814: NSC = 300.48, EC = 82.58, CR = 0.27, CD = 217.90,
Absolute P = 1.38025792673e-025

```
0                       m a r y                   w a l k s     0
                        | | | |                   | | | | |
1                       | | | |   < %24 %29 %4 439 w a l k s >   1
                        | | | |   |         |               |
2 < %10 168 <    %9     | | | | > <         %4              > > 2
           |    |       | | | | |
3          < %7 %9 %11 152 m a r y >                           3
```

ALIGNMENT

ID1819: NSC = 300.48, EC = 83.54, CR = 0.28, CD = 216.94,
Absolute P = 7.09237227434e-026

```
0                       m a r y                   w a l k s     0
                        | | | |                   | | | | |
1                       | | | |   < %24 %29 %4 439 w a l k s >   1
                        | | | |   |         |               |
2 < %12 177 <       %11 | | | | > <         %4              > > 2
           |         |  | | | | |
3          < %7 %9 %11 152 m a r y >                           3
```

This means that the 'correct' full alignment, here:

ID756 : ID616 : ID501 : #1832: NSC = 284.84, EC = 80.99, CR = 0.28, CD = 203.84,
Absolute P = 4.16036621852e-025

```
0                       m a r y               w a l k          s     0
                        | | | |               | | | |          |
1                       | | | |   < %22 %27 428 w a l k >        |     1
                        | | | |   |  |                 |        |
2 < %23 433 <    %9     | | | | > < %22               > < %4    | > > 2
           |    |       | | | | |                       | |     | |
3          < %7 %9 %11 152 m a r y >                       | |     | |   3
                                                           | |     | |
4                                                          < %4 15 s >   4
```

gets swamped by the other full alignments.

%8 27/10/03

With a cost_factor of 10, the best two grammars are:

TIDIED UP GR224

Grammar GR224, Derived from ID134, G = 1093.19, E = 224.13, score = 1317.32:

ID2638: (< %3 3 j o h n >)*2

ID2639: (< 6 < %3 > < %1 > < %2 > >)*4

ID2640: (< %1 4 r u n >)*2

ID2641: (< %2 1 s >)*4

ID2642: (< %3 2 m a r y >)*2

ID2643: (< %1 5 w a l k >)*2

```
TIDIED UP GR225

Grammar GR225, Derived from ID205, G = 1119.63, E = 221.49, score = 1341.11:

ID2644: (< %2 3 j o h n >)*2

ID2645: (< 4 < %2 > < %1 > >)*4

ID2646: (< %1 1 r u n s >)*2

ID2647: (< %2 2 m a r y >)*2

ID2648: (< %1 5 w a l k s >)*2

However, with a cost_factor of 100, the best two grammars are:

TIDIED UP GR207

Grammar GR207, Derived from ID187, G = 9694.97, E = 223.50, score = 9918.47:

ID2643: (< %3 3 j o h n >)*2

ID2644: (< 6 < %3 > < %1 > < %2 > >)*4

ID2645: (< %1 4 r u n >)*2

ID2646: (< %2 1 s >)*4

ID2647: (< %3 2 m a r y >)*2

ID2648: (< %1 5 w a l k >)*2

TIDIED UP GR208

Grammar GR208, Derived from ID188, G = 9766.31, E = 233.58, score = 9999.88:

ID2649: (< %4 3 j o h n >)*2

ID2650: (< 6 < %4 > < %1 > < %3 > >)*4

ID2651: (< %1 4 r u n >)*2

ID2652: (< %3 1 s >)*4

ID2653: (< %4 2 m a r y >)*2

ID2654: (< %2 7 < %4 > < %1 > >)*2

ID2655: (< 8 < %2 > < %3 > >)*2

ID2656: (< %1 5 w a l k >)*2

This is because the symbols from New (j, o, h, etc) carry so much 'weight' that almost
any grammar that minimises the number of such symbols has a better score than ones that
don't. In this example, the next two grammars are:

TIDIED UP GR209

Grammar GR209, Derived from ID197, G = 9766.31, E = 233.58, score = 9999.88:

ID2657: (< %4 3 j o h n >)*2

ID2658: (< 6 < %4 > < %1 > < %3 > >)*4

ID2659: (< %1 4 r u n >)*2

ID2660: (< %3 1 s >)*4

ID2661: (< %4 2 m a r y >)*2

ID2662: (< %1 5 w a l k >)*2

ID2663: (< %2 7 < %4 > < %1 > >)*2

ID2664: (< 8 < %2 > < %3 > >)*2

TIDIED UP GR210

Grammar GR210, Derived from ID169, G = 10203.52, E = 206.72, score = 10410.24:

ID2665: (< %2 3 j o h n >)*2

ID2666: (< 4 < %2 > < %1 > >)*4
```

```
ID2667: (< %1 1 r u n s >)*2

ID2668: (< %2 2 m a r y >)*2

ID2669: (< %1 5 w a l k s >)*2

The last of these (GR210) corresponds to the second-best grammar when the cost_factor is 10.

If the cost_factor is 1, then the best two grammars are:

TIDIED UP GR72

Grammar GR72, Derived from ID22, G = 238.62, E = 87.10, score = 325.73:

ID4041: (< 3 j o h n r u n s >)*1

ID4042: (< 1 m a r y r u n s >)*1

ID4043: (< 2 j o h n w a l k s >)*1

ID4044: (< 4 m a r y w a l k s >)*1

TIDIED UP GR73

Grammar GR73, Derived from ID23, G = 301.30, E = 135.56, score = 436.86:

ID4045: (< 5 j o h n r u n s >)*1

ID4046: (< %1 1 r u n s >)*1

ID4047: (< 3 < %2 > < %1 > >)*1

ID4048: (< %2 2 m a r y >)*1

ID4049: (< 4 j o h n w a l k s >)*1

ID4050: (< 6 m a r y w a l k s >)*1

%9 27/10/03

A main problem with SP70 and the current version of SP71 (v 1.2) is that it does
not keep track of long-range dependencies in grammars.

With input like this:

[
    [
        (fur backbone milk mammal)
        (wings feathers backbone beak bird)
    ]
    [
    ]
]

SP70 produces a grammar like this:

Grammar ID7, G = 2425.55, E = 61.80, score = 2487.35:

ID138: (< %1 5 backbone >)

ID139: (< %2 6 fur >)

ID140: (< %2 1 wings feathers >)

ID141: (< %3 2 milk mammal >)

ID142: (< %3 3 beak bird >)

ID143: (< 4 < %2 > < %1 > < %3 > >)

As noted in sp70_od, %139, this grammar loses the association between 'fur' and
'milk mammal', and between 'wings feathers' and 'beak bird'. In order to preserve
this information, the grammar needs to include the encoding of the two original
patterns as well as the grammar itself. We are already minimising (G + E) but
we are doing it without including the patterns that represent E. If we simply
include those patterns in the grammar -- and minimise (G + E) as before -- this
should solve the problem of lost information about discontinuous associations.
It should take us a fair way down the road of developing grammars that can keep
track of discontinous dependencies in grammars.

This will be attempted in SP71, v 1.3.
```

%10 3/11/03

Here is the best grammar with the same input as in %9, using SP71, v 1.3:

Grammar GR7, Derived from GR3, G = 2214.08, E = 65.47, score = 2279.55:

Grammar patterns:

ID143: (< %3 2 milk mammal >)*1

ID144: (< 4 < %2 > < %1 > < %3 > >)*2

ID145: (< %2 6 fur >)*1

ID146: (< %1 5 backbone >)*2

ID157: (< %2 1 wings feathers >)*1

ID159: (< %3 3 beak bird >)*1

Code patterns:

ID142: (< 4 6 5 2 >)*1

ID156: (< 4 1 5 3 >)*1

The grammar itself generalises as before. But the code patterns define the two original patterns.

An alternative grammar is:

< %1 fur < %3 > milk mammal >

< %2 wings feathers < %3 > beak bird >

< %3 backbone >

This is, in itself, a lossless compression of the original patterns and does not need code patterns to ensure that it is lossless. It is also simpler than the previous grammar.

Question: how to choose between these alternatives?

It has been argued previously that there is a case for always extracting discrete matched and unmatched sections of the original patterns and making them into free-standing patterns. The reason is that, with samples of realistic size, this will happen anyway. But this strategy seems to lead to anomalous results with small samples as in the case just shown. We really need a strategy that works for samples of any size.

An alternative strategy is to 'do as little as possible'. This seems to favour the alternative grammar just shown.

Another strategy is to create new patterns *only* when two or more patterns are unified. This would naturally tend to preserve 'discontinuous' associations such as that between 'fur' and 'milk mammal' and between 'wings feathers' and 'beak bird'.

Another strategy is to make new patterns for unmatched sequences as well as matched sequences but to preserve information about their original contexts something like this:

< %3 2 milk mammal >

< 4a < %2 6 > < %1 > < %3 2 > >

< 4b < %2 1 > < %1 > < %3 3 > >

< %2 6 fur >

< %1 5 backbone >

< %2 1 wings feathers >

< %3 3 beak bird >

%11 4/11/03

Here are two ways in which SP71 may work:

1 Given an alignment like this:

```
a b c h i d e f j k g h i
| | |     | | |     | | |
a b c p q d e f r s g h i
```

it will create patterns like this:

< %1 0 a b c < %2 > d e f < %3 > g h i >

< %4 1 < %1 < %2 h i > < %3 j k > > >

< %5 2 < %1 < %2 p q > < %3 r s > > >

or, perhaps, this:

< %1 0 a b c < %2 > d e f < %3 > g h i >

< %1 0 < %2 h i > < %3 j k > >

< %1 0 < %2 p q > < %3 r s > >

2 Given the same alignment, it may create patterns like this:

Grammar GR7, Derived from GR3, G = 19254.49, E = 99.61, score = 19354.10:

Grammar patterns:

ID1342: (< %4 9 d e f >)*2

ID1343: (< %8 25 < %3 > < %5 > < %4 > < %7 > < %6 > >)*2

ID1344: (< %6 16 g h i >)*2

ID1345: (< %7 19 y z >)*1

ID1346: (< %3 6 a b c >)*2

ID1347: (< %5 12 w x >)*1

ID1363: (< %7 20 r s >)*1

ID1365: (< %5 13 p q >)*1

Code patterns:

ID1341: (< %8 25 6 12 9 19 16 >)*1

ID1359: (< %8 25 6 13 9 20 16 >)*1

The second set of patterns may be converted to something close to the first
set by applying rules something like this:

* If one pattern is contained within another in one of the full alignments
from parsing and, if it has the same frequency as the containing pattern,
then it may be merged with the containing pattern.

* A similar principle may apply to the code patterns.

This may be a falicy. Consider the following patterns:

A {B, C} D (fr 2) generates A B D (fr 1), A C D (fr 1)

and

X (B, E} Y (fr 2) generates X B Y (fr 1), X C Y (fr 1)

In this case, B has a frequency of 2 but it would not be appropriate to merge
it with the pattern A {B, C} D.

A better rule is:

* If one pattern only occurs as part of another pattern and it always appears
when the containing pattern appears, the two patterns may be merged.

* Likewise for code patterns.

Although, in principle, it might be possible to clean up grammars in this way,
a better option may be to develop SP71 like this:

1 (SP71, v 1.4). From a partial alignment of two patterns like this:

```
a b c h i d e f j k g h i
| | |     | | |     | | |
a b c p q d e f r s g h i
```

will be derived patterns like this:

```
< %1 0 a b c < %2 > d e f < %3 > g h i >

< %1 0 < %2 h i > < %3 j k > >

< %1 0 < %2 p q > < %3 r s > >
```

and, from an alignment like this:

```
a b c h i d e f     g h i
| | |     | | |     | | |
a b c p q d e f r s g h i
```

it will create patterns like this:

```
< %1 0 a b c < %2 > d e f < %3 > g h i >

< %1 0 < %2 h i > < %3 > >

< %1 0 < %2 p q > < %3 r s > >
```

2 (SP71, v 1.5). This method of forming new patterns will be invoked during parsing in Phase 1 so that, whenever a 'mis-match' occurs between two patterns from Old, the program will form new patterns instead of rejecting the offending alignment.

The focus will on the matching of *two* patterns, regardless of whether one or both of them are alignments. This means that it should not be necessary to worry about what happens when there is a partial match between the New pattern and *two or more* Old patterns. The process of looking for 'the most abstract pattern' and deriving new patterns from this pattern (and from the New pattern) should not be necessary.

%12 9/11/03

With these New patterns:

```
ID1 (fur backbone milk mammal)*1

ID2 (wings feathers backbone beak bird)*1
```

SP71, v 1.4 produces these patterns in Old:

```
ID3: (< %1 : fur backbone milk mammal >)*0

ID4: (< %2 : wings feathers backbone beak bird >)*0

ID6: (< %5 : < %3 > backbone < %4 > >)*1

ID8: (< %5 8 : < %3 wings feathers > < %4 beak bird > >)*1

ID9: (< %5 9 : < %3 fur > < %4 milk mammal > >)*1
```

The last 3 represent a 'lossless' grammar of the original two patterns but there may be a better alternative, something like this:

```
(< %1 backbone >)*1

(< %2 wings feathers < %1 > beak bird >)*1

(< %3 fur < %1 > milk mammal >)*1
```

This achieves the same effect but is simpler. It may also lend itself better to the formation of disjunctive classes.

The rule seems to be: "Extract unified patterns as new patterns and preserve the difference patterns with gaps". It achieves lossless compression even if there is more than one unified pattern in a given alignment.

This will be tried in v 1.5.

%13 10/11/03

SP71, v 1.5 creates these Old patterns:

```
ID3: (< %1 fur backbone milk mammal >)*0

ID4: (< %2 wings feathers backbone beak bird >)*0

ID7: (< %3 backbone >)*1

ID8: (< %4 8 wings feathers < %3 > beak bird >)*1

ID9: (< %5 9 fur < %3 > milk mammal >)*1
```

Now, in v 1.6, I will try getting rid of the feature in Phase 1 that discards
alignments if they contain mismatches between Old patterns and replacing it
with learning from the mismatched patterns. It may happen that the driving
pattern or the target pattern or both of them are alignments of two or more
basic patterns. However, all such alignments will be treated as simple patterns
so that the new routine that extracts new classes from an alignment between
two patterns should always work.

%14 10/11/03

MORE THOUGHTS ABOUT EXTRACTION OF PATTERNS

There seem to be two main 'styles' for the extraction of patterns from an
alignment of two patterns:

```
wings feathers backbone beak bird
                  |
fur             backbone milk mammal
```

1 'Type 1' as in:

ID6: (< %5 : < %3 > backbone < %4 > >)*1

ID8a: (< %5 8 : < %3 wings feathers > < %4 beak bird > >)*1

ID9a: (< %5 9 : < %3 fur > < %4 milk mammal > >)*1

2 'Type 2' as in:

ID7: (< %3 backbone >)*1

ID8b: (< %4 8 wings feathers < %3 > beak bird >)*1

ID9b: (< %5 9 fur < %3 > milk mammal >)*1

Advantages of Type 1 compared with Type 2 are:

* It allows one to capture the idea of a class hierarchy. This would be
even more obvious if there were several unified symbols in ID6, not just
'backbone'.

* If there was more than one unified symbol in ID6 and if these were intermingled
with non-unified symbols, then it would preserve the sequential associations
amongst the unified symbols. In type 2, this association would be lost.

* It gives us a handle on how to create disjunctive groupings based on shared
context. Examples are {'< %3 wings feathers >', < %3 fur >} and {'< %4 beak bird >',
'< %4 milk mammal >'}.

However, Type 2 also seems to have advantages compared with Type 1:

* It appears to be more economical.

* It gives us a handle on how to infer parts and subparts by multiple alignment.
In this example, the pattern '< %3 backbone >' is clearly a part of ID8b and
ID9b in the way that ID6 is not a part of ID8a and ID9a.

%15 3/12/03

Here is a possible type 3:

```
ID6: (< %5 : < %3 > < %2 > < %4 > >)*1
ID8c: (< %5 8 : < 0 > < 0 > >)*1
ID9c: (< %5 9 : < 1 > < 1 > >)*1
< %2 backbone >
< %3 0 wings feathers >
< %3 1 fur >
< %4 0 beak bird >
< %4 1 milk mammal >
```

Possible variations include:

```
ID6: (< %5 : < %3 > backbone < %4 > >)*1
ID8c: (< %5 8 : < 0 > < 0 > >)*1
ID9c: (< %5 9 : < 1 > < 1 > >)*1
< %3 0 wings feathers >
< %3 1 fur >
< %4 0 beak bird >
< %4 1 milk mammal >
```

and

ID6: (< %5 : < %3 > backbone < %4 > >)*1

```
ID8c: (< %5 8 : 0 0 >)*1
ID9c: (< %5 9 : 1 1 >)*1
< %3 0 wings feathers >
< %3 1 fur >
< %4 0 beak bird >
< %4 1 milk mammal >
```

This is close to what SP70 does, with ID8c and ID9c as the code patterns.
The argument for extracting patterns like < %2 backbone >, < %3 0 wings feathers >
and < %3 1 fur > is that they will eventually have a frequency > 1 so we may as
well treat all sub-patterns in the same way. If, in the final grammar, we find
that a sub-pattern has a frequency of 1 or that it always occurs within one
containing pattern and never in any other context, then we may clean up the
grammar by merging it with its containing pattern.

%16 9/12/03

RESULTS FROM SP71, V 1.9

With this input:

```
[
    [
        (fur backbone milk mammal)
        (wings feathers backbone beak bird)
    ]
    [
    ]
]
```

the two versions of extract_patterns() produce these two versions of the
best two grammars:

Version 1 (like type 1 in %14):

ORIGINAL GRAMMAR GR5

Grammar GR5, Derived from GR2, G = 870.72, E = 11.91, score = 882.63:

Grammar patterns:

ID28: (< %5 8 < %3 fur > < %4 milk mammal > >)*1

ID29: (< %5 < %3 > backbone < %4 > >)*2

ID36: (< %5 7 < %3 wings feathers > < %4 beak bird > >)*1

Code patterns:

ID27: (8)*1

ID35: (7)*1

CLEANED UP GRAMMAR GR5

Grammar GR5, G = 858.86, E = 11.91, score = 870.76:

Grammar patterns:

ID28: (< 8 < %3 fur > < %4 milk mammal > >)*1

ID29: (< < %3 > backbone < %4 > >)*2

ID36: (< 7 < %3 wings feathers > < %4 beak bird > >)*1

Code patterns:

ID27: (8)*1

ID35: (7)*1

TIDIED UP GR5

Grammar GR5, G = 858.86, E = 11.91, score = 870.76:

Grammar patterns:

ID28: (< 2 < %1 fur > < %2 milk mammal > >)*1

ID29: (< < %1 > backbone < %2 > >)*2

ID36: (< 1 < %1 wings feathers > < %2 beak bird > >)*1

```
Code patterns:

ID27: (2)*1

ID35: (1)*1

ORIGINAL GRAMMAR GR6

Grammar GR6, Derived from GR3, G = 892.75, E = 20.50, score = 913.24:

Grammar patterns:

ID31: (< %1 fur backbone milk mammal >)*1

ID39: (< %2 wings feathers backbone beak bird >)*1

Code patterns:

ID30: (< %1 >)*1

ID38: (< %2 >)*1

CLEANED UP GRAMMAR GR6

Grammar GR6, G = 892.75, E = 20.50, score = 913.24:

Grammar patterns:

ID31: (< %1 fur backbone milk mammal >)*1

ID39: (< %2 wings feathers backbone beak bird >)*1

Code patterns:

ID30: (< %1 >)*1

ID38: (< %2 >)*1

TIDIED UP GR6

Grammar GR6, G = 892.75, E = 20.50, score = 913.24:

Grammar patterns:

ID31: (< %1 fur backbone milk mammal >)*1

ID39: (< %2 wings feathers backbone beak bird >)*1

Code patterns:

ID30: (< %1 >)*1

ID38: (< %2 >)*1

Version 2 (like type 2 in %14):

ORIGINAL GRAMMAR GR7

Grammar GR7, Derived from GR3, G = 781.99, E = 31.68, score = 813.67:

Grammar patterns:

ID46: (< %5 8 fur < %3 > milk mammal >)*1

ID47: (< %3 backbone >)*2

ID56: (< %4 7 wings feathers < %3 > beak bird >)*1

Code patterns:

ID45: (< %5 8 >)*1

ID55: (< %4 7 >)*1

CLEANED UP GRAMMAR GR7

Grammar GR7, G = 778.54, E = 31.68, score = 810.21:

Grammar patterns:

ID46: (< %5 8 fur < %3 > milk mammal >)*1

ID47: (< backbone >)*2
```

ID56: (< %4 7 wings feathers < %3 > beak bird >)*1

Code patterns:

ID45: (< %5 8 >)*1

ID55: (< %4 7 >)*1

TIDIED UP GR7

Grammar GR7, G = 778.54, E = 31.68, score = 810.21:

Grammar patterns:

ID46: (< %3 2 fur < %1 > milk mammal >)*1

ID47: (< backbone >)*2

ID56: (< %2 1 wings feathers < %1 > beak bird >)*1

Code patterns:

ID45: (< %3 2 >)*1

ID55: (< %2 1 >)*1

ORIGINAL GRAMMAR GR4

Grammar GR4, Derived from GR2, G = 804.21, E = 20.76, score = 824.97:

Grammar patterns:

ID44: (< %1 fur backbone milk mammal >)*1

ID49: (< %2 wings feathers backbone beak bird >)*1

Code patterns:

ID43: (< %1 >)*1

ID48: (< %2 >)*1

CLEANED UP GRAMMAR GR4

Grammar GR4, G = 804.21, E = 20.76, score = 824.97:

Grammar patterns:

ID44: (< %1 fur backbone milk mammal >)*1

ID49: (< %2 wings feathers backbone beak bird >)*1

Code patterns:

ID43: (< %1 >)*1

ID48: (< %2 >)*1

TIDIED UP GR4

Grammar GR4, G = 804.21, E = 20.76, score = 824.97:

Grammar patterns:

ID44: (< %1 fur backbone milk mammal >)*1

ID49: (< %2 wings feathers backbone beak bird >)*1

Code patterns:

ID43: (< %1 >)*1

ID48: (< %2 >)*1

Comment:

These results suggest that brackets should not really have the status
IDENTIFICATION. In grammar 4 above and similar examples before that, they
serve no purpose in identifying patterns. They are simply boundary markers.

In Grammar 7, the two code patterns are: ID45: (< %3 2 >)*1 and ID55: (< %2 1 >)*1.
Here, the discrimination symbols are unnecessary and could be removed. There

seems to be a case for modifying the program so that it provides discrimination
symbols only when there are two or more alternatives in a given context.

Version 1.10 will make these adjustments.

%17 9/12/03

RESULTS FROM V 1.10

For reasons that are not entirely clear, version 2 of extract_patterns() is
leading to alignments like this:

```
0                    wings feathers      backbone   beak bird                0
                       |      |            |        |    |
1               < %4 wings feathers < %3    |     > beak bird >               1
                |                   | |     |     |           |
2               |                   < %3 backbone >           |               2
                |                     |     |                 |
3 < %5 <    fur <                     %3    |                 > milk mammal > 3
       |     |                              |                    |     |    |
4      < %1 fur                          backbone               milk mammal > 4


0   0

1   1

2   2

3 > 3

4   4
```

By contrast, version 1 of extract_patterns() produces 'correct' alignments like this:

```
0             wings feathers   backbone      beak bird     0
                |      |          |           |    |
1 < %5 7 < %3 wings feathers >    |     < %4 beak bird > > 1
  | |    | |                |    |     | |            | |
2 < %5   < %3               > backbone < %4           > > 2
```

Rather than trying to sort out the reasons for these differences, it is probably
better to develop version 3 of extract_patterns() along these lines:

* All coherent substrings, either matched or not matched, will be made into
separate patterns.

* There may be a cleaning up process when grammars are compiled that merges
patterns where one always references the other and the other is always
referenced by the first.

This will be done in v 2.0.

%18 11/12/03

RESULTS FROM SP71, V 2.0

With this input:

```
[
    [
        (fur backbone milk mammal)
        (wings feathers backbone beak bird)
    ]
    [
    ]
]
```

and with a cost factor of 100 (weighting the New symbols heavily),
the best two grammars found are:

ORIGINAL GRAMMAR GR7

Grammar GR7, Derived from GR3, G = 4147.73, E = 43.29, score = 4191.02:

Grammar patterns:

ID134: (< %5 10 milk mammal >)*1

ID135: (< %6 < %3 > < %4 > < %5 > >)*2

ID136: (< %3 8 fur >)*1

```
ID137: (< %4 backbone >)*2

ID148: (< %5 9 beak bird >)*1

ID150: (< %3 7 wings feathers >)*1

Code patterns:

ID133: (< %6 8 10 >)*1

ID147: (< %6 7 9 >)*1

CLEANED UP GRAMMAR GR7

Grammar GR7, G = 4147.73, E = 43.29, score = 4191.02:

Grammar patterns:

ID134: (< %5 10 milk mammal >)*1

ID135: (< %6 < %3 > < %4 > < %5 > >)*2

ID136: (< %3 8 fur >)*1

ID137: (< %4 backbone >)*2

ID148: (< %5 9 beak bird >)*1

ID150: (< %3 7 wings feathers >)*1

Code patterns:

ID133: (< %6 8 10 >)*1

ID147: (< %6 7 9 >)*1

TIDIED UP GR7

Grammar GR7, G = 4147.73, E = 43.29, score = 4191.02:

Grammar patterns:

ID134: (< %3 1 milk mammal >)*1

ID135: (< %4 < %1 > < %2 > < %3 > >)*2

ID136: (< %1 3 fur >)*1

ID137: (< %2 backbone >)*2

ID148: (< %3 4 beak bird >)*1

ID150: (< %1 2 wings feathers >)*1

Code patterns:

ID133: (< %4 3 1 >)*1

ID147: (< %4 2 4 >)*1

ORIGINAL GRAMMAR GR4

Grammar GR4, Derived from GR2, G = 4457.91, E = 20.78, score = 4478.68:

Grammar patterns:

ID132: (< %1 fur backbone milk mammal >)*1

ID139: (< %2 wings feathers backbone beak bird >)*1

Code patterns:

ID131: (< %1 >)*1

ID138: (< %2 >)*1

CLEANED UP GRAMMAR GR4

Grammar GR4, G = 4457.91, E = 20.78, score = 4478.68:

Grammar patterns:

ID132: (< %1 fur backbone milk mammal >)*1
```

```
ID139: (< %2 wings feathers backbone beak bird >)*1

Code patterns:

ID131: (< %1 >)*1

ID138: (< %2 >)*1

TIDIED UP GR4

Grammar GR4, G = 4457.91, E = 20.78, score = 4478.68:

Grammar patterns:

ID132: (< %1 fur backbone milk mammal >)*1

ID139: (< %2 wings feathers backbone beak bird >)*1

Code patterns:

ID131: (< %1 >)*1

ID138: (< %2 >)*1

[continued on 12/12/03]
```

These results look good but the program suffers from a problem: legal matches between brackets are enforced *after* many illegal matches have been recorded in the hit structure. This is already inefficient and has the potential to become excessively inefficient.

A possibly better idea would be to take advantage of the fact that only certain matches between brackets and class symbols are valid. These are the ones that were established when patterns were first created. For example, the initial and final brackets of '< %4 backbone >' should only be matched with the brackets around '%4' in the pattern '< %6 < %3 > < %4 > < %5 > >'.

This constraint can be enforced by using some kind of index of valid matches between symbols. An index is not required for symbols like '%4' because any match between two such symbols is necessarily valid. A complication in maintaining an index is that constraints need to be applied to matches between alignments of two or more patterns as well as between individual patterns. This may be handled by looking down columns to decide whether a match is valid as well as looking at symbols directly. Another complication is that the index would need to be edited or recreated whenever the set of Old patterns was purged.

An alternative idea is the constraint that each pair of 'internal' brackets (like those around '%4', above) can only be matched by 'external' brackets at the beginnings and ends of patterns. A snag with this idea is that it may lead to alignments like this:

```
0                    wings feathers      backbone   beak bird                0
                       |      |            |        |    |
1               < %4 wings feathers < %3    |     > beak bird >               1
                |                   | |     |     |           |
2               |                   < %3 backbone >           |               2
                |                     |     |                 |
3 < %5 <    fur <                     %3    |                 > milk mammal > 3
       |     |                              |                    |     |    |
4      < %1 fur                          backbone                milk mammal > 4
```

Here, the brackets around '%3' in row 3 have been matched to external brackets around the pattern in row 1 whereas the 'correct' matching of brackets and class symbols is the one shown in rows 1 and 2.

On balance, it looks as if the indexing idea is more promising. This will be tried in SP71, v 2.1. This seems to work: it gives the same results, is significantly faster and appears to eliminate all 'illegal' matchings of brackets.

%19 19/12/03

For this New pattern:

```
ID1 (j o h n r u n s)*1
```

the second best 'grammar' is:

```
TIDIED UP GR3

Grammar GR3, G = 16666.25, E = 20.96, score = 16687.22:
```

```
Grammar patterns:

ID116: (< %1 1 j o h n r u >)*1

ID117: (< %4 < %1 > < %2 > < %3 > >)*1

ID118: (< %2 n >)*1

ID119: (< %3 2 s >)*1

Code patterns:

ID115: (< %4 1 2 >)*1
```

Here, the 'discrimination' symbols '1' (in ID116) and '2' (in ID119) are not necessary because there is only one alternative for each of class '%1' and class '%3'. With these symbols removed, the grammar is like this:

```
Grammar patterns:

ID116: (< %1 j o h n r u >)*1

ID117: (< %4 < %1 > < %2 > < %3 > >)*1

ID118: (< %2 n >)*1

ID119: (< %3 s >)*1

Code patterns:

ID115: (< %4 >)*1
```

The function that cleans up unnecessary symbols in grammars should be able to remove these symbols.

With an example like this, the cleanup can go further. Any pattern that only ever appears in one context can be merged with that context. The result in this case would be:

```
Grammar patterns:

ID117: (< %4 j o h n r u n s >)*1

Code patterns:

ID115: (< %4 >)*1
```

%20 22/1/04

In SP71, v 2.1, the system for constraining matches between brackets to be 'legal' has the effect of forbidding some matches that are legal. For example,

```
0 m a r y       r u n s   0
                | | | |
1 < %1 5 j o h n r u n s > 1
```

Yields ID15 (< %7 15 m a r y >)*1 and ID13 (< %8 13 < %7 > < %4 > >)*2

but later:

```
0       j o h n w a l k s   0
        | | | |         |
1 < %1 5 j o h n r u n   s > 1
```

yields various patterns including ID212 (< %19 212 < %7 > < %18 > < %4 > >)*2

The system for checking legal matches fails to pick up the fact that '< %7 >' in ID212 is a valid match for the corresponding symbols in ID15 - because they were derived in different contexts.

In v 2.2, this system will be scrapped in favour of a system that records correspondences between left and right brackets in any one pattern. This should not suffer from the above problem. Correspondences of brackets will be checked in each hit sequence before it is converted into an alignment.

This issue has a bearing on the SP-neural ideas. If 'legal' connections are to be hardwired, this implies that new wiring would be needed not only between each pattern and the context from which it was derived but between each pattern and all the different contexts in which it may appear legally appear - which may include contexts from which it was not derived.

```
This solution turns out to be very inefficient because of the multiplicity of
alternative alignments resulting from the unconstrained matching of brackets.
Version 2.3 will be based on v 2.1 and will search for legal matches rather
than deriving them when patterns are extracted from hit sequences.

%21 23/1/04

RESULTS FROM SP71, V 2.3

With the input file:

[
    [
        (j o h n r u n s)
        (m a r y r u n s)
        (j o h n w a l k s)
        (m a r y w a l k s)
    ]
    [
    ]
]

the best grammar found is now:

ORIGINAL GRAMMAR GR72

Grammar GR72, Derived from GR43, G = 8766.46, E = 165.89, score = 8932.35:

Grammar patterns:

ID10 (< %4 10 s >)*4

ID15 (< %7 15 m a r y >)*2

ID16 (< %7 16 j o h n >)*2

ID218 (< %18 %16 218 w a l k >)*2

ID219 (< %19 219 < %7 > < %18 > < %4 > >)*4

ID223 (< %18 223 r u n >)*2

Code patterns:

ID1954 (< %19 219 16 223 10 >)*1

ID1969 (< %19 219 15 223 10 >)*1

ID1993 (< %19 219 16 %16 218 10 >)*1

ID2022 (< %19 219 15 %16 218 10 >)*1

CLEANED UP GRAMMAR GR72

Grammar GR72, G = 8766.46, E = 165.89, score = 8932.35:

Grammar patterns:

ID10 (< %4 10 s >)*4

ID15 (< %7 15 m a r y >)*2

ID16 (< %7 16 j o h n >)*2

ID218 (< %18 %16 218 w a l k >)*2

ID219 (< %19 219 < %7 > < %18 > < %4 > >)*4

ID223 (< %18 223 r u n >)*2

Code patterns:

ID1954 (< %19 219 16 223 10 >)*1

ID1969 (< %19 219 15 223 10 >)*1

ID1993 (< %19 219 16 %16 218 10 >)*1

ID2022 (< %19 219 15 %16 218 10 >)*1

TIDIED UP GR72
```

```
Grammar GR72, G = 8766.46, E = 165.89, score = 8932.35:

Grammar patterns:

ID10 (< %4 1 s >)*4

ID15 (< %5 2 m a r y >)*2

ID16 (< %5 3 j o h n >)*2

ID218 (< %2 %1 4 w a l k >)*2

ID219 (< %3 5 < %5 > < %2 > < %4 > >)*4

ID223 (< %2 6 r u n >)*2

Code patterns:

ID1954 (< %3 5 3 6 1 >)*1

ID1969 (< %3 5 2 6 1 >)*1

ID1993 (< %3 5 3 %1 4 1 >)*1

ID2022 (< %3 5 2 %1 4 1 >)*1

Here are some things that seem to need fixing:

1 There should only be a 'discrimination' symbol in a pattern if two
or more patterns can occur in the same context.

2 A pattern like ID10 (< %4 1 s >)*4 that only occurs in one context
may be merged with that context.

3 The distinction between 'class' symbols (that currently start with '%')
and 'discrimination' symbols should be dropped.

4 The cleaning up process should apply to code patterns just as much
as the main grammar. At present, the symbol '%16' is retained after
cleaning up although it is not necessary. This seems to be because
it appears in the code patterns.

%22 28/1/04

RESULTS FROM SP71, V 2.4

The best grammar found is:

ORIGINAL GRAMMAR GR72

Grammar GR72, Derived from GR43, G = 8563.41, E = 124.48, score = 8687.89:

Grammar patterns:

ID10 (< 7 6 s >)*4

ID15 (< 12 11 m a r y >)*2

ID16 (< 12 13 j o h n >)*2

ID229 (< 24 22 w a l k >)*2

ID230 (< 26 < 12 > < 24 > < 7 > >)*4

ID234 (< 24 25 r u n >)*2

Code patterns:

ID2366 (< 26 13 25 6 >)*1

ID2367 (< 26 11 25 6 >)*1

ID2368 (< 26 13 22 6 >)*1

ID2369 (< 26 11 22 6 >)*1

CLEANED UP GRAMMAR GR72

Grammar GR72, G = 8563.41, E = 124.48, score = 8687.89:

Grammar patterns:

ID10 (< 7 6 s >)*4
```

```
ID15 (< 12 11 m a r y >)*2

ID16 (< 12 13 j o h n >)*2

ID229 (< 24 22 w a l k >)*2

ID230 (< 26 < 12 > < 24 > < 7 > >)*4

ID234 (< 24 25 r u n >)*2

Code patterns:

ID2366 (< 26 13 25 6 >)*1

ID2367 (< 26 11 25 6 >)*1

ID2368 (< 26 13 22 6 >)*1

ID2369 (< 26 11 22 6 >)*1

GRAMMAR GR72 AFTER MERGING OF PATTERNS

Grammar GR72, G = 8563.41, E = 124.48, score = 8687.89:

Grammar patterns:

ID15 (< 12 11 m a r y >)*2

ID16 (< 12 13 j o h n >)*2

ID229 (< 24 22 w a l k >)*2

ID234 (< 24 25 r u n >)*2

ID2406 (< 26 < 12 > < 24 > s >)*4

Code patterns:

ID2366 (< 26 13 25 >)*1

ID2367 (< 26 11 25 >)*1

ID2368 (< 26 13 22 >)*1

ID2369 (< 26 11 22 >)*1

TIDIED UP GR72

Grammar GR72, G = 8563.41, E = 124.48, score = 8687.89:

Grammar patterns:

ID15 (< 2 1 m a r y >)*2

ID16 (< 2 3 j o h n >)*2

ID229 (< 5 4 w a l k >)*2

ID234 (< 5 6 r u n >)*2

ID2406 (< 7 < 2 > < 5 > s >)*4

Code patterns:

ID2366 (< 7 3 6 >)*1

ID2367 (< 7 1 6 >)*1

ID2368 (< 7 3 4 >)*1

ID2369 (< 7 1 4 >)*1

This looks OK but the second best grammar (before cleaning up) is:

ORIGINAL GRAMMAR GR63

Grammar GR63, Derived from GR32, G = 8982.57, E = 89.80, score = 9072.37:

Grammar patterns:

ID11 (< 7 8 r u n s >)*2
```

```
ID13 (< 14 < 12 > < 7 > >)*2

ID15 (< 2 1 m a r y >)*2

ID16 (< 2 3 j o h n >)*2

ID689 (< 29 < 12 > < 28 > >)*2

ID690 (< 28 w a l k s >)*2

Code patterns:

ID2330 (< 14 13 8 >)*1

ID2331 (< 14 11 8 >)*1

ID2332 (< 29 13 >)*1

ID2333 (< 29 11 >)*1

Apart from some anomalies in the code patterns, the main problem here
is that the system has failed to recognise that 'r u n s' and 'w a l k s'
belong in the same class. Also, the class symbols for 'j o h n' and
'm a r y' have been changed from 12 to 2 by the cleaning up process
applied to the first grammar. This means that each grammar should
have its own copies of grammar patterns.

The main problem seems to arise from the way that the patterns and
classes are isolated from hit sequences:

HIT SEQUENCE #14:

0 m a r y     r u n s   0
              | | | |
1 < 1 j o h n r u n s > 1

Non-hit pattern ID15 (< 12 11 m a r y >)*1

< (0, 4.00, LB, ID), 12 (1, 8.00, CCS, ID),
11 (2, 8.00, CCS, ID), m (3, 408.75, DATA, CNT),
a (4, 308.75, DATA, CNT), r (5, 308.75, DATA, CNT),
y (6, 408.75, DATA, CNT), > (7, 4.00, RB, ID), frequency = 1.

Non-hit pattern ID16 (< 12 13 j o h n >)*1

< (0, 4.00, LB, ID), 12 (1, 8.00, CCS, ID),
13 (2, 8.00, CCS, ID), j (3, 408.75, DATA, CNT),
o (4, 408.75, DATA, CNT), h (5, 408.75, DATA, CNT),
n (6, 308.75, DATA, CNT), > (7, 4.00, RB, ID), frequency = 1.

Contents-symbol match found between ID14 (r u n s)*1 and ID11 (< 7 8 r u n s >)*1

Unified pattern ID11 (< 7 8 r u n s >)*1

< (0, 4.00, LB, ID), 7 (1, 8.00, CCS, ID),
8 (2, 8.00, CCS, ID), r (3, 308.75, DATA, CNT),
u (4, 408.75, DATA, CNT), n (5, 308.75, DATA, CNT),
s (6, 308.75, DATA, CNT), > (7, 4.00, RB, ID), frequency = 1.

Abstract pattern ID13 (< 14 < 12 > < 7 > >)*2

< (0, 4.00, LB, ID), 14 (1, 8.00, CCS, ID),
< (2, 4.00, LB, CNT), 12 (3, 8.00, CCS, CNT),
> (4, 4.00, RB, CNT), < (5, 4.00, LB, CNT),
7 (6, 8.00, CCS, CNT), > (7, 4.00, RB, CNT),
> (8, 4.00, RB, ID), frequency = 2.

and then:

HIT SEQUENCE #2007:

0 m a r y     w a l k s   0
              | | | | |
1 < 20 j o h n w a l k s > 1

Contents-symbol match found between ID691 (m a r y)*1 and ID15 (< 12 11 m a r y >)*1

Contents-symbol match found between ID692 (j o h n)*1 and ID16 (< 12 13 j o h n >)*3

Non-hit pattern ID15 (< 12 11 m a r y >)*2

< (0, 4.00, LB, ID), 12 (1, 8.00, CCS, ID),
11 (2, 8.00, CCS, ID), m (3, 408.75, DATA, CNT),
```

```
a (4, 308.75, DATA, CNT), r (5, 308.75, DATA, CNT),
y (6, 408.75, DATA, CNT), > (7, 4.00, RB, ID), frequency = 2.

Non-hit pattern ID16 (< 12 13 j o h n >)*4

< (0, 4.00, LB, ID), 12 (1, 8.00, CCS, ID),
13 (2, 8.00, CCS, ID), j (3, 408.75, DATA, CNT),
o (4, 408.75, DATA, CNT), h (5, 408.75, DATA, CNT),
n (6, 308.75, DATA, CNT), > (7, 4.00, RB, ID), frequency = 4.

Unified pattern ID690 (< 28 w a l k s >)*1

< (0, 4.00, LB, ID), 28 (1, 8.00, CCS, ID),
w (2, 408.75, DATA, CNT), a (3, 308.75, DATA, CNT),
l (4, 408.75, DATA, CNT), k (5, 408.75, DATA, CNT),
s (6, 308.75, DATA, CNT), > (7, 4.00, RB, ID), frequency = 1.

Abstract pattern ID689 (< 29 < 12 > < 28 > >)*2

< (0, 4.00, LB, ID), 29 (1, 8.00, CCS, ID),
< (2, 4.00, LB, CNT), 12 (3, 8.00, CCS, CNT),
> (4, 4.00, RB, CNT), < (5, 4.00, LB, CNT),
28 (6, 8.00, CCS, CNT), > (7, 4.00, RB, CNT),
> (8, 4.00, RB, ID), frequency = 2.

Question: the program seems not to have found the alignment:

0    j o h n w a l k s 0
     | | | |
1 < 1 j o h n r u n s > 1

If it had, then 'w a l k s' and 'r u n s' might have been put in the
same class. Likewise for the alignment:

0     m a r y w a l k s 0
      | | | |
1 < 10 m a r y r u n s > 1

The reason these have not been formed appears to be because the
'extraction_limit' was set to 2 and other hit sequences had higher scores
than the ones just shown. Here are examples of other hit sequences with
higher scores:

HIT SEQUENCE #618:

0    j o h      n w a l k s   0
     | | |      |         |
1 < 1 j o h n r u n        s > 1

and

HIT SEQUENCE #617:

0    j o h n w a l k s   0
     | | | |         |
1 < 1 j o h n r u n   s > 1

Before checking these things out, the program will be modified so that each
grammar has its own copies of grammar patterns. This should eliminate
anomalies arising from the cleaning up process.

These changes have been made but the program still fails to find grammars in
which 'r u n s' and 'w a l k s' are assigned to the same class. The reason
seems to be that alignments like this:

HIT SEQUENCE #617:

0    j o h n w a l k s   0
     | | | |         |
1 < 1 j o h n r u n   s > 1

and

HIT SEQUENCE #3147:

0     m a r y w a l k s   0
      | | | |         |
1 < 10 m a r y r u n   s > 1

are always better than alignments in which 'r u n s' and 'w a l k s' are
unmatched sequences and so the latter kinds of hit sequences are never formed.

%23 28/1/04
```

FURTHER RESULTS FROM SP71, V 2.4

After modification as described in the previous section, the best grammar found is:

ORIGINAL GRAMMAR GR114

Grammar GR114, Derived from GR85, G = 9132.57, E = 166.53, score = 9299.09:

Grammar patterns:

ID7061 (< 20 12 13 j o h n >)*2

ID7062 (< 37 29 r u n >)*2

ID7063 (< 31 7 6 s >)*4

ID7064 (< 57 < 12 > < 37 > < 31 > >)*4

ID7066 (< 12 11 m a r y >)*2

ID7068 (< 37 35 w a l k >)*2

Code patterns:

ID7065 (< 57 20 13 29 7 6 >)*1

ID7067 (< 57 11 29 7 6 >)*1

ID7069 (< 57 20 13 35 7 6 >)*1

ID7070 (< 57 11 35 7 6 >)*1

CLEANED UP GRAMMAR GR114

Grammar GR114, G = 9132.57, E = 166.53, score = 9299.09:

Grammar patterns:

ID7061 (< 20 12 13 j o h n >)*2

ID7062 (< 37 29 r u n >)*2

ID7063 (< 31 7 6 s >)*4

ID7064 (< 57 < 12 > < 37 > < 31 > >)*4

ID7066 (< 12 11 m a r y >)*2

ID7068 (< 37 35 w a l k >)*2

Code patterns:

ID7065 (< 57 20 13 29 7 6 >)*1

ID7067 (< 57 11 29 7 6 >)*1

ID7069 (< 57 20 13 35 7 6 >)*1

ID7070 (< 57 11 35 7 6 >)*1

The problem here is that the cleaning up process does not remove class
symbols 20 and 13 because these appear in the code patterns. Only one
of them is needed, not both. Likewise, for the symbols 7 and 6.

What is needed is a cleaning-up process that can recognise when there
are more 'discrimination' symbols than are necessary for unique
identification of a pattern amongst those in the main grammar.

%24 28/1/04

RESULTS FROM SP71, V 2.5

With the new cleaning up process, the best grammar found is:

ORIGINAL GRAMMAR GR114

Grammar GR114, derived from GR85, G = 9132.57, E = 166.53, score = 9299.09:

Grammar patterns:

ID7355 (< 20 12 13 j o h n >)*2

```
ID7356 (< 37 29 r u n >)*2

ID7357 (< 31 7 6 s >)*4

ID7358 (< 57 < 12 > < 37 > < 31 > >)*4

ID7360 (< 12 11 m a r y >)*2

ID7362 (< 37 35 w a l k >)*2

Code patterns:

ID7359 (< 57 20 13 29 7 6 >)*1

ID7361 (< 57 11 29 7 6 >)*1

ID7363 (< 57 20 13 35 7 6 >)*1

ID7364 (< 57 11 35 7 6 >)*1

CLEANED UP GRAMMAR GR114

Grammar GR114, G = 9076.50, E = 166.53, score = 9243.03:

Grammar patterns:

ID7355 (< 20 12 j o h n >)*2

ID7356 (< 37 29 r u n >)*2

ID7357 (< 31 7 s >)*4

ID7358 (< 57 < 12 > < 37 > < 31 > >)*4

ID7360 (< 12 11 m a r y >)*2

ID7362 (< 37 35 w a l k >)*2

Code patterns:

ID7359 (< 57 20 29 7 >)*1

ID7361 (< 57 11 29 7 >)*1

ID7363 (< 57 20 35 7 >)*1

ID7364 (< 57 11 35 7 >)*1

GRAMMAR GR114 AFTER MERGING OF PATTERNS

Grammar GR114, G = 9030.36, E = 166.53, score = 9196.89:

Grammar patterns:

ID7355 (< 20 12 j o h n >)*2

ID7356 (< 37 29 r u n >)*2

ID7360 (< 12 11 m a r y >)*2

ID7362 (< 37 35 w a l k >)*2

ID7502 (< 57 < 12 > < 37 > s >)*4

Code patterns:

ID7359 (< 57 20 29 >)*1

ID7361 (< 57 11 29 >)*1

ID7363 (< 57 20 35 >)*1

ID7364 (< 57 11 35 >)*1

TIDIED UP GR114

Grammar GR114, G = 9030.36, E = 166.53, score = 9196.89:

Grammar patterns:

ID7355 (< 3 2 j o h n >)*2

ID7356 (< 6 4 r u n >)*2
```

```
ID7360 (< 2 1 m a r y >)*2

ID7362 (< 6 5 w a l k >)*2

ID7502 (< 7 < 2 > < 6 > s >)*4

Code patterns:

ID7359 (< 7 3 4 >)*1

ID7361 (< 7 1 4 >)*1

ID7363 (< 7 3 5 >)*1

ID7364 (< 7 1 5 >)*1

This looks to be 'correct'.

The second-best grammar is this:

ORIGINAL GRAMMAR GR115

Grammar GR115, derived from GR78, G = 9173.71, E = 165.56, score = 9339.26:

Grammar patterns:

ID7369 (< 20 12 13 j o h n >)*2

ID7370 (< 37 29 r u n >)*2

ID7371 (< 31 7 6 s >)*4

ID7372 (< 38 < 20 > < 37 > < 31 > >)*2

ID7374 (< 12 11 m a r y >)*2

ID7375 (< 57 < 12 > < 37 > < 31 > >)*4

ID7377 (< 37 35 w a l k >)*2

Code patterns:

ID7373 (< 38 12 13 29 7 6 >)*1

ID7376 (< 57 11 29 7 6 >)*1

ID7378 (< 38 12 13 35 7 6 >)*1

ID7379 (< 57 11 35 7 6 >)*1

CLEANED UP GRAMMAR GR115

Grammar GR115, G = 9140.54, E = 165.56, score = 9306.10:

Grammar patterns:

ID7369 (< 20 12 13 j o h n >)*2

ID7370 (< 37 29 r u n >)*2

ID7371 (< 31 7 s >)*4

ID7372 (< 38 < 20 > < 37 > < 31 > >)*2

ID7374 (< 12 11 m a r y >)*2

ID7375 (< 57 < 12 > < 37 > < 31 > >)*4

ID7377 (< 37 35 w a l k >)*2

Code patterns:

ID7373 (< 38 12 13 29 7 >)*1

ID7376 (< 57 11 29 7 >)*1

ID7378 (< 38 12 13 35 7 >)*1

ID7379 (< 57 11 35 7 >)*1

GRAMMAR GR115 AFTER MERGING OF PATTERNS
```

```
Grammar GR115, G = 9045.09, E = 165.56, score = 9210.65:

Grammar patterns:

ID7370 (< 37 29 r u n >)*2

ID7374 (< 12 11 m a r y >)*2

ID7377 (< 37 35 w a l k >)*2

ID7503 (< 38 j o h n < 37 > < 31 > >)*2

ID7504 (< 57 < 12 > < 37 > s >)*4

Code patterns:

ID7373 (< 38 12 29 >)*1

ID7376 (< 57 11 29 >)*1

ID7378 (< 38 12 35 >)*1

ID7379 (< 57 11 35 >)*1

TIDIED UP GR115

Grammar GR115, G = 9045.09, E = 165.56, score = 9210.65:

Grammar patterns:

ID7370 (< 6 3 r u n >)*2

ID7374 (< 2 1 m a r y >)*2

ID7377 (< 6 5 w a l k >)*2

ID7503 (< 7 j o h n < 6 > < 4 > >)*2

ID7504 (< 8 < 2 > < 6 > s >)*4

Code patterns:

ID7373 (< 7 2 3 >)*1

ID7376 (< 8 1 3 >)*1

ID7378 (< 7 2 5 >)*1

ID7379 (< 8 1 5 >)*1

The original grammar appears to be legal as is the grammar after cleaning
up but anomalies have crept in when patterns are merged.
This needs looking at.

One anomaly is that < 31 > has been replaced by 's' in ID7504 but not
in ID7503. The single_referent() method seems to be at fault and
there may be double counting of patterns as well.

%25 29/1/04

MORE ABOUT GR115

These two patterns:

ID7372 (< 38 < 20 > < 37 > < 31 > >)*2

and

ID7375 (< 57 < 12 > < 37 > < 31 > >)*4

represent *alternative* analyses of the raw sentences. This kind of
thing is acceptable in a grammar but it calls in question one or two
assumptions that are made elsewhere in the model:

When looking for patterns that can be merged, it is not safe to
assume that, if a given reference has only one referent and if the
frequencies of the refering pattern and the referent pattern are
the same, then they can be merged. The referent pattern may also
be referenced from elsewhere in the grammar and, in that case,
it needs to be kept as a separate entity.

The grammar::merge_patterns() method will be modified to take
account of this.
```

%26 30/1/04

After the modification described at the end of the last section, GR115 is:

ORIGINAL GRAMMAR GR115

Grammar GR115, derived from GR78, G = 9173.71, E = 165.56, score = 9339.26:

Grammar patterns:

ID7369 (< 20 12 13 j o h n >)*2

ID7370 (< 37 29 r u n >)*2

ID7371 (< 31 7 6 s >)*4

ID7372 (< 38 < 20 > < 37 > < 31 > >)*2

ID7374 (< 12 11 m a r y >)*2

ID7375 (< 57 < 12 > < 37 > < 31 > >)*4

ID7377 (< 37 35 w a l k >)*2

Code patterns:

ID7373 (< 38 12 13 29 7 6 >)*1

ID7376 (< 57 11 29 7 6 >)*1

ID7378 (< 38 12 13 35 7 6 >)*1

ID7379 (< 57 11 35 7 6 >)*1

CLEANED UP GRAMMAR GR115

Grammar GR115, G = 9140.54, E = 165.56, score = 9306.10:

Grammar patterns:

ID7369 (< 20 12 13 j o h n >)*2

ID7370 (< 37 29 r u n >)*2

ID7371 (< 31 7 s >)*4

ID7372 (< 38 < 20 > < 37 > < 31 > >)*2

ID7374 (< 12 11 m a r y >)*2

ID7375 (< 57 < 12 > < 37 > < 31 > >)*4

ID7377 (< 37 35 w a l k >)*2

Code patterns:

ID7373 (< 38 12 13 29 7 >)*1

ID7376 (< 57 11 29 7 >)*1

ID7378 (< 38 12 13 35 7 >)*1

ID7379 (< 57 11 35 7 >)*1

GRAMMAR GR115 AFTER MERGING OF PATTERNS

Grammar GR115, G = 9091.23, E = 165.56, score = 9256.79:

Grammar patterns:

ID7370 (< 37 29 r u n >)*2

ID7371 (< 31 7 s >)*4

ID7374 (< 12 11 m a r y >)*2

ID7375 (< 57 < 12 > < 37 > < 31 > >)*4

ID7377 (< 37 35 w a l k >)*2

ID7503 (< 38 j o h n < 37 > < 31 > >)*2

Code patterns:

ID7373 (< 38 12 29 7 >)*1

ID7376 (< 57 11 29 7 >)*1

ID7378 (< 38 12 35 7 >)*1

ID7379 (< 57 11 35 7 >)*1

TIDIED UP GR115

Grammar GR115, G = 9091.23, E = 165.56, score = 9256.79:

Grammar patterns:

ID7370 (< 6 3 r u n >)*2

ID7371 (< 4 9 s >)*4

ID7374 (< 2 1 m a r y >)*2

ID7375 (< 8 < 2 > < 6 > < 4 > >)*4

ID7377 (< 6 5 w a l k >)*2

ID7503 (< 7 j o h n < 6 > < 4 > >)*2

Code patterns:

ID7373 (< 7 2 3 9 >)*1

ID7376 (< 8 1 3 9 >)*1

ID7378 (< 7 2 5 9 >)*1

ID7379 (< 8 1 5 9 >)*1

Comments:

* In ID7369 (< 20 12 13 j o h n >)*2, symbol '13' is retained in the
cleaned up version because of the rule that there should be one unreferenced
class symbol to serve as a 'discrimination' symbol. A better rule might be
that there should be one more symbol than is needed to differentiate the
pattern from other patterns in the same class. In this case, the classes
20 and 12 would do. This may be something to try to tidy up but it is
probably not top priority.

* The merging of 'j o h n' into ID7503 to make '< 38 j o h n < 37 > < 31 > >'
means that '< 12 >' in ID7375 (< 57 < 12 > < 37 > < 31 > >)*4 now only
references 'm a r y'. So there needs to be another round of the merging
process to check on possibilities like this and to merge them as well.
This will be done.

Now GR115 is:

ORIGINAL GRAMMAR GR115

Grammar GR115, derived from GR78, G = 9173.71, E = 165.56, score = 9339.26:

Grammar patterns:

ID7369 (< 20 12 13 j o h n >)*2

ID7370 (< 37 29 r u n >)*2

ID7371 (< 31 7 6 s >)*4

ID7372 (< 38 < 20 > < 37 > < 31 > >)*2

ID7374 (< 12 11 m a r y >)*2

ID7375 (< 57 < 12 > < 37 > < 31 > >)*4

ID7377 (< 37 35 w a l k >)*2

Code patterns:

ID7373 (< 38 12 13 29 7 6 >)*1

ID7376 (< 57 11 29 7 6 >)*1

ID7378 (< 38 12 13 35 7 6 >)*1

```
ID7379 (< 57 11 35 7 6 >)*1

CLEANED UP GRAMMAR GR115

Grammar GR115, G = 9140.54, E = 165.56, score = 9306.10:

Grammar patterns:

ID7369 (< 20 12 13 j o h n >)*2

ID7370 (< 37 29 r u n >)*2

ID7371 (< 31 7 s >)*4

ID7372 (< 38 < 20 > < 37 > < 31 > >)*2

ID7374 (< 12 11 m a r y >)*2

ID7375 (< 57 < 12 > < 37 > < 31 > >)*4

ID7377 (< 37 35 w a l k >)*2

Code patterns:

ID7373 (< 38 12 13 29 7 >)*1

ID7376 (< 57 11 29 7 >)*1

ID7378 (< 38 12 13 35 7 >)*1

ID7379 (< 57 11 35 7 >)*1

GRAMMAR GR115 AFTER MERGING OF PATTERNS

Grammar GR115, G = 9040.07, E = 165.56, score = 9205.63:

Grammar patterns:

ID7370 (< 37 29 r u n >)*2

ID7371 (< 31 7 s >)*4

ID7377 (< 37 35 w a l k >)*2

ID7503 (< 38 j o h n < 37 > < 31 > >)*2

ID7504 (< 57 m a r y < 37 > < 31 > >)*2

Code patterns:

ID7373 (< 38 29 7 >)*1

ID7376 (< 57 29 7 >)*1

ID7378 (< 38 35 7 >)*1

ID7379 (< 57 35 7 >)*1

TIDIED UP GR115

Grammar GR115, G = 9040.07, E = 165.56, score = 9205.63:

Grammar patterns:

ID7370 (< 4 1 r u n >)*2

ID7371 (< 2 7 s >)*4

ID7377 (< 4 3 w a l k >)*2

ID7503 (< 5 j o h n < 4 > < 2 > >)*2

ID7504 (< 6 m a r y < 4 > < 2 > >)*2

Code patterns:

ID7373 (< 5 1 7 >)*1

ID7376 (< 6 1 7 >)*1

ID7378 (< 5 3 7 >)*1
```

```
ID7379 (< 6 3 7 >)*1

GR116 comes out the same after cleaning, merging of patterns and tidying up.

%27 12/5/04

RESULTS FROM SP71, V 2.5

With this input:

[
    [
        (j o h n r u n s)
        (m a r y r u n s)
        (j o h n w a l k s)
        (m a r y w a l k s)
    ]
    [
    ]
]

the best cleaned up grammar is:

CLEANED UP GRAMMAR GR114

Grammar GR114, G = 9076.50, E = 166.53, score = 9243.03:

Grammar patterns:

ID7355 (< 20 12 j o h n >)*2

ID7356 (< 37 29 r u n >)*2

ID7357 (< 31 7 s >)*4

ID7358 (< 57 < 12 > < 37 > < 31 > >)*4

ID7360 (< 12 11 m a r y >)*2

ID7362 (< 37 35 w a l k >)*2

Code patterns:

ID7359 (< 57 20 29 7 >)*1

ID7361 (< 57 11 29 7 >)*1

ID7363 (< 57 20 35 7 >)*1

ID7364 (< 57 11 35 7 >)*1

and, after merging of patterns and tidying up, the result is:

TIDIED UP GR114

Grammar GR114, G = 9030.36, E = 166.53, score = 9196.89:

Grammar patterns:

ID7355 (< 3 2 j o h n >)*2

ID7356 (< 6 4 r u n >)*2

ID7360 (< 2 1 m a r y >)*2

ID7362 (< 6 5 w a l k >)*2

ID7502 (< 7 < 2 > < 6 > s >)*4

Code patterns:

ID7359 (< 7 3 4 >)*1

ID7361 (< 7 1 4 >)*1

ID7363 (< 7 3 5 >)*1

ID7364 (< 7 1 5 >)*1

With this input:

[
    [
```

```
        (t h a t b o y r u n s)
        (t h a t g i r l r u n s)
        (t h a t b o y w a l k s)
        (t h a t g i r l w a l k s)
        (s o m e b o y r u n s)
        (s o m e g i r l r u n s)
        (s o m e b o y w a l k s)
        (s o m e g i r l w a l k s)
        (t h a t b o y r u n s)
        (t h a t g i r l r u n s)
        (t h a t b o y w a l k s)
        (t h a t g i r l w a l k s)
        (s o m e b o y r u n s)
        (s o m e g i r l r u n s)
        (s o m e b o y w a l k s)
        (s o m e g i r l w a l k s)
    ]
    [
    ]
]

the best cleaned up grammar is:

CLEANED UP GRAMMAR GR1104

Grammar GR1104, G = 17638.28, E = 821.64, score = 18459.91:

Grammar patterns:

ID74128 (< 13 r u n s >)*8

ID74129 (< 32 11 b o y >)*8

ID74130 (< 81 9 t h a t >)*8

ID74131 (< 14 < 9 > < 11 > < 13 > >)*4

ID74133 (< 24 11 g i r l >)*8

ID74135 (< 60 w a l k s >)*8

ID74136 (< 156 < 81 > < 11 > < 60 > >)*8

ID74139 (< 126 83 81 s o m e >)*8

ID74140 (< 106 < 83 > < 11 > < 13 > >)*4

ID74143 (< 100 98 s o m e b o y >)*4

ID74144 (< 125 < 98 > < 60 > >)*4

Code patterns:

ID74132 (< 14 81 32 >)*1

ID74134 (< 14 81 24 >)*1

ID74137 (< 156 9 32 >)*1

ID74138 (< 156 9 24 >)*1

ID74141 (< 106 126 81 32 >)*1

ID74142 (< 106 126 81 24 >)*1

ID74145 (< 125 100 >)*1

ID74146 (< 156 126 83 24 >)*1

These results are not exactly right but it seems worth proceeding to
the next stage of development and seeing whether current problems
might 'come out in the wash'.

For version 3.0, it seems worth trying these features:

1 Any kind of mismatch or incomplete matching of a pattern during parsing
should lead to the creation of new coded patterns to be added to Old.

2 After each pattern from new is parsed, with the creation of new patterns
for Old, all the New patterns up to and including the current New pattern
are reparsed, sifting and sorting is applied, and all Old patterns other
than those in the best one or two grammars are discarded. This should
help to reduce the size of Old and speed up later processing. It also
```

enables one to get an idea of how the grammars are developing.

%28 24/5/04

RESULTS FROM VERSION 3.0

In this version, the program re-parses all the New patterns up to including
the 'current' New pattern, selects 'full' parses and makes grammars
immediately after each New pattern has been processed to find new patterns
to add to Old. Then it discards all the Old patterns formed up to that point
and replaces them with the patterns in the best one or two grammars that
have been formed.

With this input:

```
[
    [
        (t h a t b o y r u n s)
        (t h a t g i r l r u n s)
        (t h a t b o y w a l k s)
        (t h a t g i r l w a l k s)
    ]
    [
    ]
]
```

the program forms these full parses after the third pattern has been processed:

ID611: (< 1 t h a t b o y r u n s >)*1

ID630: (< 30 < 25 t h a t b o y > < 36 27 28 r u n > < 29 s > >)*1

ID635: (< 14 < 9 t h a t > < 22 16 11 12 b o y > < 13 r u n s > >)*1

ID648: (< 8 t h a t g i r l r u n s >)*1

ID681: (< 14 < 9 t h a t > < 11 10 g i r l > < 13 r u n s > >)*1

ID712: (< 24 t h a t b o y w a l k s >)*1

ID771: (< 30 < 25 t h a t b o y > < 27 26 w a l k > < 29 s > >)*1

A nice feature of this version is that the initial 'bad' analysis of the
first pattern is quickly lost because the corresponding 'bad' patterns
in Old have been discarded.

The main problem now is that the program does not recognise that
'< 16 t h a t b o y >' can be analysed into '< 9 t h a t >' and
'< 11 12 b o y >'. Given that recognition and a bit of tidying up,
the results at this stage would be essentially perfect.

Processing the fourth pattern does not help. The full parses at that stage are:

ID1170: (< 30 < 25 t h a t b o y > < 47 36 27 28 r u n > < 29 s > >)*1

ID1174: (< 14 < 9 t h a t > < 22 16 11 12 b o y > < 13 r u n s > >)*1

ID1239: (< 8 t h a t g i r l r u n s >)*1

ID1256: (< 50 < 49 t h a t g i r l > < 47 36 27 28 r u n > < 29 s > >)*1

ID1257: (< 14 < 9 t h a t > < 11 10 g i r l > < 13 r u n s > >)*1

ID1274: (< 24 t h a t b o y w a l k s >)*1

ID1291: (< 30 < 25 t h a t b o y > < 27 26 w a l k > < 29 s > >)*1

ID1292: (< 52 < 9 t h a t > < 22 16 11 12 b o y > < 51 w a l k s > >)*1

ID1309: (< 42 t h a t g i r l w a l k s >)*1

ID1329: (< 50 < 49 t h a t g i r l > < 27 26 w a l k > < 29 s > >)*1

ID1330: (< 52 < 9 t h a t > < 11 10 g i r l > < 51 w a l k s > >)*1

In general, the problem is that if 'r u n s' and 'w a l k s' are analysed
into a stem part and a terminal 's', then the first two words in the
sentence are always joined into one block. If the first two words are
correctly split, then the final word is not analysed into its constituent parts.

Here are two tentative possible solutions:

1 In Phase 1, before any pattern is added to Old, it should be analysed as

if it were New and, if it can be parsed into something smaller, this analysis
should be added to the list of Old patterns.

2 Probably a better idea is, during phase 1, to add to Old all the 'full'
encodings of the New pattern that are recognised during parsing. This should
automatically identify analyses that break up chunks like
'< 49 t h a t g i r l >'. For example, 't h a t g i r l w a l k s'
would probably be parsed as '< 9 t h a t > < 11 10 g i r l > < 27 26 w a l k >
< 29 s >' and this would lead to the creation of a new code pattern
something like this:

'< < 9 > < 11 10 > < 27 26 > < 29 > >'

This could be generalised by leaving out '10' and '26' or the generalisation
could be achieved by partial matching between this pattern and '< < 9 >
< 22 16 11 12 > < 47 36 27 28 > < 29 > >'

This second idea will be tried in version 3.01. But first, each new version of
Old will be cleaned up, with merging of patterns and renumbering of code symbols.

%29 25/5/04

INTERMEDIATE STRUCTURE

The second idea described in %28 will not work as originally conceived because
SP71 does not form alignments like this:

```
0    t h a t               b o y r u n s 0
     | | | |               | | |
1    | | | |   < 22 16 11 12 b o y >      1
     | | | |
2 < 9 t h a t >                           2
```

so it is not possible to make intermediate stucture by simply contatenating
the relevant substructures.

Alignments like the one shown are implicit because the program does form the
constituent alignments and they are implicitly concatenated by the sequencing
of the symbols in New. But these alignments are staging posts on the way to
bigger alignments that may very well be 'correct'. It seems premature to
start creating new structures when the final 'best' alignment is still not known.

An alternative possibility is to form intermediate structures from
alignments like this:

```
0        t h a t        b o y   w a l k s 0
         | | | |        | | |
1    < 8 t h a t >      | | |           1
     | |        |      | | |
2    | |        | < 6 3 b o y >         2
     | |        | |   |      |
3 < 5 < 8        > <   3      > < 4 > >  3
```

The general idea would be to let parsing proceed as far as it can and *then*
look for incompletely-matched patterns. New structures can be derived from
these incompletely-matched patterns.

The difficulty here is that it is not clear what to do with alignments
like these:

```
0        t h a      t      g i r l r u n s    0
         | | |      |             | | | |
1        | | |      |   < 5 4 b o y r u n s >  1
         | | |      |   | |               |
2    < 2 t h a >    |   | |               |   2
     | |      |     |   | |               |
3 < 6 < 2      > < 3 | > < 5               > > 3
               | | | |
4              < 3 t >                       4

0        t h a      t      g i    r l r u n s    0
         | | |      |             |    | | |
1        | | |      |   < 5 4 b o y r    u n s >  1
         | | |      |   | |                 |
2    < 2 t h a >    |   | |                 |   2
     | |      |     |   | |                 |
3 < 6 < 2      > < 3 | > < 5                 > > 3
               | | | |
4              < 3 t >                        4

0        t h a  t g i    r l    r u n s    0
         | | |                | | | |
1        | | |       < 5 4 b o y r u n s >  1
```

```
        | | |       | |              |
2    < 2 t h a >      | |              |   2
     | |      |      | |              |
3 < 6 < 2      > < 3 > < 5              > > 3

0        t h a      t      g i r l r u n s    0
         | | |                    | | | |
1        | | |          < 5 4 b o y r u n s >   1
         | | |          | |               |
2    < 2 t h a >        | |               |   2
     | |      |        | |               |
3 < 6 < 2      > < 3   > < 5               > > 3
                 | |   |
4                < 3 t >                      4
```

With the last one in particular, it is not clear whether 't g i r l' in New is an alternative to 'b o y' in '< 5 4 b o y r u n s >' or to every part of an Old pattern between 't h a' and 'r u n s'.

Part of the answer may be that the alignment should not have been allowed to get this far. When this alignment was formed:

```
0 t h a t g i r l r u n s   0
                | | | |
1 < 5 4 b o y     r u n s > 1
```

then learning should have been invoked because the Old pattern was incompletely matched. In short, no alignment should be allowed to proceed to a later cycle if its C-symbols are incompletely matched.

This approach to learning is sufficiently different from previous approaches to justify a new version number: 4.0.

Here is a tentative rule for version 4.0: allow parsing to proceed as normal but, with any alignment in which the Old C-symbols are incompletely matched, use it for learning and do not allow the alignment to proceed to further cycles in the parsing process.

It looks as if this should only apply to 'data' symbols within Old patterns, otherwise the system would not be able to build mult-level parsings with the current parsing strategy. This might change if hit structures could record hits with two or or more driving patterns and/or target patterns.

Another tentative rule is that new patterns should be created from matched symbols in the target pattern and residual symbols in the target pattern but not from the New pattern.

%30 26/5/04

MORE THOUGHTS ON LEARNING PROCESSES

Some tentative rules:

1 Add the New pattern to Old as it is (with added ID-symbols).

2 If a New pattern cannot be parsed, but parts of it form partial matches with one or more Old patterns, create new patterns from the matching parts, each with ID-symbols, and add these patterns to Old. Also, add the New pattern to Old with encodings in the parts corresponding to the newly-created unified parts.

3 If a New pattern cannot be 'fully' parsed, but parts of it can be parsed in terms of patterns in Old, encode those parts that can be fully parsed and add the whole pattern to Old, with encodings in the parts that can be parsed.

4 If a New pattern can be 'fully' parsed, add its encoding to Old (with added ID-symbols).

This may give us a handle on the formation of intermediate-level structures and also the formation of discontinuous dependencies.

These rules will be implemented in SP71, v 4.0 (instead of what was proposed in %29).

%31 2/6/04

MORE THOUGHTS

1 Given an alignment like this:

```
t h e r e d     a p p l e
```

```
| | |          | | | | |
t h e g r e e n a p p l e
```

it is tempting to form unifications like this: '< 1 t h e < 2 > a p p l e >'
so that the system can eventually accommodate class hierarchies, phrases
like 'turn ... on' and discontinuous dependencies.

But the counter-argument and previously-adopted principle is that
every coherent sequence should be made into a new pattern because
this means an overall simplification of the system. Also, there
is the expectation that each such coherent sequence is likely to
be recognised two or more times and, if not, this can be sorted
out when grammars are cleaned up. And the sequential relationship
between 't h e' and 'a p p l e' is captured in the encodings of
full alignments - and these can be added to Old. Another argument
for forming each coherent sequence into a new pattern is that it
facilitates the generalisations that will be needed in any
realistic scheme for grammatical induction.

These remarks also apply to the scheme described in %30.

2 Before any pattern is added to Old, there is a need to check that Old
does not already contain a pattern with the same CONTENTS symbols and,
if so, to merge the two patterns. This suggests a possible answer to
the problem of forming intermediate-level structures: instead of doing
a simple match with pre-existing Old symbols, why not treat the
newly-created Old pattern as if it was a New pattern and parse it it
terms of pre-existing Old patterns. This should achieve the effect of
finding patterns with the same CONTENTS symbols but it should also
lead to parsings that recognise sub-structure within the newly-created
Old pattern. This possible scheme is described and discussed in %28
and %29.

A possible snag with this scheme is that it may lead to complex patterns
of recursion - because one or more new patterns may be created during
the parsing of a newly-created Old pattern and these need to be treated
in the same way. This is all potentially rather complex and difficult to
manage. Another possible snag is that, if sub-structure was encoded and
then the encoded patterns were treated in the same way, this could lead
to a situation where the encoded part of the pattern was *decoded* by
the parsing process while any residual non-encoded part of the pattern
was further encoded in the normal way.

In general, there seem to many complexities and difficulties along
this road. But it would still be good if the process of checking that
a newly-created Old pattern does not have the same CONTENTS symbols as
a pre-existing pattern could be somehow merged with the parsing process.

3 Another possible avenue is to modify the parsing process so that there
can be one *or more* target patterns on any one cycle. This would have
two potential benefits:

* Any given New pattern may be parsed into one *or more* separate structures.
By combining these in a higher-level pattern, the system may develop a
knowledge of intermediate-level structures.

* The parsing process should be more efficient (ie there may be fewer
cycles to achieve a final parsing).

4 There are still attractions in the idea that any kind of mismatch
between patterns during parsing should be a signal for learning.
Attractions of this idea include:

* It avoids discarding alignments during parsing. Any mismatch is
simply a signal for learning.

* It should give a handle on the learning of disjunctive structures:
the mismatch becomes an alternative in a given context.

Version 5.0 will go back to v 3.0, retain 'learning on mis-match',
and try to add the multiple targets feature to the parsing process.

FURTHER THOUGHTS:

Trying to add multiple targets to the process of building a hit
structure is likely to lead to an explosion in the size of the
hit structure.

Perhaps a better idea is to wait until normal parsing is finished
and then explore what concatenation of parsings is possible.

Likewise, it is probably best to wait until parsing is finished
before exploring what new structures can be derived from partial alignments.

%32 4/6/04

RESULTS FROM SP71, V 4.1 (WAS 5.0)

What was v 5.0 is really an extension of 4.0 so has been renumbered as 4.1.

With this input:

NEW PATTERNS:

ID1 (t h a t b o y r u n s)*1

ID2 (t h a t g i r l r u n s)*1

ID3 (t h a t b o y w a l k s)*1

ID4 (t h a t g i r l w a l k s)*1

the best full parsings after ID3 are:

ID1158: (< 1 t h a t b o y r u n s >)*1

ID1252: (< 34 < 29 t h a t b o y > < 31 32 r u n > < 33 s > >)*1

ID1260: (< 6 < 8 t h a t > < 3 4 b o y > < 5 r u n s > >)*1

ID1495: (< 7 t h a t g i r l r u n s >)*1

ID1601: (< 6 < 8 t h a t > < 3 2 g i r l > < 5 r u n s > >)*1

ID1881: (< 28 t h a t b o y w a l k s >)*1

ID1990: (< 34 < 29 t h a t b o y > < 31 30 w a l k > < 33 s > >)*1

This still leaves the problem that the system does not recognise
that '< 29 t h a t b o y >' can be parsed into '< 8 t h a t >'
and '< 3 4 b o y >'. Concatenation of small alignments, as planned,
may provide an answer but it is likely to lead to the recreation
of analyses like ID1260. It is tempting to look again at the idea
of parsing proposed new additions to old_patterns into smaller
patterns ...

%33 7/6/04

ANOTHER WAY FORWARD?

Here is another possible handle on the problem:

Modify the parsing process so that, at the end of each cycle, the system
tries to find *combinations* of sub-parsings that describe the New pattern
economically. This means a process of heuristic search through the
possibilities, a bit like the way the system builds alternative grammars.
At each stage in the process, the system should retain a small sub-set
of the possible combinations, probably about 2.

By contrast with the current system, an alignment should only be passed
on to the next cycle of the parsing process if it is 'full', meaning
that all the CONTENTS symbols in Old patterns have been matched.

When parsing has proceeded as far as it can go on this basis, then
learning can occur as before: look for unmatched portions of the New
pattern and unmatched portions of Old patterns and derive new
patterns accordingly.

These ideas will be tried in version 5.0.

%34 11/6/04

RESULTS FROM SP71, V 5.0

With this input:

```
[
    [
        (j o h n l o v e s m a r y)
    ]
    [
        (< !0 j o h n >)
        (< !1 l o v e s >)
        (< !2 m a r y >)
        (< !3 < 0 > < 1 > < 2 > >)
    ]
```

```
]

combine_alignments() now produces this composite alignment:

ID13: NSC = 5720.97, EC = 0.00, CR = 1.#J, CD = 5720.97,
Absolute P = 1

0     j o h n l o v e s m a r y   0
      | | | | | | | | | | | | |
1 < 0 j o h n | | | | | | | | | > 1
              | | | | | | | | |
2 < 1         l o v e s | | | | > 2
                        | | | |
3 < 2                   m a r y > 3

Sequence ID13 as flat pattern:

ID13 (< 0 j o h n > < 1 l o v e s > < 2 m a r y >)*1

When the sequence is printed as a flat pattern, we can see that the
Old symbols that are not aligned with New symbols are in the right
positions. But the function that writes out the alignments of symbols
puts them in different positions. That function needs to be modified
so that we can see the 'true' positions of all symbols when they appear
in an alignment. This needs to be done both for the horizontal
presentation and the vertical presentation. This will be done
in version 5.1.

In v 5.1, the composite alignment now looks like:

ID13: NSC = 5720.97, EC = 30.15, CR = 189.77, CD = 5690.82,
Absolute P = 8.41002923806e-010

0     j o h n       l o v e s       m a r y   0
      | | | |       | | | | |       | | | |
1 < 0 j o h n >     | | | | |       | | | |   1
                    | | | | |       | | | |
2               < 1 l o v e s >     | | | |   2
                                    | | | |
3                               < 2 m a r y > 3

and

ID13: NSC = 5720.97, EC = 30.15, CR = 189.77, CD = 5690.82,
Absolute P = 8.41002923806e-010

    <
    0
j - j
o - o
h - h
n - n
    >
        <
        1
l ----- l
o ----- o
v ----- v
e ----- e
s ----- s
        >
            <
            2
m --------- m
a --------- a
r --------- r
y --------- y
            >

%35 12/6/04

RESULTS FROM SP71, V 5.1

The current function for comparing patterns containing brackets
produces this result:

ID17 : ID13 : ID5 : #41 NSC = 5310.44, EC = 33.53, CR = 0.01, CD = 5276.91,
Absolute P = 0.00

0         j o h n       l o v e s       m a r y               0
          | | | |       | | | | |       | | | |
1     < 0 j o h n >     | | | | |       | | | |               1
      | |               | | | | |       | | | |
```

```
2     | |          < 2 l o v e s >     | | | |               2
      | |                              | | | |
3     | |                         < 3 m a r y >             3
      | |                                                   |
4 < 4 < 0                                                   > < 2 > < 3 > > 4
```

and it cannot produce the 'correct' result. The function will be re-written in v 5.2.

The new function will take advantage of the fact that each left bracket constrains the choice of right bracket and vice versa. This constrains the variety of legal matches between brackets.

%36 14/6/04

RESULTS FROM SP71, V 5.2

With this input:

```
[
    [
        (j o h n l o v e s m a r y)
    ]
    [
        (< !0 j o h n >)
        (< !2 l o v e s >)
        (< !3 m a r y >)
        (< !4 < 0 > < 2 > < 3 > >)
    ]
]
```

the program now produces this alignment:

ID17 : ID13 : ID5 : #47 NSC = 5310.44, EC = 15.47, CR = 0.00, CD = 5294.97, Absolute P = 0.00

```
0         j o h n       l o v e s       m a r y     0
          | | | |       | | | | |       | | | |
1     < 0 j o h n >     | | | | |       | | | |     1
      | |         |     | | | | |       | | | |
2     | |         | < 2 l o v e s >     | | | |     2
      | |         | | |           |     | | | |
3     | |         | | |           | < 3 m a r y >   3
      | |         | | |           | | |         |
4 < 4 < 0         > < 2           > < 3         > > 4
```

but, on the same cycle (cycle 2) it makes the same composite alignment as before. We either need to check for duplicates before adding alignments to current_alignments or we need to prevent re-making of the same composite alignments in some other way.

At present, we are following the tentative rule that alignments are only used as driving patterns and never as target patterns. If that is valid, then current_alignments need only contain alignments that are driving patterns *on the current cycle*. This should prevent 'old' alignments hanging around and being re-used to make the same composite alignments over and over again. All alignments that are not valid driving patterns on a given cycle may be discarded.

Another problem is that, at present, composite alignments include sub-sequences of the best alignment. A possible solution is to start new combinations *only* from sub-alignments that ***do not fit*** into previously-started combinations.

These things will be attempted in v 5.3.

%37 15/6/04

RESULTS FROM SP71, V 5.3

With this input:

```
[
    [
        (j o h n l o v e s m a r y)
    ]
    [
        (< !0 j o h n >)
        (< !2 l o v e s >)
        (< !3 m a r y >)
        (< !4 < 0 > < 2 > < 3 > >)
        (< !6 o h n l o >)
        (< !7 v e s m a >)
        (< !5 < 6 > < 7 > >)
```

```
   ]
]
```

the program now forms alignments like this:

ID26 : ID20 : ID5 : #71 NSC = 5539.99, EC = 15.33, CR = 0.00, CD = 5524.66,
Absolute P = 0.00

```
0        j o h n      l o v e s      m a r y    0
         | | | |      | | | | |      | | | |
1    < 0 j o h n >    | | | | |      | | | |    1
     | |        |    | | | | |      | | | |
2    | |        | < 2 l o v e s >    | | | |    2
     | |        | | |          |    | | | |
3    | |        | | |          | < 3 m a r y >   3
     | |        | | |          | | |        |
4 < 4 < 0        > < 2          > < 3        > > 4
```

and this:

ID27 : ID22 : ID8 : #77 NSC = 4186.99, EC = 14.18, CR = 0.00, CD = 4172.81,
Absolute P = 0.00

```
0        j o h n l o      v e s m a    r y 0
           | | | | |      | | | | |
1    < 6   o h n l o >    | | | | |        1
     | |            |    | | | | |
2    | |            | < 7 v e s m a >      2
     | |            | | |          |
3 < 5 < 6            > < 7          > >    3
```

and it avoids keeping combinations of alignments that are merely
a sub-set of combinations already formed.

The next stage is to use combinations as the basis for learning:

1 For each of the 'best' combinations, create an abstract pattern,
add it to Old, and create a composite alignment based on that
combination and abstract pattern.

This will be done in v 5.4.

The immediate problem is that, at present, the program forms
composite alignments after every cycle (which can then be
matched with abstract patterns in subsequent cycles) and then
deletes all combinations at the end of every cycle. But it was
envisaged that learning would occur when all parsing cycles
in phase 1 had been completed. This needs sorting out.

A possible answer is to wait until all parsing cycles have been
completed and then process *all* remaining alignments to derive
code patterns from them. With each composite alignment, this
should achieve the effect of creating an 'abstract' pattern
that ties all the basic alignments together. This would
neatly integrate the creation of codes from alignments and
the creation of abstract patterns. Unlike the current procedure
for creating codes, each new code/abstract pattern should have
its own ID-symbols.

%38 16/6/04

SP71, v 5.4: at present, during parsing (phase 1), the program
only retains alignments from the most recent cycle. This means
that 'good' alignments from early cycles are lost. This needs
to be changed so that it retains alignments from all cycles but
only uses a subset as driving patterns for the next cycle. Then,
when all cycles are complete, the best alignments from all cycles
can be used to derive new patterns to be added to Old. Likewise,
in phase 2, the best alignments from all cycles need to be used
for calculating frequencies and deriving grammars.

These changes have been made except that new patterns are
created by sequence::create_patterns() as before with
alignments containing only two rows, an abstract pattern is
created in that method as before and an abstract pattern is
derived from combination::make_composite_aligment() by
creating a code pattern and adding ID-symbols.

Point of note:

1 Because the system is able to form composite alignments, it is
able to form grammars like this:

GRAMMAR GR5

Grammar GR5, derived from NULL, G = 4706.36, E = 36.70, score = 4743.07:

Grammar patterns:

ID59 (< 2 t h a >)*1

ID60 (< 3 t >)*1

ID61 (< 5 4 b o y r u n s >)*1

Code patterns:

ID58 (< 2 > < 3 > < 5 4 >)*1

In other words, the grammar, excluding the code pattern, does
not need to contain an abstract pattern.

2 Although it is a neat idea to make abstract patterns by the
same process that makes code patterns, it is not clear what
should happen when the sub-alignments contain 'discrimination'
symbols. The resulting code pattern will encode a specific
pattern from New and will lack the generality of a truly
abstract pattern. However, since this possible merging of
processes *is* a neat idea, it deserves further thought.

%39 21/6/04

FURTHER THOUGHTS ON CREATION OF ABSTRACT PATTERNS BY CREATION OF 'CODE' PATTERNS

Perhaps the best way forward is to try creating a 'code' pattern
for all 'completed' alignments (those in current_alignments when
all cycles have been completed), adding those code patterns to
Old patterns, and then seeing what the result is.

There may be a case for comparing all proposed new additions to Old
with existing patterns in Old and this may be a mechanism for
forming abstractions.

Versions 5.5 will distinguish between 'current' alignments and
'driving' alignments, as described in %38, and will add a code
pattern for *every* completed alignment to old_patterns, as
just described.

%40 22/6/04

INTERMEDIATE RESULTS FROM SP71, V 5.5

After two New patterns have been processed, the Old patterns are:

OLD PATTERNS:

ID49 (< 1 t h a t b o y r u n s >)*1

ID56 (< 2 t >)*1

ID57 (< 3 h a t b o y r u n s >)*1

ID62 (< 11 t h a t g i r l r u n s >)*1

ID78 (< 12 t h a t >)*1

ID82 (< 14 13 g i r l >)*1

ID83 (< 14 15 b o y >)*1

ID81 (< 16 r u n s >)*1

ID77 (< 17 < 12 > < 14 > < 16 > >)*1

ID87 (< 18 t h a t >)*1

ID91 (< 20 19 g i >)*1

ID92 (< 20 21 b o y >)*1

ID90 (< 22 r >)*1

ID94 (< 23 l r >)*1

ID93 (< 24 u n s >)*1

```
ID86 (< 25 < 18 > < 20 > < 22 > < 23 > < 24 > >)*1

An obvious problem is that there are duplicates. For example,
there is:

ID78 (< 12 t h a t >)*1

from:

0    t h a t      g i r l r u n s   0
     | | | |            | | | |
1 < 1 t h a t b o y       r u n s > 1

and:

ID87 (< 18 t h a t >)*1

from:

0    t h a t      g i r l r u n s   0
     | | | |          |    | | |
1 < 1 t h a t b o y    r    u n s > 1

It looks as if we need to institute a rule that:

Before any pattern is added to old_patterns, a check is made to see
whether old_patterns already contains a pattern with the same CONTENTS
symbols. If it does, the two patterns are merged in the sense that a
copy of each CLASS_SYMBOL in the newly-created pattern is added to
the pre-existing pattern unless a copy is already present in the
pre-existing pattern.

This rule will be implemented in v 5.5.

%41 24/6/04

FURTHER INTERMEDIATE RESULTS FROM V 5.5

The program has been modified so that it does *not* form an abstract
pattern explicitly when it creates new sub-patterns but it does
create an 'abstract' pattern implicitly by creating an encoding
for each good alignment. Also, the distinction between 'code'
patterns and other patterns has been dropped.

With these four patterns:

NEW PATTERNS:

ID1 (t h a t b o y r u n s)*1

ID2 (t h a t g i r l r u n s)*1

ID3 (t h a t b o y w a l k s)*1

ID4 (t h a t g i r l w a l k s)*1

the best grammar is:

GRAMMAR GR80

Grammar GR80, derived from GR62, G = 10434.73, E = 337.55, score = 10772.28:

Grammar patterns:

ID548 (< 3 11 6 7 33 > < 1 5 13 14 8 > < 10 >)*1

ID549 (< 3 11 6 7 33 t h a t >)*4

ID550 (< 1 5 13 14 8 b o y >)*2

ID551 (< 10 r u n s >)*2

ID563 (< 3 11 6 7 33 > < 1 4 34 > < 10 >)*1

ID565 (< 1 4 34 g i r l >)*2

ID583 (< 3 11 6 7 33 > < 1 5 13 14 8 > < 9 32 35 >)*1

ID586 (< 9 32 35 w a l k s >)*2

ID637 (< 3 11 6 7 33 > < 1 4 34 > < 9 32 35 >)*1

Points to note:
```

1 This grammar is, its essentials, the five words in the original 4
sentences (each with code symbols) together with an 'abstract'
pattern representing the encoding of each of the four sentences.
Each of the latter patterns represents an encoding of a specific
sentence so it is not truly general or abstract.

2 There are far more ID/CLASS_SYMBOLs than are needed and the excess
are not removed by cleaning up because there is at least one reference
for each referent. This excess of class symbols can mean that 'good'
alignments receive unecessarily high scores (because of the excess
of ID symbols) and so may be deselected.

In v 5.6, this excess of code symbols may be tamed by giving using
the same CLASS_SYMBOLs for every instance of a given sequence of
CONTENTS symbols rather than creating new ones for each instance.
This implies generalisation from one context to another - which is
probably something that is needed in the long run.

%42 25/6/04

INTERMEDIATE RESULTS FROM SP71, V 5.6

With these four New patterns:

ID1 (t h a t b o y r u n s)*1

ID2 (t h a t g i r l r u n s)*1

ID3 (t h a t b o y w a l k s)*1

ID4 (t h a t g i r l w a l k s)*1

the best grammar is currently:

Grammar GR131, derived from GR98, G = 9471.92, E = 147.07, score = 9618.99:

Grammar patterns:

ID722 (< 5 t h a t >)*4

ID723 (< 7 b o y >)*2

ID724 (< 1 r u n s >)*2

ID728 (< 6 g i r l >)*2

ID743 (< 43 w a l k s >)*2

This is 'correct' in the sense that it enables all four sentences to
be analysed in terms of the three words they each contain.

The reason that the program does not detect the division of 'r u n s'
and 'w a l k s' into a stem with a terminal 's' is that the full
patterns 't h a t b o y r u n s' and 't h a t g i r l r u n s' have
been removed from Old by the time the last two patterns appear
(containing 'w a l k s').

The program needs to be modified to form code/abstract patterns
after the formation of each composite alignment.

With this modification, the best 3 grammars are:

GRAMMAR GR182

Grammar GR182, derived from GR146, G = 10822.95, E = 110.57, score = 10933.52:

Grammar patterns:

ID1079 (< 1 t h a t >)*4

ID1080 (< 3 b o y >)*2

ID1081 (< 5 r u n s >)*2

ID1099 (< 2 g i r l >)*2

ID1130 (< 4 w a l k s >)*2

GRAMMAR GR183

Grammar GR183, derived from GR140, G = 10862.75, E = 94.50, score = 10957.24:

```
Grammar patterns:

ID1079 (< 1 t h a t >)*4

ID1080 (< 3 b o y >)*2

ID1081 (< 5 r u n s >)*2

ID1099 (< 2 g i r l >)*2

ID1106 (< 4 w a l k s >)*2

ID1107 (< 59 < 1 > < 3 > < 4 > >)*1

GRAMMAR GR162

Grammar GR162, derived from GR146, G = 10862.16, E = 95.08, score = 10957.24:

Grammar patterns:

ID1079 (< 1 t h a t >)*4

ID1080 (< 3 b o y >)*2

ID1081 (< 5 r u n s >)*2

ID1099 (< 2 g i r l >)*2

ID1130 (< 4 w a l k s >)*2

ID1159 (< 55 < 1 > < 2 > < 4 > >)*1

GRAMMAR GR172

Grammar GR172, derived from GR146, G = 10862.16, E = 95.08, score = 10957.24:

Grammar patterns:

ID1079 (< 1 t h a t >)*4

ID1080 (< 3 b o y >)*2

ID1081 (< 5 r u n s >)*2

ID1099 (< 2 g i r l >)*2

ID1130 (< 4 w a l k s >)*2

ID1199 (< 60 < 1 > < 2 > < 4 > >)*1

The main problem here is that each 'abstract' pattern only encodes one
specific sentence. Another problem is that the CONTENTS symbols in
pattern ID1159 are the same as the CONTENTS symbols in pattern ID1199.

%43 21/7/04

Current thoughts:

1 Each pattern should have an ID-symbol for the pattern itself and zero
or more ID-symbols for the context or contexts in which it may appear,
if any. At present, the program provides each pattern with *only* a
context ID-symbol and it deliberately generalises from one context
to another. This avoids the proliferation of class symbols but is
probably not entirely satisfactory in the long run because, while
there should be some generalisation across contexts, there will also
be cases where different contexts should be distinguished. For
example, a word like 'bank' should be marked as *both* a noun and a verb.

The program needs to be modified to give each pattern its own unique
ID-symbol, regardless of whether it gets one or more context ID-symbols.

2 It seems to be necessary to check each proposed new pattern against
those already in Old to see that it does not have the same CONTENTS
symbols as an existing pattern. If this kind of match is found, then
the proposed new pattern is discarded and the existing pattern is
used instead. Probably, the existing pattern needs to be modified
to receive the context ID-symbol of the proposed new pattern. To
avoid undue proliferation of these context ID-symbols, they probably
need to be tamed by some other method than the existing method, as
described below.

3 Checking proposed new patterns against existing Old patterns is
rather similar to matching New patterns from the environment against
```

existing Old patterns. In the latter case, we may get exact matches
with CONTENTS symbols of the Old patterns or we may get partial
matches. Partial matches lead to the creation of new patterns.

So, if we are to follow through logically, checking a proposed new
pattern against Old patterns should sometimes lead to partial matches
and, in such cases, we would expect to generate new patterns -- which
would themselves need to be checked against Old patterns. This
recursive loop could lead to a mushrooming of proposed new patterns
and could be difficult to manage.

4 A process of matching proposed new patterns against Old patterns
seems to be needed in order to generalise 'code' patterns that the
program creates. For example, ID1199 (< 60 < 1 > < 2 > < 4 > >)*1
may be matched against ID1107 (< 59 < 1 > < 3 > < 4 > >)*1 which
should lead to something like IDxxxx (< yy < 1 > < 2 3 > < 4 > >).

%44 22/7/04

(continued)

5 Regarding the possibility of a recursive explosion described
under 3, this should be kept in check if the learning system is
repeatedly purging itself of 'bad' patterns and only keeping
patterns in the best one or two grammars. This should ensure
that Old does not contain much redundancy in the form of repetitions
of patterns or parts of patterns and thus reduce the chance of a
recursive explosion.

6 Whether or not the recursive explosion is checked as described
under 5, a possible way to manage it might be to put the CONTENTS
symbols of proposed new patterns into the repository of New patterns,
ahead of what is already there. Then the proposed new patterns would
be processed in exactly the same way as New patterns from
the environment.

7 An alternative scheme which may be used as a stop-gap, pending
something better, is as follows: an arbitrary restriction may be
imposed that, with regard to patterns derived from an incomplete
match between New and Old, only the first step in the recursive
loop will be taken (ie the only proposed new patterns that will
be matched to Old patterns will be ones derived from a match between
an original New pattern and an Old pattern). In addition, the system
may look for opportunities to merge abstract patterns as
described next.

8 It looks as if the process is splitting into two more-or-less
distinct modules:

* If there is a partial match between a New pattern and a
pattern in Old, derive new patterns or merge the patterns.

* If there is an exact match between a New pattern and one or
more patterns in Old, encode the New pattern, or part of it,
in terms of the Old patterns. The encoding is then a proposed
new pattern to be added to Old.

In both cases, proposed new patterns (which may be DATA patterns
or CODE patterns) may be treated as New patterns to be analysed
in the same way. This should achieve recursive parsing *and*
recursive derivation of new patterns.

This means a fairly radical reorganisation of the program and
should be marked with a new version number.

A snag with this approach is that we would have to ditch a lot
of code used for building and displaying multiple alignments.
If multiple alignments are to be displayed, they would have
to be reconstructed from successive encodings.

%45 23/7/04

(continued)

9 Comparing proposed new patterns with existing patterns may
provide a means of identifying significant suffixes (or prefixes).
For example, matching 'r u n s' with 'w a l k s' should reveal
the terminal 's'. But this kind of comparison between relatively
small patterns is also likely to throw up a lot of rubbish.

In earlier versions of SP71, the terminal 's' gets identified because
whole sentences are compared with each other. This expands the
context and thus reduces the amount of rubbish but, in the long

run, it may not be a viable means of suppressing rubbish. What
makes the terminal 's' viable in the long run is the large number
of different words to which it applies. It may be that we have to
put up with the rubbish in order to allow 'good' substructures
an opportunity to reveal themselves. It is possible that the
larger context of a whole sentence may also help to show what
is significant even though that structure is expressed in the
form of abstract patterns rather than specific sentences.

10 Here is a tentative structure for SP71, v 6.0:

#1 Instead of building multiple alignments explicitly, as is
done at present, the system will derive encodings from the matches
that it finds between patterns and then repeat the process by looking
for matches for the encodings. If multiple alignments are needed for
display purposes, these may be derived from the sequence of encodings.
This is a major reorganisation and will be attempted first.

#2 Once the system has been reorganised to process encodings
repeatedly, then the next part of the system may be attempted:
applying the same principles to newly-created encodings so that
the system can learn part-whole hierarchies (and heterarchies)
and class-inclusion hierarchies (and heterarchies).

%46 26/7/04

(continued)

11 SP71, v 6.0, has been developed to the point where parsing in all
cycles after the first is done using code patterns derived from
previous cycles. On each cycle, all codes from previous cycles are
deleted and the search for alignments continues with codes derived
from the previous cycle only. The process for selecting alignments
without undue bias to one part of the New pattern or another is
applied only in the first cycle.

The next step is to write a function that can derive an alignment
recursively from a code pattern and the code patterns that it
contains.

12 A snag with the present scheme is that the scoring method loses
information about the number of original New symbols in an alignment.
This can be remedied by carrying this information from code pattern
to code pattern in a succession of encodings.

13 A possible alternative to the present scheme is to stick to the
original method for forming multiple alignments but to derive encodings
as and when required. This may be a path to follow at some stage but,
for the time being, it is probably best to continue developing v 6.0
along the current lines.

%47 27/7/04

(continued)

There are snags in creating the function build_multiple_alignments() in
a 'kosha' way. Without the flexibility of the full SP system, it is necessary
to make awkward fudges and simplifying assumptions. In the long run, it
is better to create multiple alignments form code patterns using the SP
system itself. For the time being, the writing of this function will be
halted (and the code put in odds_and_ends.cpp).

Further development will concentrate on working with simple code patterns.
Now that parsing with code patterns is working reasonably well, we may now
look at learning with code patterns - meaning deriving new patterns from
partial matches between patterns.

At present, there seems no escape from the need to check each proposed
new pattern against the patterns in Old, with the possibility that this
will yield more partial matches leading to the creation of more new
patterns ... and so on. We shall follow this path unless something
better turns up.

The focus now is on the development of 'sequence::create_patterns()':
deriving new patterns from a partial match between two patterns. Here,
the focus is on CONTENTS symbols, not IDENFIFICATION symbols.

The aim is to *integrate* parsing and learning: instead of a succession
of parsing 'cycles', the program will either find an exact match between
the New pattern (or part of it) and the CONTENTS symbols of an Old
pattern, or it will find a partial match. In both cases, the result
will be one or more encodings of the New pattern, either in terms of
the ID-symbols of a pre-existing Old pattern or in terms of newly-created

Old patterns.

For the time being, the potential recursive explosion will be tamed
by *not* matching newly-created patterns against the Old patterns.

'sequence::create_patterns()' will be renamed as 'sequence::create_encoding()'
and this single function will either create a parsing-style encoding or it
will create new patterns and derive an encoding from them. This function
will also supercede the function 'sequence::make_code()'. The latter
function already does a form of finding partial matches between patterns,
so it is logical to integrate it with 'sequence::create_encoding()' which
will be designed to be a more general method for processing partial matches.

%48 29/7/04

SP71, v 6.1

This version of the program now integrates the creation of new patterns
with the parsing process.

At present, the program distinguishes between two situations:

* Alignments that are 'full' in the sense that they encode all the
CONTENTS symbols of the target pattern but they do not necessarily
encode all the CONTENTS symbols of the driving pattern.

* Alignments that are partial matches between the CONTENTS symbols
of the driving pattern and the target pattern.

In the former case, the program derives a code pattern for the
driving pattern. In the latter case, it creates new patterns and
could potentially encode the driving pattern in terms of
those patterns.

Questions that arise:

1 It would be nice if the process of deriving an encoding from a 'full'
alignment were fully integrated with the process of forming new
pattern from a partial alignment between two patterns. This is because:

* Both processes depend on their being an incomplete match between
the two patterns. It would be logical to treat them in the same way.

* The decision to regard an alignment as 'full' even if its driving
pattern is not full matched is really a hang-over from when the
parsing process would only work if alignments with partially-matched
New patterns were accepted. This is no longer necessary so there
may be a case for insisting that the driving pattern must be fully
matched as well as the target pattern. But this issue goes away
if the process of deriving new patterns is integrated with the
process of creating an encoding.

2 If 'a b c d e f g h i' is matched with 'a b c g h i' the system
needs to create a disjunctive group comprising 'd e f' and a 'null'
pattern represented by something like '< 1 2 >'. Since a null element
is likely to be required in several different contexts, this raises
the question of whether we create separate null elements for every
different context, each with its own identifier, or whether we have
a single null element that may be listed as appearing in several
different contexts. This relates to the general question of how
disjunctive classes are specified:

* Every member of a given disjuntive class has an ID-symbol representing
that class or ...

* Every context lists the patterns that may appear in that context.
For example, 'a b c < 22 1 42 87 35 21 > g h i', or something similar,
could be a way of specifying which patterns could appear between 'a b c'
and 'g h i'. It looks as if some kind of 'punctuation' symbol would be
needed to achieve legal alignments.

The second option could lead to the first if recurrent groupings of
ID-symbols were given their own ID-symbol. This would then be, in
effect, a symbol for the whole disjunctive set of ID-symbols.

3 If the test for 'full' alignments were to insist that CONTENTS symbols
in both the driving pattern and the target pattern had to be fully
matched, this would mean that alignments like this:

```
    a b c d e f g h i
    | | |
< 1 a b c >
```

would be regarded as partial matches so that they could be processed
to create new patterns (which in this case would be something like
'< 2 d e f g h i >'). This would open the door to the creation of
'intermediate' level structures - which are the kinds of structures
that SP70 could not learn and which SP71 is intended to learn.

4 Here are some tentative ideas for integrating
'sequence::create_encoding()' and 'sequence::make_code()':

* The integrated function would be modelled on
'sequence::create_encoding()'.

* Where the CONTENTS symbols of a target pattern are completely matched,
the method would return the ID-symbols of that pattern as an encoding
of the relevant portion of the driving pattern.

* Where CONTENTS symbols in the driving pattern or the target pattern
are not fully matched, the method would create new patterns with their
own encodings (as at present). The method should be able to form 'null'
patterns.

* If a coherent sequence of one or more unmatched symbols is too small
to justify the 'cost' of encoding them as a separate pattern, they would
be passed through directly into the encoding.

The next step is to enable 'sequence::create_encoding()' to be able to
create null patterns. This has now been done.

%49 29/7/04

SP71, v 6.1, gives results like this:

From alignment ID42

```
0     < 0 5 > < 2 6 > < 3 7 >   0
      | |   | | |   | | |   |
1 < 4 < 0   > < 2   > < 3   > > 1
```

encoding is:

ID46 (< 4 >)*1

NSC = 5867.96, EC = 11.32, CR = 518.52, CD = 5856.65

< (0, 2.84, LB, CN), 4 (1, 5.64, CS, CN),
> (2, 2.84, RB, CN), frequency = 1.

The reason that the result is '< 4 >' and not '< 4 5 6 7 >' is that
'sequence::make_code()' looks for unmatched ID-symbols rather than unmatched
symbols of any kind. If 'sequence::make_code()' is integrated with
'sequence::create_encoding()', then this problem may be cured. In that
case, an alignment like this:

```
    a b c d e f g h i
    | | |
< 1 a b c >
```

may result in an encoding something like this: '< 1 > d e f g h i' or,
perhaps, '< 1 > < 2 >', where '< 2 >' is the code for '< 2 d e f g h i >'.

%50 30/7/04

FURTHER THOUGHTS (SP71, V 6.1): INTEGRATION OF make_code() and
create_encoding()

With an alignment like this:

```
    a b c d e f g h i
    | | |
< 1 a b c >
```

the best encoding looks something like this: '< 1 > d e f g h i'.

With an alignment like this:

```
0     < 0 5 > < 2 6 > < 3 7 >   0
      | |   | | |   | | |   |
1 < 4 < 0   > < 2   > < 3   > > 1
```

the best encoding looks something like this: '< 4 5 6 7 >'.

In these two cases, the rule seems to be:

"If the CONTENTS symbols of the target pattern are fully matched, substitute
the ID-symbols for that pattern. Also, add to the encoding any driving
symbols in the alignment that are not matched."

[An issue here is the positioning of the driving symbols. In the first
case, they follow the last bracket of the encoding. In the second case, they
precede the last bracket of the encoding. A possible answer here is that,
in the first case, the relative positions of the unmatched driving symbols
and the last bracket are undefined whereas in the second case the relative
positions are defined. A tentative rule is: "If the relative positions of
symbols are defined, then honour those positions. Otherwise, try to ensure
coherence of encoding symbols."]

With an alignment like this:

```
    a b c d e f g h i
    | | |       | | |
< 1 a b c x y z g h i >
```

the best encoding looks something like this:

```
< 1 a b c < 2 > g h i >
< 2 3 d e f >
< 2 4 x y z >
< 1 3 >
```

So a tentative rule covering all cases is:

"If the CONTENTS symbols of the target pattern are fully matched, substitute
the ID-symbols for that pattern. Also, add to the encoding any driving
symbols in the alignment that are not matched.

If the CONTENTS symbols of the target pattern are *not* fully matched,
create new patterns for the matched and unmatched parts of the alignment
and then encode the driving pattern in terms of those patterns."

This (composite) rule will be tried in v 6.2.

Further thoughts:

With an alignment like this:

```
0           < 2 6 >         0
            | |   |
1 < 4 < 0 > < 2   > < 3 > > 1
```

the best encoding looks something like this: '< 4 < 0 > < 2 6 > < 3 > >'.
Alternatively, it may be argued that, since the CONTENTS symbols of the
target pattern are not all matched, it is not appropriate to use it at all.
Another possible encoding is: '< 7 < 2 6 > >'. This is created by merging
the driving pattern with that part of the target pattern that it has been
matched to and then giving the new pattern its own ID-symbols.

%51 1/8/04

SP71, v 6.2, does not yet fully integrate the process of creating new
patterns and creating encodings. But the process of creating encodings
now allows unmatched symbols in the driving pattern to be encorporated
into the encoding. The two functions are moving closer together and it
should be possible to integrate them fully at some stage.

%52 3/8/04

With an alignment like this:

```
0           < 5 > < 7 8 >   0
            | | | | |   |
1 < 9 < 3 > < 5 > < 7   > > 1
```

the program is creating illegal patterns like this:

ID27 (< 13 < 5 > < 7 >)*1

and this:

ID30 (< 17 > >)*1

and a 'silly' pattern like this:

ID31 (< 15 14 8 >)*1

The illegal patterns can be avoided if, instead of creating a separate
unified pattern for each coherent subsequence that is unified, the program

creates a single unified pattern containing slots for the variable parts of
the alignment. What this means, in effect, is that the role of the unified
pattern and the role of the abstract pattern are merged. There is no need
for a separate abstract pattern.

This makes good sense if we look ahead to the possibility that the system
will detect and abstract patterns for discontinuous dependencies in syntax.
It also makes sense if we are thinking of class hierarchies where the
information at any one level in a hierarchy above the 'instance' level
is a unification of two or more patterns relating to that level.

There may also be a case for encoding the unmatched portions of the alignment
in terms of single patterns corresponding to the original driving pattern or
target pattern rather than a succession of discrete patterns whose sequence
must then be recorded in a separate encoding pattern.

These things should be explored in a new version of the program: SP71, v 6.3.

%53 4/8/04

This alignment:

```
0     t h a t       g i r l r u n s   0
      | | | |             | | | |
1 < 1 t h a t b o y       r u n s > 1
```

gives rise to these patterns:

ID36 (< 11 10 g i r l >)*1

ID37 (< 11 12 b o y >)*1

ID33 (< 13 t h a t < 11 > r u n s >)*2

Then, in Phase 2, we get alignments like this:

```
0      t h a t       b o y r u n s   0
       | | | |           | | | |
1 < 13 t h a t < 11 >      r u n s > 1
```

and this:

```
0         t h a t b o y   r u n s 0
                  | | |
1 < 11 12         b o y >         1
```

From the first of these alignments is derived encodings like these:

ID59 (< 22 21 b o y >)*1

ID60 (< 22 23 < 11 > >)*1

ID56 (< 24 t h a t < 22 > r u n s >)*3

If the parameters are set to be generous enough, the second alignment
would lead to the encoding '< 11 12 >'.

Problems:

1 It is not appropriate for the system to create patterns like ID56, ID59
and ID60. This merely adds complexity to what was already encoded in
a sensible way.

2 At present, the system does not make the 'correct' connection between
the first and second alignments, above, that would lead to an encoding
corresponding to an alignment like this:

```
0      t h a t        b o y   r u n s   0
       | | | |        | | |   | | | |
1 < 13 t h a t < 11   | | | > r u n s > 1
               | |    | | | |
2              < 11 12 b o y >          2
```

3 If adjustments were made so that the program could make that kind of
connection, a problem arises from the fact that the current version of
the program makes alignments between code patterns, not alignments. This
means that the system has no way to check that the ordering of New symbols
is honoured in every alignment. The system would make an alignment between,
for example, '< 13 t h < 11 > a t r u n s >' and '< 11 12 b o y >'
even if the New pattern were to be 't h a t b o y r u n s'.

[continued 5/8/04] It looks as if we need to return to alignments as the
basis for learning rather than encodings of alignments. This would avoid

the awkward problem of building alignments from encodings (for display purposes). Since each alignment contains all the information that is contained in the corresponding encoding, any learning that can be done with encodings should also be possible with alignments.

SP71, v 7.0, will follow this path, building on the developments in SP71, v 6.3.

%54 5/8/04

FURTHER THOUGHTS ABOUT SP71, V 7.0

Regarding the proposals in %53, these amount to the following rules:

1 If the CONTENTS symbols of the target pattern are fully matched, make an alignment in which the matched CONTENTS symbols are converted into columns as usual and the unmatched columns of the driving pattern are copied into columns of the same depth as the new alignment and these are added at one or both ends of the columns corresponding to the target pattern.

2 If the CONTENTS symbols of the target pattern are not fully matched, then the system creates new patterns in the normal way.

In this version of the program, it is probably not necessary to invoke the combine_alignments() function. The effect of this function can be achieved via repeated alignment of single patterns. And the use of this function complicates the learning process because it obscures the simple idea that each hit sequence is between two patterns and *only* two patterns.

In short, the combine_alignments() function will be dropped and the process for building multiple alignments will be modified in accordance with rules 1 and 2.

%55 10/8/04

INTERIM RESULTS FROM SP71, V 7.0

In the program as it stands now, combine_alignments() has been removed and, on each cycle of Phase 1, the program checks to see whether the target Old pattern has been 'fully' matched (all CONTENTS symbols matched) or not. If it has, the program makes an alignment (which incorporates all the unmatched symbols in the driving pattern) and uses that alignment as a driving pattern on the next cycle. If the CONTENTS symbols are not fully matched, the hit sequence is marked as a candidate for the learning of new patterns.

There are two problems here:

1 With an input file like this:

```
[
    [
        (j o h n l o v e s m a r y)
    ]
    [
        (< !0 !5 j o h n >)
        (< !2 !6 l o v e s >)
        (< !3 !7 m a r y >)
        (< !4 < 0 > < 2 > < 3 > >)
    ]
]
```

the program makes the 'correct' alignments on the next cycle but, with the Phase 2 'learning' part of the program disabled, it cannot go on to build an alignment that includes '< !4 < 0 > < 2 > < 3 > >' because none of the alignments formed on the first cycle can make a 'full' CONTENTS symbol match with that pattern.

2 If the Phase 2 learning part of the program were not disabled, each partial alignment like this:

```
0 j o h n        l o v e s   m a r y 0
                 | | | | |
1          < 2 6 l o v e s >        1
           | |            |
2 < 4 < 0 > < 2           > < 3 > > 2
```

would lead to learning from the partial match. And this learning would be 'spurious' because it would lead to the re-learning of already-existing patterns.

To avoid these problems:

1 It looks as if the combine_alignments() function needs to be re-introduced.
We need also to return to the scheme in which unmatched New symbols
are *not* included in alignments.

2 We need to relax the rule that requires target patterns to be fully
matched before they can be used in alignments.

3 Learning should probably be postponed until all cycles of recognise()
have been completed.

These modifications will be made in v 7.1.

%56 11/8/04

INTERIM COMMENTS ON SP71, V 7.1

This version of the program is nearly back to where we were before:
building complete alignments (rather than code patterns) and combining
alignments at the end of each cycle. The main difference compared with
SP61 or SP62 is that alignments are always built by aligning a driving
pattern or alignment with one of the Old patterns rather than allowing
alignments to be target patterns. This should facilitate learning because
it reduces the complication of learning new patterns from alignments.

Rather than attempt to learn new patterns on every cycle of the recognise()
function, there seems to be a case for doing this when the system cannot
find any more ways of encoding the New pattern in terms of Old patterns
(as suggested in %55). This modification will be introduced into the
current version.

Alignments for learning will have these properties:

1 The New pattern is partially matched or the CONTENTS symbols of the
last Old pattern are partially matched, or both these things.

2 The CONTENTS symbols of all other Old patterns must be fully matched.

Given these conditions, it is possible to derive new patterns from the
New pattern or the Old pattern or both of them.

It should also be possible to derive intermediate levels of structure -
which is one of the main goals in developing SP71.

Given the use of combine_alignments(), it should be possible to insist that
driving alignments for the next cycle must always be ones in which the
CONTENTS symbols are fully matched. If this fails, then parsing fails
and learning begins.

%57 12/8/04

FURTHER INTERIM COMMENTS ON SP71, V 7.1

Given an appropriate grammar, the program now forms alignments like this:

```
0                 j o h n        l o v e s        m a r y           0
                  | | | |        | | | | |        | | | |
1            < 0 5 j o h n >      | | | | |        | | | |           1
             | |          |      | | | | |        | | | |
2            | |          | < 2 6 l o v e s >      | | | |           2
             | |          | | |           |      | | | |
3            | |          | | |           | < 3 7 m a r y >          3
             | |          | | |           | | |          |
4 < 4 < 100 > < 0          > < 2           > < 3          > < 102 > > 4
```

What sort of new patterns should this give rise to?

In general, with this version of the program, we need consider *only* the
New pattern and the latest Old pattern to have been added to an alignment.
In both cases, new patterns are derived from the given pattern if the given
pattern is partially matched (or possibly if there is a null element - see
below). The CONTENTS symbols of all other Old patterns should be fully matched
and so they should not be involved in learning.

As before, it looks as if we should look for *coherent* subsequences of
CONTENTS symbols that are matched or unmatched. In this case,
'< 0 > < 2 > < 3 >' is a coherent matched subsequence of row 4. So
the program should form new patterns something like this:

```
< 104 < 100 > < 103 > < 102 > >
< 103 < 0 > < 2 > < 3 > >
```

What about an alignment like this?

```
0                 j o h n       x y z l o v e s        m a r y           0
                  | | | |           | | | | |        | | | |
1           < 0 5 j o h n >         | | | | |        | | | |           1
            | |          |         | | | | |        | | | |
2           | |          | < 2 6    l o v e s >      | | | |           2
            | |          | | |                |      | | | |
3           | |          | | |                | < 3 7 m a r y >          3
            | |          | | |                | | |          |
4 < 4 < 100 > < 0          > < 2                > < 3          > < 102 > > 4
```

This may be processed in two stages:

1 In the first stage, the unmatched subsequence 'x y z' will lead to patterns like this:

```
< 105 < 100 > < 0 > < 103 > < 2 > < 3 > < 102 > >
< 103 104 x y z >
< 105 106 >
```

The second pattern clearly lies between 'j o h n' and 'l o v e s' but it is not so clear where it lies in relation to the pattern '< 4 ... >'. The rule seems to be to establish the encodings for 'j o h n' and 'l o v e s' in the pattern in row 4 and put the new structure between those two encodings.

2 In the second stage, '< 2 > < 3 >' is seen to be a coherent, matched subsequence which leads to these two patterns:

```
< 107 < 100 > < 0 > < 103 > < 108 > < 102 > >
< 108 109 < 2 > < 3 > >
```

The subsequence '< 0 >' is not abstracted as a separate pattern because it is already a coherent structure with its own code symbols.

In general, the process for abstracting new patterns must be sensitive to pre-established coherent sequences and the 'reference' symbols that identify them.

Question: with an alignment like this:

```
0 j o h n x y z l o v e s      m a r y           0
                               | | | |
1                       < 3 7 m a r y >          1
                        | |          |
2 < 4 < 100 > < 0 > < 2 > < 3          > < 102 > > 2
```

why should the program not abstract patterns like these:

```
< 103 104 j o h n x y z l o v e s >
< 103 105 < 100 > < 0 > < 2 > >
< 106 < 103 > < 3 > < 102 > >
```

This is a possibility but there may be a case for concentrating on the unification of patterns that involve two or more pre-established patterns or references to them.

%58 13/8/04

FURTHER THOUGHTS ON SP71, V 7.1

Given that the program uses combine_alignments(), there may be a case for insisting that an alignment must be 'full' in the sense that all New symbols are matched and all CONTENTS symbols of Old patterns must be matched before the next cycle can be started. If this condition does not obtain, then parsing should stop and learning should proceed.

So an alignment like this:

```
0       j o h n   x y z l o v e s        m a r y           0
        | | | |       | | | | |        | | | |
1 < 0 5 j o h n >       | | | | |        | | | |           1
                        | | | | |        | | | |
2                < 2 6 l o v e s >       | | | |           2
                                         | | | |
3                                  < 3 7 m a r y >          3
```

should give patterns like this:

```
< 8 9 x y z >
< 8 10 >
< 11 < 0 > < 8 > < 2 > < 3 > >
```

And an alignment like this:

```
0       j o h n         l o v e s         m a r y   0
        | | | |         | | | | |         | | | |
1 < 0 5 j o h n >       | | | | |         | | | |   1
                        | | | | |         | | | |
2               < 2 6 l o v e s >         | | | |   2
                                          | | | |
3                               < 3 7 m a r y > 3
```

may be taken on to the next cycle but, if no appropriate pattern is found, the learning processes should create a pattern like this:

```
< 8 < 0 > < 2 > < 3 > >
```

In short, the overall goal of the parsing and learning processes is to map every New pattern to a *single* abstract Old pattern via one or more levels of encoding. If parsing fails to obtain such a mapping, then learning processes should take over.

An alignment like this:

```
0       j o h n         l o v e s         m a r y v e r y m u c h 0
        | | | |         | | | | |         | | | |
1 < 0 5 j o h n >       | | | | |         | | | |                 1
                        | | | | |         | | | |
2               < 2 6 l o v e s >         | | | |                 2
                                          | | | |
3                               < 3 7 m a r y >               3
```

should lead to the creation of patterns like these:

```
< 8 < 0 > < 2 > < 3 > >
< 9 v e r y m u c h >
< 10 < 8 > < 9 > >
```

or, perhaps, these:

```
< 8 < 0 > < 2 > < 3 > < 9 > >
< 9 v e r y m u c h >
```

The former gives us a handle on intermediate-level structures but the latter does not.

A question arises whether this approach to the abstraction of structure is going to give us a satisfactory handle on the indentification of intermediate-level structures in the long run. There may be a case for matching Old patterns against Old patterns in order to discover such structures, something like this:

```
0 < 8 < 0 > < 2 > < 3 > < 9 > >         0
      | | | | | | | | |
1 < 8 < 0 > < 2 > < 3 > < 10 > < 11 > > 1
```

giving rise to new patterns like these:

```
< 12 < 0 > < 2 > < 3 > >
< 13 < 12 > < 9 > >
< 14 < 12 > < 10 > < 11 > >
```

or, perhaps, these:

```
< 12 < 0 > < 2 > < 3 > < 9 > >
< 9 15 < 10 > < 11 > >
```

In the latter example, '< 10 > < 11 >' is assigned to the pre-existing class '< 9 >'.

Another possibility is:

```
< 12 < 0 > < 2 > < 3 > >
< 13 < 12 > < 9 > >
< 9 15 < 10 > < 11 > >
```

Here, the system recognises both the unified 'chunk' which is '< 12 < 0 > < 2 > < 3 > >' and the disjunctive class.

As a tentative rule: "Always make coherent unified sequences into chunks and always recognises disjunctive classes."

%59 16/8/04

(continued from %58) In SP71, v 7.1, the above rule will be used unless
or until it proves to be unsatisfactory. A probable rider to the rule
is that "If an unmatched portion of the New pattern is found to lie
'opposite' a reference to a pre-existing class, then that portion of
the New pattern will be made into a new pattern and assimilated to the
pre-existing class." Another probable rider is that "A 'full' alignment
should map the New pattern, directly or indirectly, on to a *single* Old
pattern. If the New pattern is fully matched to two or more independent
Old patterns (as, for example, after the application of combine_alignments()),
then these should be conjoined, via references, into a single Old pattern.
This should be done only *after* a cycle to check that the given
combination of Old patterns is not *already* represented by an Old pattern."

Another thought is that the parsing/recognition process will naturally
give rise to matches between Old patterns, when a fully-matched New pattern
is matched against the Old patterns. In this case, the lowest Old pattern
in the driving alignment is being matched against other Old patterns. It
may be possible to derive new patterns from mis-matches arising in this
kind of situation. For example, the driving alignment might be something
like this:

```
    p u t         i t   o n
    | | |         | |   | |
< 3 p u t < prn 10 i t > o n >
```

and this might be aligned with '< 4 p u t < np > o n >' like this:

```
    p u t         i t   o n
    | | |         | |   | |
< 3 p u t < prn 10 i t > o n >
|   | | | |           | | | |
< 4 p u t < np        > o n >
```

From this kind of alignment, we may infer that the class '< prn >' or
'i t' may be assimilated to the class '< np >'.

If we are to take advantage of this kind of mismatch for learning, we
need to be prepared to derive new patterns from the mis-matches at
the *end* of the parsing/recognition process rather than simply terminating
that process when one or more alignments have been found that fully match
the New pattern. In general, continue the parsing/recognition cycles
until a mis-match is found, not until a full match is obtained.

%60 25/8/04

If we adopt the tentative rule that "the learning of new patterns is invoked
if and only if the New pattern is not fully matched, directly or indirectly,
to *one* Old pattern", then sub-structure can be learned from alignments
like this:

```
    j o h n       l o v e s m a r y
    | | | |       | | | | |
< 1 j o h n >     | | | | |
                  | | | | |
              < 2 l o v e s >
```

Here, the program should form Old patterns like these:

```
< 3 < 1 > < 2 > >
< 4 m a r y >
< 5 < 3 > < 4 > >
```

rather than these:

```
< 3 < 1 > < 2 > < 4 > >
< 4 m a r y >
```

Apart from the fact that it recognises a sub-structure, the justification
for favouring the first set of patterns over the second is that it conforms
to the rule that "any coherent sequence of structures should be formed
into a discrete pattern". This rule is applied at the lowest level
(when patterns like '< 1 j o h n >' and '< 2 l o v e s >' are identified),
so it seems reasonable that it should be applied at higher levels too.

Likewise, sub-structures can be learned from alignments like this:

```
        j o h n       l o v e s m a r y
        | | | |       | | | | |
    < 1 j o h n >     | | | | |
    | |         |     | | | | |
    | |         | < 2 l o v e s >
    | |         | | |           |
< 3 < 1         > < 2           > < 4 > >
```

As before, the sequence '< 1 > < 2 >' should be recognised as a coherent
sub-structure by making it into a discete pattern with its own ID-symbols.
'm a r y' should be made into a discrete pattern with its own ID-symbols
and assigned to the category '< 4 >'.

%61 26/8/04

OPTIONS FOR CREATING NEW PATTERNS

With an alignment like this:

```
0                 j o h n        l o v e s        m a r y           0
                  | | | |        | | | | |        | | | |
1            < 0 5 j o h n >      | | | | |        | | | |           1
             | |          |      | | | | |        | | | |
2            | |          | < 2 6 l o v e s >      | | | |           2
             | |          | | |            |      | | | |
3            | |          | | |            | < 3 7 m a r y >         3
             | |          | | |            | | |          |
4 < 4 < 100 > < 0          > < 2            > < 3          > < 102 > > 4
```

The most appropriate new patterns seem to be ones like these:

```
< 5 < 0 > < 2 > < 3 > >            'unified' new pattern
< 6 < 100 > < 5 > < 102 > >        'unmatched' new pattern
```

Notice that these omit the unmatched 'code' symbols '5', '6' and '7'.
Question: should these be included in the new patterns or should they
be included in separate 'code' patterns?

Here is a tentative rule: "Form a 'unified' pattern from the hit symbols
in the target pattern and also form an 'unmatched' new pattern out of the
unmatched CONTENTS symbols of the target pattern (row 4). The latter
should incorporate references that tie the 'unmatched' new pattern to
the unified pattern."

An alternative tentative rule is "Form a 'unified' pattern from the hit
symbols in the target pattern and also form an 'unmatched' new pattern
out of the unmatched CONTENTS symbols of the target pattern (row 4). The
former should incorporate references that tie the 'unmatched' new pattern
to the unified pattern."

Here are the patterns created according to the second rule:

```
< 5 < 100 > < 0 > < 2 > < 3 > < 102 > >            'unified' new pattern
< 6 < 100 > < 102 > >                              'unmatched' new pattern
```

The result in this case does not look very sensible! In fact, pattern '5'
has the same CONTENTS symbols as pattern '4'.

Here is another tentative rule:

"Form new patterns from each matched or unmatched *coherent* sequence of two
or more DATA symbols in the case of 'basic' patterns or two or more 'references'
in the case of 'abstract' patterns. Then create a new abstract pattern to tie
these elements together, including singletons left over from the
original patterns."

In this example, the result would be:

```
< 5 < 0 > < 2 > < 3 > >            'unified' new pattern
< 6 < 100 > < 5 > < 102 > >        'abstract' new pattern
```

which looks reasonable.

For an alignment like this:

```
a b c x d e f y g h i z j k l
      |       |       |
p q r x s t u y v w 1 z 2 3 4
```

the results would be:

```
< 5 #1 a b c >
< 5 #2 p q r >
< 6 #3 d e f >
< 6 #4 s t u >
< 7 #5 g h i >
< 7 #6 v w 1 >
< 8 #7 j k l >
< 8 #8 2 3 4 >
< 9 < 5 > x < 6 > y < 7 > z < 8 > >
```

OK, this looks reasonable except that there is a need to encode
specific sequences with code patterns as well. Question: should these
code patterns be formed when the abstract pattern is formed or at some
other time?

But what if the alignment was this:

```
a b c x d e f y g     z j k l
      |       |       |
p q r x s t u y v w 1 z 2 3 4
```

In this case, there seems to be a need to form 'g' into a pattern such
as '< 7 #5 g >' so that its place in the overall scheme can be marked.
This is despite the fact that it is a singleton DATA_SYMBOL.

For an alignment like this:

```
a x b
  |
p x q
```

the best result seems to be simply to increment the frequency value
of 'x'.

A possible alternative is:

```
< 1 #1 a >
< 1 #2 p >
< 2 #3 b >
< 2 #4 q >
< 3 < 1 > x < 2 > >
```

with code patterns as before.

With an alignment like this:

```
      x d e f y       z j k l
      |       |       |
p q r x       y v w 1 z
```

the best result may be something like this:

```
< 5 #1 >
< 5 #2 p q r >
< 6 #3 d e f >
< 6 #4 >
< 7 #5 >
< 7 #6 v w 1 >
< 8 #7 j k l >
< 8 #8 >
< 9 < 5 > x < 6 > y < 7 > z < 8 > >
```

Here is another tentative rule:

"1 From the matched target symbols, make a new pattern from each
sequence of two or more DATA_SYMBOLs or two or more 'references'.
2 Make a new pattern from every unmatched sequence of target symbols
and every corresponding sequence of New symbols, including singletons
and null patterns. 3 Make an abstract pattern from these elements."

Given this rule, there will probably also be a need for some kind
of cleaning up operation.

With this alignment:

```
0                   j o h n         l o v e s         m a r y           0
                    | | | |         | | | | |         | | | |
1            < 0 #5 j o h n >       | | | | |         | | | |           1
             | |           |       | | | | |         | | | |
2            | |           | < 2 #6 l o v e s >       | | | |           2
             | |           | | |             |       | | | |
3            | |           | | |             | < 3 #7 m a r y >         3
             | |           | | |             | | |           |
4 < 4 < 100 > < 0           > < 2             > < 3           > < 102 > > 4
```

the resulting patterns would be:

```
< 5 < 0 > < 2 > < 3 > >
< 6 #1 < 100 > >
< 6 #2 >
< 7 #3 < 102 > >
< 7 #4 >
```

```
< 8 < 6 > < 5 > < 7 > >
< 8 #1 #3 >     (code pattern)
< 8 #2 #4 >     (code pattern)
```

This looks pretty clumsy compared with:

```
< 5 < 0 > < 2 > < 3 > >            'unified' new pattern
< 6 < 100 > < 5 > < 102 > >        'abstract' new pattern
```

Here is another tentative rule:

"Make a new pattern from every coherent sequence of two or more matched DATA_SYMBOLs or 'references' in the target pattern and likewise for coherent unmatched sequences. Make a new pattern from every coherent sequence of one or more unmatched New symbols. Derive an abstract pattern from the resulting patterns. Include a code pattern for the New pattern."

In this case, the result would be two patterns just shown together with the code pattern '< 5 #5 #6 #7 >'.

%62 6/12/04

NEW START

Version 7.2 of SP71 will attempt to implement these ideas:

1 Learning will wait until has completed its recognition of Old patterns as far as possible.

2 During recognition, an alignment cannot proceed to the next cycle unless all the CONTENTS symbols of its Old patterns are fully matched.

[3 During recognition, a New pattern may be parsed into two or more *independent* Old patterns. During learning, these may be combined into an abstract pattern. This may be put on hold for the time being.]

4 Recognition is always between a driving New pattern or a driving alignment and an Old pattern, never between two alignments. This should facilitate learning (see next).

5 Because of 4, there will always be *one* Old pattern that is significant for learning. Because of 2, we can be sure that the *only* Old pattern that is not fully matched is the current target Old pattern (although in specific cases, it may be only the New pattern that is not fully matched). Thus learning is about the relationship between the New pattern and one Old pattern.

6 Given a partial match between one New pattern and one Old pattern, the system will create new Old patterns as follows:

"Make a new pattern from every coherent sequence of two or more matched CONTENTS symbols in the target pattern and likewise for coherent unmatched sequences. Make a new pattern
from every coherent sequence of one or more unmatched New symbols. Derive an abstract pattern from the resulting patterns. Include a code pattern for the New pattern."

%63 15/12/04

RESULTS FROM SP71, V 7.2

With input like this:

```
[
    [
        (j o h n r u n s)
    ]
    [
        (< !N !0 m a r y >)
        (< !N !1 j o h n >)
        (< !V !0 r u n s >)
        (< !V !1 w a l k s >)
        (< !S < N > < V > < Adv > >)
        (< !Adv !0 f a s t >)
        (< !Adv !1 s l o w l y >)
    ]
]
```

the program produces results like these:

```
Learning from alignment ID21:

0          j o h n        r u n s           0
           | | | |        | | | |
```

```
1     < N 1 j o h n >       | | | |            1
      | |           |       | | | |
2     | |           | < V 0 r u n s >          2
      | |           | | |           |
3 < S < N           > < V           > < Adv > > 3
```

Patterns:

ID33 (< 3 < N > < V > >)*1

ID34 (< 4 5 >)*1

ID35 (< 4 6 < Adv > >)*1

ID30 (< 7 < 3 > < 4 > >)*1

This seems to be basically 'correct' except that ID33 has the same frequency as ID30 from which it is referenced, which suggests that a better result might have been:

Patterns:

ID34 (< 4 5 >)*1

ID35 (< 4 6 < Adv > >)*1

ID30 (< 7 < N > < V > < 4 > >)*1

However, the result is interesting because IT SHOWS THAT V 7.2 OF SP71 CAN FORM INTERMEDIATE-LEVEL STRUCTURES, which in this example is the pattern ID33. This is what we have been aiming for.

With input like this:

```
[
    [
        (j o h n x y z r u n s)
    ]
    [
        (< !N !0 m a r y >)
        (< !N !1 j o h n >)
        (< !V !0 r u n s >)
        (< !V !1 w a l k s >)
        (< !S < N > < V > < Adv > >)
        (< !Adv !0 f a s t >)
        (< !Adv !1 s l o w l y >)
    ]
]
```

the program now produces results like this:

Learning from alignment ID22:

```
0           j o h n       x y z r u n s            0
            | | | |             | | | |
1     < N 1 j o h n >           | | | |            1
      | |           |           | | | |
2     | |           | < V 0     r u n s >          2
      | |           | | |               |
3 < S < N           > < V               > < Adv > > 3
```

Patterns:

ID34 (< 3 < N > >)*1

ID35 (< 4 5 x y z >)*1

ID36 (< 4 6 >)*1

ID37 (< 7 < V > >)*1

ID38 (< 8 9 >)*1

ID39 (< 8 10 < Adv > >)*1

ID31 (< 11 < 3 > < 4 > < 7 > < 8 > >)*1

This basically correct except that each of ID34 and ID36 is a reference to a reference and they both have the same frequency. A better result might have been something like:

ID35 (< 4 5 x y z >)*1

ID36 (< 4 6 >)*1

ID38 (< 8 9 >)*1

ID39 (< 8 10 < Adv > >)*1

ID31 (< 11 < N > < 4 > < V > < 8 > >)*1

In general, there may be a case for some post-processing of patterns
along these lines:

* Where a pattern has the same frequency as another pattern from which
it is referenced, then the reference may be replaced by the CONTENTS
symbols of the given pattern and the pattern may be deleted.

* Where the CONTENTS symbols of a pattern are merely a aingle reference,
then all references to that pattern may be replaced by the reference within
the pattern and the pattern may be deleted.

%64 17/12/04

FURTHER RESULTS FROM SP71, V 7.2

In the model as it stands now, an alignment like this:

ID42 : ID41 : ID24 : #64 NSC = 3226.47, EC = 32.00, CR = 100.83, CD = 3194.47,
Absolute P = 0.00

```
0         w e r u n        f a      s         t     0
          | | | | |        | |      |         |
1     < 3 w e r u n >      | |      |         |     1
      | |          |      | |      |         |
2     | |          | < 4 6 f a >    |         |     2
      | |          | | |      |     |         |
3     | |          | | |      | < 7 s >       |     3
      | |          | | |      | | |   |       |
4     | |          | | |      | | |   | < 8 10 t >   4
      | |          | | |      | | |   | | |     |
5 < 11 < 3         > < 4      > < 7   > < 8     > > 5
```

counts as 'FULL_A' because all the New symbols are matched and all the CONTENTS
symbols of all the Old symbols are matched.

But an alignment like this:

ID69 : ID68 : ID24 : #163 NSC = 3562.33, EC = 32.01, CR = 0.01, CD = 3530.32,
Absolute P = 0.00

```
0         w e r u n            s        l o w l y     0
          | | | | |            |        | | | | |
1     < 3 w e r u n >          |        | | | | |     1
      | |          |          |        | | | | |
2     | |          |      < 7 s >      | | | | |     2
      | |          |      | |   |      | | | | |
3     | |          |      | |   | < 8 9 l o w l y >   3
      | |          |      | |   | | |           |
4 < 11 < 3         > < 4 > < 7   > < 8           > > 4
```

only counts as 'FULL_B' because one of the Old symbols does not have all its
CONTENTS symbols matched. However, '< 4 >' in row 4 represents a class that
includes a NULL element. So there is a case for regarding it as NULL in
itself - which means that the whole alignment should, perhaps, be classified
as 'FULL_A'.

This issue may have a bearing on whether or not the program manages to find
'correct' grammars. At this stage, where the program is finding mainly 'bad'
patterns, the issue is probably not critical. It is probably too soon to
know whether this issue will matter at later stages, where the program is
finding 'good' patterns.

%65 17/12/04

An alternative to this:

Learning from alignment ID20:

```
0     w e r u n     s l o w l y 0
      | | | | |     |
1 < 1 w e r u n f a s t >       1
```

Patterns:

ID27 (< 3 w e r u n >)*1

```
ID28 (< 4 5 >)*1

ID29 (< 4 6 f a >)*1

ID30 (< 7 s >)*1

ID31 (< 8 9 l o w l y >)*1

ID32 (< 8 10 t >)*1

ID24 (< 11 < 3 > < 4 > < 7 > < 8 > >)*1

is this:

Learning from alignment ID20:

0     w e r u n     s l o w l y 0
      | | | | |     |
1 < 1 w e r u n f a s t >       1

Patterns:

ID27 (< 3 w e r u n >)*1

ID29 (< 4 f a >)*1

ID30 (< 7 s >)*1

ID31 (< 8 9 l o w l y >)*1

ID32 (< 8 10 t >)*1

ID24 (< 11 < 3 > < 4 > < 7 > < 8 > >)*1

ID25 (< 12 < 3 > < 7 > < 8 > >)*1

In other words, we can substitute 'direct' encoding for the use of 'null'
patterns such as ID28. It is not clear at this point which is best but we
can get round this problem by allowing the system to use *both* kinds
of encoding.

This issue has a bearing on whether a set of New patterns that contains
sentences like 'w e r u n f a s t' and 'w e r u n' should best be encoded
like this:

< 1 w e >
< 2 r u n >
< 3 f a s t >
< 3 >
< 4 < 1 > < 2 > < 3 > >

[17 symbols]

or like this:

< 1 w e >
< 2 r u n >
< 3 f a s t >
< 4 < 5 > < 3 > >
< 5 < 1 > < 2 > >

[18 symbols]

In this example, the number of symbols is very close. Which is best for a
particular set of sentence probably will depend on the exact frequency of
occurrence of patterns, sub-patterns and symbols. This should 'come out
in the wash' when the program is run.

Only the second example illustrates intermediate structures (in keeping
with the main goal of this phase of development).

Providing alternative encodings of null elements should solve the problem
described in %64: there should *always* be a 'FULL_A' alignment for each
New pattern in Phase 2.

This is a good place to start a new version of the program. The main goal
of version 7.3 will be:

* To provide both kinds of encoding of null elements as described above.

And these ideas (from %63) may also be developed:
```

* Where a pattern has the same frequency as another pattern from which
it is referenced, then the reference may be replaced by the CONTENTS
symbols of the given pattern and the pattern may be deleted.

* Where the CONTENTS symbols of a pattern are merely a aingle reference,
then all references to that pattern may be replaced by the reference within
the pattern and the pattern may be deleted.

%66 18/12/04

RESULTS FROM SP71, V 7.3

With additional abstract patterns as described in %65, we now get alignments like:

ID43 : ID42 : ID24 : #64 NSC = 3226.47, EC = 32.00, CR = 100.83, CD = 3194.47,
Absolute P = 0.00

```
0          w e r u n         f a       s          t     0
           | | | | |         | |       |          |
1      < 3 w e r u n >       | |       |          |     1
       | |           |       | |       |          |
2      | |           | < 4 6 f a >     |          |     2
       | |           | | |       |     |          |
3      | |           | | |       | < 7 s >        |     3
       | |           | | |       | | |   |        |
4      | |           | | |       | | |   | < 8 10 t >   4
       | |           | | |       | | |   | | |      |
5 < 11 < 3           > < 4       > < 7   > < 8      > > 5
```

and:

ID91 : ID90 : ID24 : #284 NSC = 3562.33, EC = 32.01, CR = 0.01, CD = 3530.32,
Absolute P = 0.00

```
0          w e r u n             s         l o w l y     0
           | | | | |             |         | | | | |
1      < 3 w e r u n >           |         | | | | |     1
       | |           |           |         | | | | |
2      | |           |       < 7 s >       | | | | |     2
       | |           |       | | |   |     | | | | |
3      | |           |       | | |   | < 8 9 l o w l y >   3
       | |           |       | | |   | | |             |
4 < 11 < 3           > < 4 > < 7   > < 8             > > 4
```

as before. But we also get alignments like this:

ID44 : ID42 : ID25 : #73 NSC = 3226.47, EC = 40.00, CR = 80.66, CD = 3186.47,
Absolute P = 0.00

```
0          w e r u n         f a       s          t     0
           | | | | |         | |       |          |
1      < 3 w e r u n >       | |       |          |     1
       | |           |       | |       |          |
2      | |           | < 4 6 f a >     |          |     2
       | |           |                 |          |
3      | |           |             < 7 s >        |     3
       | |           |             | | |   |      |
4      | |           |             | | |   | < 8 10 t >   4
       | |           |             | | |   | | |      |
5 < 12 < 3           >             < 7   > < 8      > > 5
```

and this:

ID92 : ID90 : ID25 : #293 NSC = 3968.87, EC = 24.00, CR = 165.37, CD = 3944.87,
Absolute P = 0.00

```
0          w e r u n       s         l o w l y     0
           | | | | |       |         | | | | |
1      < 3 w e r u n >     |         | | | | |     1
       | |           |     |         | | | | |
2      | |           | < 7 s >       | | | | |     2
       | |           | | |   |       | | | | |
3      | |           | | |   | < 8 9 l o w l y >   3
       | |           | | |   | | |             |
4 < 12 < 3           > < 7   > < 8             > > 4
```

ID92 solves the previous problem but we get the seamingly anomalous alignment ID44.

%67 20/12/04

FURTHER NOTES

Alignment ID44 (above) is anomalalous in the sense that it provides full

matching for all the New symbols and all the Old symbols but an encoding as '12 4' would not be decodable because the position of '4' relative to the rest of the alignment is not defined. It looks as if alignments like this should be discarded as 'illegal'. But it is not entirely clear how one might recognise such alignments as illegal.

Meanwhile, we seem to need another version of SP71. V 7.4 will not create two abstract patterns when there are null elements (as in v 7.3) but will create new abstract patterns when required.

The problem arises from alignments like this:

```
ID69 : ID68 : ID24 : #163 NSC = 3562.33, EC = 32.01, CR = 0.01, CD = 3530.32,
Absolute P = 0.00

0         w e r u n            s        l o w l y    0
          | | | | |            |        | | | | |
1     < 3 w e r u n >          |        | | | | |    1
      | |          |          |        | | | | |
2     | |          |      < 7 s >      | | | | |    2
      | |          |      | |   |      | | | | |
3     | |          |      | |   | < 8 9 l o w l y >  3
      | |          |      | |   | | |          |
4 < 11 < 3          > < 4 > < 7   > < 8           > > 4
```

This is formed in Phase 2, which comes after the learning process in Phase 1 and at a stage when there should be at least one FULL_A alignment for each New pattern.

If an alignment like this were to be deemeed legal, there would still be a problem because the null pattern '< 4 5 >' would be assigned zero frequency so it would not appear in any grammar created by the program.

Another problem is that, if null patterns are preserved in some way, there will be a proliferation of null patterns for all the different contexts in which they may occur - unless there is some mechanism for recognising that they are all the same and assigning them all to the same class. This would have the effect of merging all classes containing a null pattern - and this might be a generalisation too far.

It looks as if null patterns may create more problems than they solve. For the time being, the program will be developed in such a way that it never forms null patterns and the apparent need for a null pattern is met along the lines of v 7.3.

When there is an apparent need for null patterns, V 7.4 will create two versions of the abstract pattern (as in v 7.3) but will not create any null patterns. Also, a substructure like '< 7 > < 8 >' in row 4 of ID69 will be formed into a discrete pattern and the ID-symbol for that pattern will be used in the two versions of the abstract pattern.

Apart from the last feature, the program has been modified as described and is now producing alignments like these:

```
ID43: NSC = 3226.47, EC = 24.00, CR = 134.44, CD = 3202.47,
Absolute P = 5.96046447754e-008

0        w e r u n      f a      s        t    0
         | | | | |      | |      |        |
1    < 3 w e r u n >    | |      |        |    1
     | |          |     | |      |        |
2    | |          | < 4 f a >    |        |    2
     | |          | | |    |     |        |
3    | |          | | |    | < 5 s >      |    3
     | |          | | |    | | |   |      |
4    | |          | | |    | | |   | < 6 8 t >   4
     | |          | | |    | | |   | | |    |
5 < 9 < 3          > < 4    > < 5   > < 6    > > 5
```

and

```
ID92: NSC = 3968.87, EC = 24.00, CR = 165.37, CD = 3944.87,
Absolute P = 5.96046447754e-008

0         w e r u n      s        l o w l y    0
          | | | | |      |        | | | | |
1     < 3 w e r u n >    |        | | | | |    1
      | |          |     |        | | | | |
2     | |          | < 5 s >      | | | | |    2
      | |          | | |   |      | | | | |
3     | |          | | |   | < 6 7 l o w l y >  3
      | |          | | |   | | |            |
4 < 10 < 3          > < 5   > < 6            > > 4
```

This is in line with current thinking.

However, the program is also producing anomalous alignments like these:

```
ID44: NSC = 3226.47, EC = 32.00, CR = 100.83, CD = 3194.47,
Absolute P = 2.32830643654e-010

0         w e r u n      f a      s        t    0
          | | | | |      | |      |        |
1     < 3 w e r u n >    | |      |        |    1
      | |          |    | |      |        |
2     | |          | < 4 f a >    |        |    2
      | |          |              |        |
3     | |          |          < 5 s >      |    3
      | |          |          | |   |      |
4     | |          |          | |   | < 6 8 t >   4
      | |          |          | |   | | |    |
5 < 10 < 3         >          < 5   > < 6    > > 5

and

ID52: NSC = 3226.47, EC = 32.00, CR = 100.83, CD = 3194.47,
Absolute P = 2.32830643654e-010

0             w e r u n      f a      s        t      0
              | | | | |      | |      |        |
1         < 3 w e r u n >    | |      |        |      1
          | |          |    | |      |        |
2         | |          | < 4 f a >    |        |      2
          | |          | | |    |    |        |
3         | |          | | |    | < 5 s >      |      3
          | |          | | |    | | |  |        |
4         | |          | | |    | | |  | < 6 8 t >    4
          | |          | | |    | | |  | | |    |
5 < 10    < 3          > | |    | < 5  > < 6    > >   5
          | |          | | |    | | |  | | |    |
6     < 9 < 3          > < 4    > < 5  > < 6    >   > 6

and

ID54: NSC = 3226.47, EC = 32.00, CR = 100.83, CD = 3194.47,
Absolute P = 2.32830643654e-010

0             w e r u n      f a      s        t      0
              | | | | |      | |      |        |
1         < 3 w e r u n >    | |      |        |      1
          | |          |    | |      |        |
2         | |          | < 4 f a >    |        |      2
          | |          | | |    |    |        |
3         | |          | | |    | < 5 s >      |      3
          | |          | | |    | | |  |        |
4         | |          | | |    | | |  | < 6 8 t >    4
          | |          | | |    | | |  | | |    |
5 < 9     < 3          > < 4    > < 5  > < 6    > >   5
          | |          |          | |  | | |    |
6    < 10 < 3          >          < 5  > < 6    >   > 6

It is not clear at present what should be done about these.

There is also a FULL_B alignment like this:

ID91: NSC = 3562.33, EC = 32.01, CR = 0.01, CD = 3530.32,
Absolute P = 2.31305015528e-010

0        w e r u n           s        l o w l y    0
         | | | | |           |        | | | | |
1    < 3 w e r u n >         |        | | | | |    1
     | |          |         |        | | | | |
2    | |          |      < 5 s >      | | | | |    2
     | |          |      | |  |      | | | | |
3    | |          |      | |  | < 6 7 l o w l y >  3
     | |          |      | |  | | |           |
4 < 9 < 3         > < 4 > < 5  > < 6           > > 4

This is eliminated from Phase 2 in the normal way.

Alignments like ID52 and ID54 may be ruled out if there is a rule that
ID-symbols must be matched to C-symbols and vice versa. This can be
made optional (as in SP62).

%68 21/12/04
```

FURTHER NOTES

A switch has been added (ID_C_SYMBOL_CONSTRAINT) which, when ON, ensures
that matches are only between symbols with different status. For this to
work, it is necessary that all the symbols in all New patterns should be
given the status of IDENTIFICATION symbols.

With the switch ON, all alignments like ID52 and ID54 (above) are
eliminated. But 'illegal' alignments like ID44 remain.

An alignment like this:

ID44 : ID42 : ID25 : #73 NSC = 3226.47, EC = 32.00, CR = 100.83, CD = 3194.47,
Absolute P = 0.00

```
0         w e r u n      f a      s        t    0
          | | | | |      | |      |        |
1     < 3 w e r u n >    | |      |        |    1
      | |          |    | |      |        |
2     | |          | < 4 f a >    |        |    2
      | |          |              |        |
3     | |          |          < 5 s >      |    3
      | |          |          | |   |      |
4     | |          |          | |   | < 6 8 t >   4
      | |          |          | |   | | |    |
5 < 10 < 3         >          < 5   > < 6    > > 5
```

leads to a spuriously high count of 2 for the pattern in row 5 because
the same pattern fits the sentence 'w e r u n s l o w l y'. This is a very
good reason for ruling out alignments like this as 'illegal'. It is not
immediately clear how one can test for this kind of alignment.

%69 22/12/04

FURTHER NOTES ON SP71, V 7.4

Alignments like ID44, above, can be tested for by the following rule: "There
should be only one LEFT_BRACKET with the status IDENTIFICATION and one
RIGHT_BRACKET with the same status. Any alignment that does not fit this
rule is illegal."

As I write this, I can see that this rule is nonsense! It immediately
rules out alignments created by combining smaller alignments.

What is wrong with ID44 is that it tries to mix two kinds of encoding:
1 encoding New as a succession of smaller patterns and 2 encoding New as an
abstract pattern with 'corrections' or 'additions'. So now we may make a
new stab at defining a rule that would identify such anomalous alignments:

"Given that BRACKET symbols may be embedded within an alignment, ID-symbols
are only permitted at level 0."

According to this rule, alignments produced by combining smaller alignments
would be legal but alignments like ID44 would be illegal.

This has been completed satisfactorily and seems to work as intended.

The next step is to make discrete patterns from any sequence of two or
more 'references', as described in %67.

%70 23/12/04

FURTHER NOTES ON SP71, V 7.4

With regard to making discrete sub-patterns from any sequence of two or more
references (as above), it looks as if the most straightforward way is by essentially
the same mechanism as is used for other learning. With an alignment like this:

ID69 : ID68 : ID24 : #175 NSC = 3562.33, EC = 32.01, CR = 0.01, CD = 3530.32,
Absolute P = 0.00

```
0        w e r u n           s        l o w l y    0
         | | | | |           |        | | | | |
1    < 3 w e r u n >         |        | | | | |    1
     | |          |          |        | | | | |
2    | |          |      < 5 s >      | | | | |    2
     | |          |      | |   |      | | | | |
3    | |          |      | |   | < 6 7 l o w l y >  3
     | |          |      | |   | | |           |
4 < 9 < 3         > < 4 > < 5   > < 6           > > 4
```

the system should reprocess the pattern in row 4 to become something like:

```
< 20 < 5 > < 6 > >
< 21 < 3 > < 20 > >
```

This is because ID69 is a partial alignment and any partial alignment should be a signal for learning.

The trouble with this idea is that an alignment like ID69 is only formed in Phase 2 and learning occurs in Phase 1!

%71 24/12/04

FURTHER NOTES ON SP71, V 7.4

The program now forms 'correct' alignments with intermediate structures like these:

ID53 : ID46 : ID25 : #98 NSC = 3226.47, EC = 24.00, CR = 134.44, CD = 3202.47, Absolute P = 0.00

```
0          w e r u n       f a           s         t       0
           | | | | |       | |           |         |
1      < 3 w e r u n >     | |           |         |       1
       | |           |     | |           |         |
2      | |           | < 4 f a >         |         |       2
       | |           | | |     |         |         |
3      | |           | | |     |     < 5 s >       |       3
       | |           | | |     |     | |   |       |
4      | |           | | |     |     | |   | < 6 8 t >     4
       | |           | | |     |     | |   | | |     |
5      | |           | | |     | < 9 < 5   > < 6     > >   5
       | |           | | |     | | |                   |
6 < 11 < 3           > < 4     > < 9                   > > 6
```

and

ID79 : ID72 : ID25 : #189 NSC = 3537.27, EC = 32.01, CR = 0.01, CD = 3505.26, Absolute P = 0.00

```
0          w e r u n               s         l o w l y       0
           | | | | |               |         | | | | |
1      < 3 w e r u n >             |         | | | | |       1
       | |           |             |         | | | | |
2      | |           |           < 5 s >       | | | | |       2
       | |           |           | |   |       | | | | |
3      | |           |           | |   | < 6 7 l o w l y >     3
       | |           |           | |   | | |             |
4      | |           |       < 9 < 5   > < 6             > >   4
       | |           |       | |                         |
5 < 11 < 3           > < 4 > < 9                         > > 5
```

and

ID80 : ID72 : ID24 : #183 NSC = 3968.87, EC = 24.00, CR = 165.37, CD = 3944.87, Absolute P = 0.00

```
0          w e r u n           s         l o w l y       0
           | | | | |           |         | | | | |
1      < 3 w e r u n >         |         | | | | |       1
       | |           |         |         | | | | |
2      | |           |     < 5 s >       | | | | |       2
       | |           |     | |   |       | | | | |
3      | |           |     | |   | < 6 7 l o w l y >     3
       | |           |     | |   | | |             |
4      | |           | < 9 < 5   > < 6             > >   4
       | |           | | |                           |
5 < 10 < 3           > < 9                           > > 5
```

However, there seems to be problems in grammar formation that need sorting out.

%72 28/12/04

FURTHER NOTES ON SP71, V 7.4

Many of the anomalies have been ironed out but the program is tending to produce grammars comprising patterns which are the same as the original sentences with ID-symbols added. To try to understand what is going on, the program is tested with this input:

```
[
    [
        (w e r u n f a s t)
        (w e w a l k f a s t)
    ]
```

```
    [
    ]
]

The best grammars in this case are:

GRAMMAR GR13

Grammar GR13, derived from GR5, G = 6113.71, E = 69.19, score = 6182.90:

Grammar patterns:

ID80 (< 3 w e >)*2

ID81 (< 4 6 r u n >)*1

ID82 (< 7 f a s t >)*2

ID102 (< 4 5 w a l k >)*1

GRAMMAR GR11

Grammar GR11, derived from GR4, G = 6152.22, E = 33.68, score = 6185.90:

Grammar patterns:

ID76 (< 3 w e >)*2

ID77 (< 4 6 r u n >)*1

ID78 (< 7 f a s t >)*2

ID79 (< 8 < 3 > < 4 > < 7 > >)*2

ID95 (< 4 5 w a l k >)*1

GRAMMAR GR10

Grammar GR10, derived from GR5, G = 6152.22, E = 51.43, score = 6203.65:

Grammar patterns:

ID80 (< 3 w e >)*2

ID81 (< 4 6 r u n >)*1

ID82 (< 7 f a s t >)*2

ID91 (< 4 5 w a l k >)*1

ID93 (< 8 < 3 > < 4 > < 7 > >)*2

GRAMMAR GR14

Grammar GR14, derived from GR4, G = 6152.22, E = 51.43, score = 6203.65:

Grammar patterns:

ID76 (< 3 w e >)*2

ID77 (< 4 6 r u n >)*1

ID78 (< 7 f a s t >)*2

ID79 (< 8 < 3 > < 4 > < 7 > >)*2

ID105 (< 4 5 w a l k >)*1

GRAMMAR GR6

Grammar GR6, derived from GR3, G = 8644.70, E = 22.76, score = 8667.46:

Grammar patterns:

ID75 (< 1 w e r u n f a s t >)*1

ID83 (< 2 w e w a l k f a s t >)*1

Comments:

1 Grammar GR6 has a higher (worse) score than the 'encoded' grammars,
in accordance with our intuitions.
```

2 GR11, which has an abstract pattern, has a higher (worse) score
than GR13, in contradiction of our intuitions.

3 GR11, GR10, and GR14 are identical in terms of the sequences of symbols
in the patterns. There is a case for checking for duplicate grammars and
removing them.

4 All three of those grammars have a G value of 6152.22, but GR11 has an
E value of 33.68 whereas GR10 and GR14 have E values of 51.43. It is not
clear why there should be these differences in E values and, consequently,
in compression scores.

5 G values are very much higher than E values which means they have a
disproportionate effect on overall score. This is probably because DATA
symbols are given the bit costs that they have in New patterns, whereas
they should probably have much lower bit costs in Old patterns.

A new version of SP71 will be started (v 7.5) to try to get to the bottom
of these points and correct any errors that there may be in the program.

%73 30/12/04

NOTES ON SP71, V 7.5

The apparent anomalies in scoring noted in %72 have appeared again
as follows:

GR11 (ID13, 18, 19, 20, 21)

G = 149.58
E = 33.68
S = 183.25

GR10 (ID13, 18, 19, 20, 21)

G = 149.58
E = 51.43
S = 201.01

GR14 (ID13, 18, 19, 20, 21)

G = 149.58
E = 51.43
S = 201.01

The anomaly apparently arises because of the sequence in which each grammar
is created and the alignments they are derived from:

GR4 (ID34, full) -> GR14 (ID59, composite)

GR4 (ID34, full) -> GR11 (ID61, full)

GR5 (ID33, composite) -> GR10 (ID61, full)

The alignments marked 'full' are those that have an abstract pattern to tie
the elements together. This abstract pattern means that their encoding costs
are lower. The alignments marked 'composite' are simply a concatenation of
elements without an abstract pattern to tie them together and their encoding
costs are higher.

GR14 and GR10 are each derived from one composite alignment and one full
alignment and their E and S values are relatively high. Grammar GR11 is
derived from two full alignments which means that its E and S values are lower.

So the varying E and S values for identical grammars is 'correct' and in
accordance with how the program was intended to operate.

Since, in general, we don't want duplicate grammars to be produced, it looks
as if the next step to take is to compare each new grammar with those that
have been produced before and, where two grammars are found to be identical,
to delete one of them. If two identical grammars have different scores, then
the one to delete is the one with the worst score. This feature will be
programmed into SP71, v 7.5.

This has been done and it seems to work.

Now most of the points from %73 have been covered except that the program
is tending to favour grammars consisting of whole sentences instead of
ones where words are encoded in an abstract sentence pattern. This seems
to be a matter of the 'weights' of different kinds of symbol and this
needs more thought.

If the program is going to recognise intermediate structure, it should

be able to recognise that the first sentence in the example here:

```
j o h n r u n s
j o h n r u n s f a s t
```

is an element within the longer sentence. And it should not matter in what order the sentences are presented. This will make our next test problem.

%74 31/12/04

FURTHER NOTES ON SP71, V 7.5

With this input:

```
[
    [
       (j o h n r u n s)
       (j o h n r u n s f a s t)
    ]
    [
    ]
]
```

The program forms this alignment (as expected):

```
ID42: NSC = 3366.46, EC = 10.76, CR = 312.81, CD = 3355.70,
Absolute P = 0.000575891732354

0     j o h n r u n s f a s t 0
      | | | | | | | |
1 < 1 j o h n r u n s >       1
```

but the one from which learning is done is:

```
ID66: NSC = 3369.62, EC = 27.54, CR = 0.01, CD = 3342.08,
Absolute P = 5.13528168354e-009

0     j o h n r u n s f a                   s   t 0
      | | | | | | | |                     |
1 < 1 j o h n r u n s >                     |     1
                                            |
2                                     < 6 7 s >   2
                                      | |     |
3                     < 9 < 2 > < 5 > < 6     > > 3
```

This is odd because ID42 has a higher CD score. This needs looking at.

The problem is that ID42 is being classified as a FULL_B alignment and, for that reason, is not a candidate for learning!

Now that the program is corrected so that learning can be done from PARTIAL alignments or from FULL_B alignments, the best alignment is:

```
ID59: NSC = 3805.69, EC = 25.92, CR = 146.85, CD = 3779.78,
Absolute P = 1.57929184473e-008

0     j o h n r u n s f a     s t 0
      | | | | | | | |         |
1 < 1 j o h n r u n s >       |   1
                              |
2                     < 6 7 s > 2
```

The NSC for ID42 is 3366.46 but the NSC for ID59 is 3805.69 and the difference (439.23) is accounted for by bit_cost of the extra 's'.

Learning with ID59 leads to an error, possibly because the learning method is not designed to work with composite alignments. The program will be modified so that this kind of alignment is marked and excluded from the learning process.

At present, the program produces results like this:

```
Learning from alignment ID41:

0     j o h n r u n s a w a y 0
      | | | | | | | |
1 < 1 j o h n r u n s >       1

Patterns:

ID73 (< 12 j o h n r u n s >)*1

ID74 (< 13 a w a y >)*1
```

```
ID76 (< 14 < 12 > < 13 > >)*1

ID68 (< 15 < 14 > >)*1
```

This is basically 'correct' except for two points:

* There is a need to recognise that the contents symbols of '< 1 j o h n r u n s >' and '< 12 j o h n r u n s >' are the same. In general, it looks as if there is a need to check that a pattern has not already been created before giving it ID-symbols and adding it to Old.

* ID68 should not be produced at all.

Further development should attend to these points.

%75 4/1/05

FURTHER NOTES ON SP71, V 7.5

With this input:

```
[
    [
       (j o h n r u n s)
       (j o h n r u n s a w a y)
    ]
    [
    ]
]
```

the program now learns new patterns like this:

CONTENTS symbols of pattern ID69 match those of
pre-existing pattern ID3 (< 1 j o h n r u n s >)*1

Learning from alignment ID39:

```
0     j o h n r u n s a w a y 0
      | | | | | | | |
1 < 1 j o h n r u n s >       1
```

Patterns:

```
ID70 (< 11 12 a w a y >)*1

ID72 (< 13 < 1 > < 11 > >)*1
```

Here, ID3 was created as a 'receptacle' pattern from the first New pattern and then, when the program examines ID39, it discovers that the hit sequence in the alignment matches the CONTENTS symbols of ID3, so it deletes the newly-created pattern representing the hit sequence and uses ID3 instead.

Question: What happens if the New patterns are delivered in the reverse order?

With this input:

```
[
    [
       (j o h n r u n s a w a y)
       (j o h n r u n s)
    ]
    [
    ]
]
```

the program learns new patterns like this:

CONTENTS symbols of pattern ID78 match those of
pre-existing pattern ID59 (< 14 j o h n r u n s >)*1

Learning from alignment ID60:

```
0     j o h n r u n s           0
      | | | | | | | |
1 < 1 j o h n r u n s a w a y > 1
```

Patterns:

```
ID80 (< 15 16 a w a y >)*1

ID81 (< 17 < 14 > < 15 > >)*1
```

This is essentially the same as before except that ID numbers and ID-symbols
are different.

%76 4/1/05

FURTHER NOTES ON SP71, V 7.5

There seems to be a case for returning to the earlier coding scheme where
all DATA_SYMBOLS (symbol types that are found in the New patterns) have a
higher bit_cost than all other symbols, regardless of whether they are in
New patterns or Old patterns. This would provide an incentive for economising
on such symbols amongst the Old patterns and would thus favour grammars
that are 'coded' rather than grammars that are simply a copy of the New
patterns or something close to that.

It is true that an Old symbol that matches a New symbol may be seen as a
relatively small 'code' for that symbol but this is really a red herring
in the context of grammar learning. In order to explore the coding issues
that are the current focus of interest, it is probably best to return to
the previous scheme for the weighting of DATA_SYMBOLs and this will be done.

To get a tighter grip on coding issues, we shall go back to this simple
example:

```
[
    [(j o h n r u n s)
    (j o h n w a l k s)
    (m a r y r u n s)
    (m a r y w a l k s)
    ]
    [
    ]
]
```

At present, the best grammar obtained is:

Grammar GR177, derived from grammar GR147 and alignment ID444,
G = 8111.66, E = 119.57, score = 8231.23:

Grammar patterns:

ID15 (< 6 8 r u n s >)*2

ID197 (< 17 18 m a r y >)*2

ID198 (< 17 19 j o h n >)*2

ID335 (< 25 w a l k s >)*2

which raises questions like:

* Why is an abstract pattern not useful.

* What about the terminal 's' on 'runs' and 'walks'?

* Why are 'runs' and 'walks' not assigned to the same class?

A possible reason for this poor best grammar is that the EXTRACTION_LIMIT
is currently set to 1. When it is set to 5, intermediate results include:

Learning from alignment ID49:

```
0          j o h      n       w a l k s     0
           | | |      |             |
1     < 2 4 j o h >    |             |     1
      | |        |    |             |
2     | |        | < 5 n >           |     2
      | |        | | |   |           |
3     | |        | | |   | < 6 7     s >   3
      | |        | | |   | | |         |
4 < 9 < 2        > < 5   > < 6         > > 4
```

Patterns:

ID59 (< 11 < 2 > < 5 > >)*1

ID60 (< 12 w a l k >)*1

ID62 (< 13 < 6 > >)*1

ID54 (< 14 < 11 > < 12 > < 13 > >)*1

```
ID55 (< 15 < 11 > < 13 > >)*1

The main anomaly here is the creation of ID62. It contains a reference to a
reference and this is unnecessary. The program will be modified to try to
detect this kind of construction and avoid it.

%77

FURTHER NOTES ON SP71, V 7.5

With this input:

[
    [(j o h n r u n s)
    (j o h n w a l k s)
    (m a r y r u n s)
    (m a r y w a l k s)
    ]
    [
    ]
]

the program currently produces learned patterns like this:

CONTENTS symbols of pattern ID71 match those of
pre-existing pattern ID60 (< 12 w a l k >)*1

CONTENTS symbols of pattern ID73 match those of
pre-existing pattern ID14 (< 6 7 s >)*1

Learning from alignment ID42:

0     j o h n w a l k s   0
      | | | |         |
1 < 1 j o h n r u n   s > 1

Patterns:

ID70 (< 15 j o h n >)*1

ID72 (< 16 18 r u n >)*1

ID65 (< 19 < 15 > < 16 > < 6 > >)*1

The main problem here is that 'r u n' has been put into a different class
from the pre-existing pattern 'w a l k'.

This looks like a good opportunity to rationalise the current unsatisfactory
system of 'class symbols' and 'discrimination symbols'. Here is a tentative
scheme:

* 'Class symbols' will be 'context symbols' and one such symbol type will
be assigned to each context.

* 'Discrimination symbols' will be unique pattern identifiers and there
will be one for each pattern.

* When a pre-existing pattern is found as above then the context symbol for
the current context will be added to that pattern.

* When grammars have been created, there will be a cleaning up of unnecessary
ID-symbols, as in previous versions of the SP70/71 series.

These changes will be attempted in SP71, v 7.6.

%78 6/1/05

NOTES ON SP71, V 7.6

With this input:

[
    [(j o h n r u n s)
    (j o h n w a l k s)
    (m a r y r u n s)
    (m a r y w a l k s)
    ]
    [
    ]
]

The best two grammars (before cleaning up) are:
```

GRAMMAR GR214

Grammar GR214, derived from grammar GR174 and alignment ID947,
G = 26637.34, E = 349.88, score = 26987.22:

Grammar patterns:

ID15 (< 26 24 4 id6 r u n s >)*2

ID70 (< 46 35 23 17 9 id13 j o h n >)*2

ID96 (< 51 49 47 18 id21 w a l k s >)*2

ID339 (< 50 48 46 42 40 33 25 23 id28 m a r y >)*2

GRAMMAR GR215

Grammar GR215, derived from grammar GR165 and alignment ID947,
G = 26671.48, E = 330.57, score = 27002.04:

Grammar patterns:

ID15 (< 26 24 4 id6 r u n s >)*2

ID70 (< 46 35 23 17 9 id13 j o h n >)*2

ID96 (< 51 49 47 18 id21 w a l k s >)*2

ID336 (< id30 < 23 > < 24 > >)*2

ID339 (< 50 48 46 42 40 33 25 23 id28 m a r y >)*2

There are two problems here:

* The abstract pattern (ID336) does not cover ID96.

* The system is missing the grammars that recognise the terminal 's'
in all sentences.

The results will be examined to see why these anomalies are occurring.

After the two sentences 'j o h n r u n s' and 'j o h n w a l k s',
the best two grammars are:

GRAMMAR GR74

Grammar GR74, derived from grammar GR13 and alignment ID238,
G = 19137.48, E = 159.15, score = 19296.62:

Grammar patterns:

ID14 (< 16 11 4 id5 s >)*2

ID60 (< 15 10 7 id10 w a l k >)*1

ID70 (< 17 9 id13 j o h n >)*2

ID72 (< 10 id15 r u n >)*1

GRAMMAR GR59

Grammar GR59, derived from grammar GR9 and alignment ID243,
G = 19181.28, E = 118.34, score = 19299.62:

Grammar patterns:

ID14 (< 16 11 4 id5 s >)*2

ID60 (< 15 10 7 id10 w a l k >)*1

ID65 (< id16 < 9 > < 10 > < 11 > >)*2

ID70 (< 17 9 id13 j o h n >)*2

ID72 (< 10 id15 r u n >)*1

In both of them the terminal 's' is recognised.

After 'm a r y r u n s' has been processed, the best two grammars are:

GRAMMAR GR123

Grammar GR123, derived from grammar GR95 and alignment ID593,

```
G = 26513.52, E = 182.51, score = 26696.03:

Grammar patterns:

ID15 (< 26 24 4 id6 r u n s >)*2

ID70 (< 35 23 17 9 id13 j o h n >)*2

ID96 (< 18 id21 w a l k s >)*1

ID339 (< 33 25 23 id28 m a r y >)*1

GRAMMAR GR114

Grammar GR114, derived from grammar GR96 and alignment ID598,
G = 26546.62, E = 163.54, score = 26710.16:

Grammar patterns:

ID15 (< 26 24 4 id6 r u n s >)*2

ID70 (< 35 23 17 9 id13 j o h n >)*2

ID96 (< 18 id21 w a l k s >)*1

ID336 (< id30 < 23 > < 24 > >)*2

ID339 (< 33 25 23 id28 m a r y >)*1

This seems to be where the rot sets in. It may be something to do with
the fact that, a present, the program does not have an ability to
recognise that if an unmatched portion of New occurs 'opposite' a
reference to a disjunctive class, then it may be assigned to that class.

%79 7/1/05

FURTHER NOTES ON SP71, V 7.6

A revision of the learning method has produced this result:

Learning from alignment ID6:

0 j o h n r u n s         0
              |
1 < id1 j o h n r u n s > 1

Patterns:

ID11 (< 2 id2 j o h n r u >)*1

ID12 (< 2 id3 j o h >)*1

ID13 (< 3 id4 n >)*1

ID14 (< 4 id5 s >)*1

ID15 (< 4 id6 r u n s >)*1

ID16 (< 5 id7 < 3 > < 4 > >)*1

ID8 (< id8 < 2 > < 5 > >)*1

This is OK but it is slightly odd that ID16 is recognised as a subsequence
of ID8. As a result of having these patterns in Old, we later get
alignments like this:

NEW ALIGNMENT (PATTERN ID2, CYCLE 3, PHASE 1)

ID59 : ID54 : ID8 : #141 NSC = 6435.02, EC = 34.99, CR = 183.93, CD = 6400.03,
Absolute P = 0.00

0              j o h                 n       w a l k s       0
               | | |                 |               |
1       < 2 id3 j o h >              |               |       1
        | |           |              |               |
2       | |           |         < 3 id4 n >          |       2
        | |           |         | |       |          |
3       | |           |         | |       | < 4 id5     s >     3
        | |           |         | |       | | |           |
4       | |           | < 5 id7 < 3       > < 4           > >   4
        | |           | | |                                 |
5 < id8 < 2           > < 5                                 > > 5
```

Tracing back how this was built, ID54 is:

NEW ALIGNMENT (PATTERN ID2, CYCLE 2, PHASE 1)

ID54 : ID53 : ID16 : #131 NSC = 6435.02, EC = 43.30, CR = 148.62, CD = 6391.72,
Absolute P = 0.00

```
0         j o h                   n       w a l k s     0
          | | |                   |               |
1 < 2 id3 j o h >                 |               |     1
                                  |               |
2                         < 3 id4 n >             |     2
                          | |       |             |
3                         | |       | < 4 id5     s >   3
                          | |       | | |           |
4                 < 5 id7 < 3       > < 4           > > 4
```

and ID53 is:

COMPOSITE ALIGNMENT (PATTERN ID2, CYCLE 1, PHASE 1)

ID53: NSC = 6435.02, EC = 46.59, CR = 138.12, CD = 6388.43,
Absolute P = 9.44930139943e-015

```
0         j o h           n w a l k   s   0
          | | |           |           |
1 < 2 id3 j o h >         |           |   1
                          |           |
2                 < 3 id4 n >         |   2
                                      |
3                             < 4 id5 s > 3
```

All these alignments are classified as FULL_B meaning that all the
CONTENTS symbols in all their Old patterns are fully matched, although
the New symbols need not all be matched. In terms of the current rules
for building alignments and for learning, these alignments are OK. But ID59
is still anomalous for the purposes of learning because the unmatched symbols
in the New pattern cannot be related in an unambiguous way to the sequence of
symbols in the bottom row of the alignment (the last row to be added to the
alignment).

So far, it has been assumed that learning of new patterns could be
achieved via the New pattern and *one* Old pattern. This has a pleasing
simplicity that should, in principle, scale up to the learning of
arbitrarily complex structures. But it does not work for an alignment
like ID59. Here are some possible answers:

1 Create a single Old pattern by unifying all the Old patterns in a
given alignment. Learning of new patterns can be done from the New pattern
and this 'derived' Old pattern. An attraction of this option is that it would
allow us to return to 'basic' parsing without the formation of composite
alignments (which have never seemed entirely satisfactory).

2 Identify the 'highest' pattern in an alignment where a mismatch with the
New pattern occurs. Then do learning from the New pattern and the Old pattern
that has been identified. A possible difficulty here is that it may not
always be clear which level is mismatched.

3 Allow learning from a composite alignment like ID53. This probably means
creating a single Old pattern from the separate Old patterns, much as in 1.

4 Stop parsing as soon as a 'mismatch' occurs and do learning at that point.
With the examples above, this would mean parsing would stop with alignment
ID54 and learning would focus on that part of the New pattern not covered by
'< 2 id3 j o h >' and the Old pattern in row 4. An apparent attraction of
this option is that it would allow the system to find new structure within
structures that are less than the length of a New pattern. As before, it may
be difficult to define 'mismatch'.

The first possibility will be tried in SP71, v 7.7.

%80 11/1/05

NOTES ON SP71, V 7.7

The following output shows programming errors:

Learning from alignment ID59:

```
0             j o h                 n       w a l k s       0
              | | |                 |               |
1       < 2 id3 j o h >             |               |       1
        | |           |             |               |
```

```
2       | |           |         < 3 id4 n >             |       2
        | |           |         | |       |            |
3       | |           |         | |       | < 4 id5    s >     3
        | |           |         | |       | | |          |
4       | |           | < 5 id7 < 3       > < 4          > >   4
        | |           | | |                              |
5 < id8 < 2           > < 5                              > > 5
```

Patterns:

ID66 (< 6 id10 < 2 j o h > < 5 < 3 n > < 4 s >)*1

ID67 (< 7 id11 w a l k >)*1

ID69 (< 8 id12 < 6 > < 7 > >)*1

ID70 (< 9 id13 < 4 s >)*1

ID71 (< 10 id14 w a l k s >)*1

ID72 (< 10 id15 > > >)*1

ID73 (< 11 id16 < 6 > < 7 > < 9 > < 10 > >)*1

ID61 (< id17 < 8 > < 11 > >)*1

But it also shows a problem in what was proposed under %79: we don't want to create abstract patterns with more than one level of nesting of patterns. So a pattern like this: '< 6 id10 < 2 j o h > < 5 < 3 n > < 4 s >' which, if it were correct, would be something like this:
'< 6 id10 < 2 j o h > < 5 < 3 > < 7 > < 4 s > > >' has too many levels.

So a new proposal for v 7.7 is to create a notional Old pattern from all the patterns in the alignment that contain DATA_SYMBOLs and have an immediate match with the New pattern. Leave out all ID symbols. In this case, the notional Old pattern would be '< 2 j o h > < 3 n > < 4 s >'.

With an alignment like this:

Learning from alignment ID325:

```
0 m a r y  r u n                                s     0
           | | |                                |
1 < 7 id11 r u n >                              |     1
                                                |
2                                 < 28 16 9 4 id4 s >   2
                                  | |             |
3                  < id29 < 27 > < 28             > > 3
```

Notional Old pattern: r u n < 27 > < 28 s >

the notional Old pattern shows that the ID-symbols around 'r u n' have been omitted. This has a bearing on the possible use of composite patterns in learning.

This alignment also shows that the proposed rule, above, would leave out the reference '< 27 >'. This may be legitimate but we need to bear it in mind.

Here is an alternative rule for forming a notional Old pattern:

"Concatenate all the Old symbols in patterns that are directly matched to the New pattern, excluding those with the type UNIQUE_ID_SYMBOL."

%81 12/1/05

FURTHER NOTES ON SP71, V 7.7

In %80 it is proposed that a notional Old pattern can be made as a concatenation of patterns that are directly matched to New and then learning can proceed from there. The trouble with this idea is that it will lose information about intermediate-level structures and hence undermine what SP71 is all about. With an alignment like this:

```
0              j o h                  n       w a l k s       0
               | | |                  |               |
1       < 2 id3 j o h >                |               |       1
        | |           |                |               |
2       | |           |         < 3 id4 n >             |       2
        | |           |         | |       |            |
3       | |           |         | |       | < 4 id5    s >     3
        | |           |         | |       | | |          |
4       | |           | < 5 id7 < 3       > < 4          > >   4
```

```
        | |           | | |                                 |
5 < id8 < 2           > < 5                                 > > 5
```

we need to preserve the intermediate-level structure '< 5 ... >' (even
though it is probably 'wrong' in this case).

An alternative idea is that, rather than trying to apply learning to the
New pattern and the lowest Old pattern (as originally conceived) or apply
learning to the New pattern and a notional Old pattern constructed from
relatively high level patterns (as in v 7.7), we should apply learning to
the New pattern and ***the most appropriate level*** in a given alignment.

In this example, 'w a l k' would fit between '>' and '<' in the middle of
the pattern id7 in row 4 so the result should be something like
'< 5 id7 < 3 > < 6 > < 4 > >' and '< 6 id9 w a l k >'. Notice that the
newly-created abstract pattern retains the same class number as the one
from which it was derived so it will fit in the same context.

If the alignment had been something like this:

```
0               j o h       x y     z   n               s       0
                | | |                   |               |
1       < 2 id3 j o h >                 |               |       1
        | |           |                 |               |
2       | |           |         < 3 id4 n >             |       2
        | |           |         | |       |             |
3       | |           |         | |       | < 4 id5     s >     3
        | |           |         | |       | | |           |
4       | |           | < 5 id7 < 3       > < 4           > >   4
        | |           | | |                                 |
5 < id8 < 2           > < 5                                 > > 5
```

then the result would have been something like
'< id10 < 2 > < 6 > < 5 > >' and the coded pattern '< 6 id11 x y z >'.

This scheme will be attempted in v 7.8. Here is a possible outline:

1 Translate the algment into al_array[] as before and mark up. Mark
the beginnings and ends of coherent sequences of New-symbol hits.
Then mark each column that is not part of a coherent sequence of New-symbol
hits with proxy_new_int_pos. This means the int_pos of the first or last New
symbol in the coherent sequence of New-symbol hits that the given column
relates to.

2 For the Old pattern in each row other than row 0, look for coherent
sequence of Old-symbol hits and gaps between them.

%82 19/1/05

NOTES ON SP71, V 7.8

Here is an alignment that has still not been fully catered for:

```
0               j o h           n       w a l k s     0
                | | |           |               |
1       < 2 id4 j o h >         |               |     1
        | |           |         |               |
2       | |           | < 3 id5 n >             |     2
        | |           | | |       |             |
3       | |           | | |       | < 4 id3     s >   3
        | |           | | |       | | |           |
4 < id7 < 2           > < 3       > < 4           > > 4
```

At present, the program successfully recognises 'w a l k' and abstracts it
as a separate pattern. It also marks the al_array[] version of the alignment
so that the ID number for that pattern extends from the column to the right
of 'n' to the column to the left of 's'. What is unclear at present is how
to handle this information.

Here are tentative rules:

1 For any given pattern in Old and for each unmatched portion of New, look
to see whether there is any unmatched portion of the Old pattern (CONTENTS
symbols only) that lies 'opposite' the unmatched portion of New. If there
is, abstract that subsequence as a new pattern and create an abstract pattern
and context that accommodates the unmatched portion of New and the unmatched
portion of Old.

2 If there are one or more unmatched portions of the Old pattern that do not
have any corresponding unmatched portion(s) of the New pattern, create an
'abstract' pattern that is modelled on the Old pattern but omits the
unmatched symbols. The relevant Old pattern will always be the last one
added and this will be the lowest Old pattern in the alignment.

3 If there are one or more unmatched portions of the New pattern that do not have any corresponding unmatched portion(s) of the Old pattern, create an abstract pattern that is modelled on the Old pattern but with one or more references added for the patterns derived from the unmatched portions of New. The relevant Old pattern will always be the last one added and this will be the lowest Old pattern in the alignment. The reference to the newly-created subsequence of the New pattern must be inserted *between* references within the Old pattern. If there are no references, the new reference must be inserted between the matched symbols in the Old pattern. For unmatched New subsequences at the beginning or end of the alignment, the new reference must be inserted at the beginning or end of the abstract pattern.

In terms of processing, the abstract pattern can always be modelled on the Old pattern but with a different 'id?' symbol and with all unmatched CONTENTS symbols replaced by an appropriate reference (for 1 and 3), or omitted (for 2). Matched subsequences of the Old pattern can also be abstracted and made into distinct patterns unless they are single references.

%83 24/1/05

FURTHER NOTES ON SP71, V 7.8

The program now learns 'correctly' from this alignment:

```
0              j o h           n       w a l k s     0
               | | |           |               |
1       < 2 id4 j o h >        |               |     1
        | |           |        |               |
2       | |           | < 3 id5 n >             |     2
        | |           | | |       |             |
3       | |           | | |       | < 4 id3     s >   3
        | |           | | |       | | |           |
4 < id7 < 2           > < 3       > < 4           > > 4
```

The results are:

Patterns:

ID52 (< id9 w a l k >)*1

No patterns found for row 1.

No patterns found for row 2.

No patterns found for row 3.

ID54 (< id10 < 2 > < 3 > >)*1

ID56 (< 4 >)*1

Context symbol 5 added to pattern ID54 (< 5 id10 < 2 > < 3 > >)*1

Context symbol 6 added to pattern ID52 (< 6 id9 w a l k >)*1

Single reference pattern ID56 is deleted.

ID58 (< id11 < 5 > < 6 > < 4 > >)*1

These patterns are abstracted as intended. But there is still a problem with the appropriate representation of 'null' elements of an abstract pattern because ID58 suggests that 'w a l k' is always present although it was not when 'id7' was created.

In fact, 'id7' is a generalisation from the two patterns from which it was derived: 'j o h n r u n s' and '< id1 j o h n r u n s >'. This generalisation suggests that the following two patterns are legal: 'j o h n r u n r u n s' and 'j o h n s'.

This over-generalisation can be corrected if, together with the 'general' patterns above, we include patterns that code the specific sequences in New and Old. In this case, the code patterns would include an ID-symbol for the abstract pattern and ID-symbols for each of the three constituents.

As things stand, the ID-symbols for each of the three constituents are sufficient and the abstract pattern adds nothing. This may explain why the program is not giving a high priority to grammars that contain an abstract pattern. We probably need to reduce the bit_cost of the 'id' symbols so that they are just sufficient to differentiate each alternative within a given context rather than being big enough to identify the pattern uniquely, regardless of the context. We probably need also to clean up the grammars at intermediate stages to eliminate ID-symbols that are not needed at all.

With an alignment like this:

```
0      j o h       n w a l k s   0
       | | |       |         |
1 < #1 j o h n r u n         s > 1
```

the program produces these results:

```
CONTENTS symbols of pattern ID68 match those of
pre-existing pattern ID52 (< 8 6 #9 w a l k >)*1

CONTENTS symbols of pattern ID70 match those of
pre-existing pattern ID10 (< 2 #4 j o h >)*1

ID71 (< #15 n r u >)*1

CONTENTS symbols of pattern ID72 match those of
pre-existing pattern ID11 (< 3 #5 n >)*1

CONTENTS symbols of pattern ID74 match those of
pre-existing pattern ID9 (< 9 4 #3 s >)*1

Context symbol 10 added to pattern ID10 (< 10 2 #4 j o h >)*1

Context symbol 11 added to pattern ID71 (< 11 #15 n r u >)*1

Context symbol 12 added to pattern ID11 (< 12 3 #5 n >)*1

Context symbol 13 added to pattern ID9 (< 13 9 4 #3 s >)*1

ID76 (< #16 < 10 > < 11 > < 12 > < 13 > >)*1
```

As before, the abstract pattern adds nothing to the efficiency of encoding because the '#' symbols have been given enough bits to identify them uniquely.

The program has failed to add in a reference to 'w a l k'. This is a programming error that needs correction. When it is corrected, the patterns will generate incorrect forms such as 'j o h n r u n w a l k s'.

Another snag is that the patterns fail to show that some elements are optional. If we are to avoid the problems associated with 'null' patterns, we probably need *two* abstract patterns, something like this:
'< #16 < 10 > < 12 > < 14 > < 13 > >' (where '< 14 >' is a reference to 'w a l k') and '< #17 < 10 > < 11 > < 12 > < 13 > >'.

The trouble with this idea is that it introduces redundancy into the grammar and this grammar could be large. On reflection, it looks as if the use of a 'null' pattern might not be so bad after all. If a null pattern is used, then the procedure that looks for matches between patterns would need to be modified so that it can recognise that one null pattern is a match for another one.

In summary, these seem to be the things that need to be done:

1 Correct the programming error so that there is a reference to 'w a l k' in the abstract pattern that is abstracted from this alignment:

```
0      j o h       n w a l k s   0
       | | |       |         |
1 < #1 j o h n r u n         s > 1
```

This has been done.

2 Introduce null elements wherever an unmatched part pattern is an alternative to nothing. Ensure that any newly-created null element can be recognised as being the same as a previously-created one. This has been done.

3 To ensure lossless compression, create code patterns for each New pattern and include them with the grammars for these patterns. This is something for version 7.9.

4 To ensure that the abstract patterns have a function in compression, only introduce '#' symbols when they are actually needed to differentiate alternatives, re-use the same symbols in different contexts, and give these symbols the minimum number of bits needed to differentiate them from each other. Again, this is something for version 7.9. The potential ambiguity arising from the re-use of the same symbols in different contexts should be reduced by the constraints on the matching of strings containing brackets. If the number of alternatives in a given context is increased, then the number of bits assigned to each '#' symbol would need to be increased.

5 Clean up grammars more often (ie get rid of context symbols that are not used in a particular grammar and, perhaps, rename the symbols from the beginning (mainly for cosmetic purposes). At the end of the processing of any given New pattern,

replace all the Old symbols with the patterns in the best grammar found for the New patterns up to and including the current New pattern. This is something for version 7.9.

%84 27/1/05

FURTHER NOTES ON SP71, V 7.8

On reflection, there is no need to include code patterns for the New patterns in the grammars that SP71 creates. This is because the sizes of the code patterns are accounted for in the size of E for each grammar and this is added to G to obtain an overall score. This has been done.

In Phase 2, the program forms selected alignments like this:

```
SELECTED ALIGNMENT (PATTERN ID1 (ID3), CYCLE 1, PHASE 2)

ID201: NSC = 5255.17, EC = 7.18, CR = 0.00, CD = 5247.99,
Absolute P = 0.00689655172414

0      j o h       n r u n s   0
       | | |       |       |
1 < #1 j o h n r u n       s > 1
```

Since Phase 2 deals only in alignments where all the CONTENTS symbols of Old patterns are fully matched, there seems to be a case for excluding this kind of alignment from the selection. This should increase the chance of including small alignments like the one that encodes the terminal 's'. This will be tried.

%85 28/1/05

FURTHER NOTES ON SP71, V 7.8

With this input:

```
[
    [(j o h n r u n s)
    (j o h n w a l k s)
    (m a r y r u n s)
    (m a r y w a l k s)
    ]
    [
    ]
]
```

the program finds plausible structures for the first two patterns. But with the third pattern, it fails to 'realise' that 'm a r y' can occur in the same contexts as 'j o h n'. So it makes alignments like this:

```
NEW ALIGNMENT (PATTERN ID3 (ID3), CYCLE 2, PHASE 2)

ID310 : ID306 : ID65 : #1026 NSC = 9480.76, EC = 28.72, CR = 0.00, CD = 9452.04,
Absolute P = 0.00

0          m a r y                      r u n           s     0
           | | | |                      | | |           |
1 < 10 #17 m a r y >                    | | |           |     1
                                        | | |           |
2                                < 8 #14 r u n >         |     2
                                 | |          |         |
3                                | |          | < 9 4 #3 s >   3
                                 | |          | | |       |
4                    < #15 < 7 > < 8          > < 9       > > 4
```

In phase 1, the program should be able to learn the correct categorization of 'm a r y' from an alignment like this:

```
SELECTED ALIGNMENT (PATTERN ID3, CYCLE 2, PHASE 1)

ID176: NSC = 4813.06, EC = 24.72, CR = 0.01, CD = 4788.35,
Absolute P = 3.61949493907e-008

0 m a r y            r u n           s     0
                     | | |           |
1            < 8 #14 r u n >         |     1
             | |          |         |
2            | |          | < 9 4 #3 s >   2
             | |          | | |       |
3 < #15 < 7 > < 8          > < 9       > > 3
```

This needs looking at.

%86

NOTES ON SP71, V 7.9

Continuing previous discussion, the reason that 'm a r y' is not recognised
as belonging in context class '7' is because alignment ID176 is not used for
learning (the program is already designed to assign 'm a r y' to '7' with an
alignment like this). However, if the number of alignments used for learning
is increased, the program does not produce an alignment like ID176 - so the
program still fails to recognise the category of 'm a r y'.

The reason that the program does not form alignments like ID176 is because there
are many more patterns in Old and the program fails to select the '< 9 4 #3 s >'
alignment. This means that it does not form any combinations in cycle 1 with
'm a r y r u n s' as the New pattern and it cannot form an alignment like ID176.
By increasing the number of alignments that are selected, this may cure this
problem. This is indeed the case but then the resulting alignment is not selected
for learning because its score is too low. This may be cured by increasing the
number of alignments used for learning. We do eventually get these results:

Learning from alignment ID497:

```
0 m a r y            r u n                s     0
                     | | |                |
1             < 8 #14 r u n >             |     1
              | |          |               |
2             | |          | < 21 14 9 4 #3 s >   2
              | |          | |       |       |
3 < #15 < 7 > < 8          > <       9       > > 3
```

Patterns:

CONTENTS symbols of pattern ID636 match those of
pre-existing pattern ID537 (< 3 22 30 29 23 #29 m a r y >)*1

Patterns for alignment ID497, row 1:

No patterns found for row 1.

Patterns for alignment ID497, row 2:

No patterns found for row 2.

Patterns for alignment ID497, row 3:

ID637 (< 7 >)*1

ID638 (< #51 < 8 > < 9 > >)*1

Context symbol 7 added to pattern ID537 (< 7 3 22 30 29 23 #29 m a r y >)*1

Single reference pattern ID637 is deleted.

Context symbol 57 added to pattern ID638 (< 57 #51 < 8 > < 9 > >)*1

ID640 (< #52 < 7 > < 57 > >)*1

UNIQUE_ID_SYMBOLS V DISCRIMINATION SYMBOLS

The idea of a 'discrimination symbol' that discriminates amongst alternatives in a
given context makes sense if any given pattern occurs in only one context. But many
patterns can occur in two or more different contexts. In that case it makes more
sense to assign each pattern a unique_id_symbol, as at present and vary the weight
attached to any given symbol, depending on how often the pattern that it identifies
is used. So SP71, v 7.10, will retain unique_id_symbols.

%87 1/2/05

NOTES ON SP71, V 7.10

This version re-introduces cleaning up of grammars (removing unnecessary code symbols)
and tidying up of grammars (renumbering code symbols to look tidy). This is done at
the end of the processing each New pattern and the Old patterns are replaced by
the patterns in the best 2 grammars.

Now, the process looks promising with the first two New patterns but after the
third New pattern, the Old patterns are:

ID1065 (< 4 #9 w a l k >)*1

ID1066 (< 1 #10 j o h n >)*2

ID1067 (< 4 #11 r u n >)*2

```
ID1068 (< 5 #12 s >)*3

ID1069 (< 2 #30 m a r y >)*1

ID1070 (< 3 #40 < 4 > < 5 > >)*3

ID1075 (< #13 < 1 > < 4 > < 5 > >)*2

This is not too bad but ID1070 is not needed at all and, as before, the program
fails to recognise that 'j o h n' and 'm a r y' are contextually equivalent.

This seems to have been due to a programming error and is now fixed:

Grammar GR131, derived from grammar GR70
and alignment ID744 (< #42 < 4 4 23 #27 m a r y > < 5 #11 r u n > < 6 43 6 34 6 #12 s > >)*1

G = 31687.36, E = 175.66, score = 31863.01.

Grammar patterns:

ID321 (< 4 #10 j o h n >)*2

ID322 (< 5 #11 r u n >)*2

ID323 (< 6 43 6 34 6 #12 s >)*3

ID403 (< #42 < 4 > < 5 > < 6 > >)*3

ID320 (< 5 #9 w a l k >)*1

ID355 (< 4 4 23 #27 m a r y >)*1

But there is at least one other error as shown by two instances of '4' in ID355.
This is simply because context symbols are added without any check on whether
they might be there already. This error is now corrected:

Grammar GR132, derived from grammar GR77
and alignment ID650 (< #42 < 4 22 #27 m a r y > < 5 #11 r u n > < 42 33 6 #12 s > >)*1

G = 31513.75, E = 144.58, score = 31658.32.

Grammar patterns:

ID284 (< 5 #9 w a l k >)*1

ID285 (< 4 #10 j o h n >)*2

ID286 (< 5 #11 r u n >)*2

ID287 (< 42 33 6 #12 s >)*3

ID319 (< 4 22 #27 m a r y >)*1

ID367 (< #42 < 4 > < 5 > < 6 > >)*3

%88 2/2/05

FURTHER NOTES ON SP71, V 7.10

With some further tidying up of the program, this input file:

[
    [(j o h n r u n s)
    (j o h n w a l k s)
    (m a r y r u n s)
    (m a r y w a l k s)
    ]
    [
    ]
]

Now gives this final result:

OLD PATTERNS DERIVED FROM THE BEST 2 GRAMMARS AT THE END OF PATTERN ID4, PHASE 2:

ID1411 (< 2 #2 w a l k >)*2

ID1412 (< 1 #1 j o h n >)*2

ID1413 (< 2 #3 r u n >)*2

ID1414 (< 3 #6 s >)*4
```

ID1415 (< 1 #4 m a r y >)*2

ID1416 (< #5 < 1 > < 2 > < 3 > >)*4

This is derived from the best two grammar which, before cleaning and tidying are:

GRAMMAR GR346

Grammar GR346, derived from grammar GR246
and alignment ID1310 (< #6 < 1 #4 m a r y > < 2 #2 w a l k > < 48 3 #5 s > >)*1

G = 29137.28, E = 148.34, score = 29285.62.

Grammar patterns:

ID878 (< 2 #2 w a l k >)*2

ID879 (< 1 #1 j o h n >)*2

ID880 (< 2 #3 r u n >)*2

ID881 (< 48 3 #5 s >)*4

ID882 (< 1 #4 m a r y >)*2

ID911 (< #48 < 1 > < 2 > < 3 > >)*4

GRAMMAR GR347

Grammar GR347, derived from grammar GR247
and alignment ID1310 (< #6 < 1 #4 m a r y > < 2 #2 w a l k > < 48 3 #5 s > >)*1

G = 29137.28, E = 148.34, score = 29285.62.

Grammar patterns:

ID878 (< 2 #2 w a l k >)*2

ID879 (< 1 #1 j o h n >)*2

ID880 (< 2 #3 r u n >)*2

ID881 (< 48 3 #5 s >)*4

ID882 (< 1 #4 m a r y >)*2

ID883 (< #6 < 1 > < 2 > < 3 > >)*4

The remaining grammars are:

GRAMMAR GR347

Grammar GR347, derived from grammar GR247
and alignment ID1310 (< #6 < 1 #4 m a r y > < 2 #2 w a l k > < 48 3 #5 s > >)*1

G = 29137.28, E = 148.34, score = 29285.62.

Grammar patterns:

ID878 (< 2 #2 w a l k >)*2

ID879 (< 1 #1 j o h n >)*2

ID880 (< 2 #3 r u n >)*2

ID881 (< 48 3 #5 s >)*4

ID882 (< 1 #4 m a r y >)*2

ID883 (< #6 < 1 > < 2 > < 3 > >)*4

GRAMMAR GR368

Grammar GR368, derived from grammar GR268
and alignment ID1325 (< 1 #4 m a r y > < 43 #50 < 2 #2 w a l k > < 48 3 #5 s > >)*1

G = 29133.57, E = 191.18, score = 29324.75.

Grammar patterns:

ID878 (< 2 #2 w a l k >)*2

ID879 (< 1 #1 j o h n >)*2

ID880 (< 2 #3 r u n >)*2

ID881 (< 48 3 #5 s >)*4

ID882 (< 1 #4 m a r y >)*2

ID918 (< 43 #50 < 2 > < 3 > >)*4

GRAMMAR GR349

Grammar GR349, derived from grammar GR250
and alignment ID1310 (< #6 < 1 #4 m a r y > < 2 #2 w a l k > < 48 3 #5 s > >)*1

G = 29173.12, E = 159.05, score = 29332.17.

Grammar patterns:

ID878 (< 2 #2 w a l k >)*2

ID879 (< 1 #1 j o h n >)*2

ID880 (< 2 #3 r u n >)*2

ID881 (< 48 3 #5 s >)*4

ID882 (< 1 #4 m a r y >)*2

ID911 (< #48 < 1 > < 2 > < 3 > >)*4

ID918 (< 43 #50 < 2 > < 3 > >)*4

GRAMMAR GR350

Grammar GR350, derived from grammar GR251
and alignment ID1310 (< #6 < 1 #4 m a r y > < 2 #2 w a l k > < 48 3 #5 s > >)*1

G = 29173.12, E = 159.05, score = 29332.17.

Grammar patterns:

ID878 (< 2 #2 w a l k >)*2

ID879 (< 1 #1 j o h n >)*2

ID880 (< 2 #3 r u n >)*2

ID881 (< 48 3 #5 s >)*4

ID882 (< 1 #4 m a r y >)*2

ID883 (< #6 < 1 > < 2 > < 3 > >)*4

ID918 (< 43 #50 < 2 > < 3 > >)*4

GRAMMAR GR361

Grammar GR361, derived from grammar GR259
and alignment ID1389 (< #51 < 1 #4 m a r y > < 43 #50 < 2 #2 w a l k > < 48 3 #5 s > > >)*1

G = 29164.47, E = 176.40, score = 29340.88.

Grammar patterns:

ID878 (< 2 #2 w a l k >)*2

ID879 (< 1 #1 j o h n >)*2

ID880 (< 2 #3 r u n >)*2

ID881 (< 48 3 #5 s >)*4

ID882 (< 1 #4 m a r y >)*2

ID918 (< 43 #50 < 2 > < 3 > >)*4

ID919 (< #51 < 1 > < 43 > >)*4

GRAMMAR GR353

Grammar GR353, derived from grammar GR256
and alignment ID1310 (< #6 < 1 #4 m a r y > < 2 #2 w a l k > < 48 3 #5 s > >)*1

G = 29204.03, E = 155.35, score = 29359.38.

Grammar patterns:

ID878 (< 2 #2 w a l k >)*2

ID879 (< 1 #1 j o h n >)*2

ID880 (< 2 #3 r u n >)*2

ID881 (< 48 3 #5 s >)*4

ID882 (< 1 #4 m a r y >)*2

ID911 (< #48 < 1 > < 2 > < 3 > >)*4

ID918 (< 43 #50 < 2 > < 3 > >)*4

ID919 (< #51 < 1 > < 43 > >)*4

GRAMMAR GR354

Grammar GR354, derived from grammar GR257
and alignment ID1310 (< #6 < 1 #4 m a r y > < 2 #2 w a l k > < 48 3 #5 s > >)*1

G = 29204.03, E = 155.35, score = 29359.38.

Grammar patterns:

ID878 (< 2 #2 w a l k >)*2

ID879 (< 1 #1 j o h n >)*2

ID880 (< 2 #3 r u n >)*2

ID881 (< 48 3 #5 s >)*4

ID882 (< 1 #4 m a r y >)*2

ID883 (< #6 < 1 > < 2 > < 3 > >)*4

ID918 (< 43 #50 < 2 > < 3 > >)*4

ID919 (< #51 < 1 > < 43 > >)*4

GRAMMAR GR372

Grammar GR372, derived from grammar GR278
and alignment ID1325 (< 1 #4 m a r y > < 43 #50 < 2 #2 w a l k > < 48 3 #5 s > >)*1

G = 35535.23, E = 208.31, score = 35743.54.

Grammar patterns:

ID878 (< 2 #2 w a l k >)*2

ID879 (< 1 #1 j o h n >)*2

ID880 (< 2 #3 r u n >)*2

ID881 (< 48 3 #5 s >)*4

ID882 (< 1 #4 m a r y >)*2

ID918 (< 43 #50 < 2 > < 3 > >)*4

ID937 (< 52 #61 r y >)*2

ID951 (< 57 #69 m a >)*2

GRAMMAR GR355

Grammar GR355, derived from grammar GR276
and alignment ID1310 (< #6 < 1 #4 m a r y > < 2 #2 w a l k > < 48 3 #5 s > >)*1

G = 35574.78, E = 176.17, score = 35750.96.

Grammar patterns:

ID878 (< 2 #2 w a l k >)*2

ID879 (< 1 #1 j o h n >)*2

```
ID880 (< 2 #3 r u n >)*2

ID881 (< 48 3 #5 s >)*4

ID882 (< 1 #4 m a r y >)*2

ID911 (< #48 < 1 > < 2 > < 3 > >)*4

ID918 (< 43 #50 < 2 > < 3 > >)*4

ID937 (< 52 #61 r y >)*2

ID951 (< 57 #69 m a >)*2

It is interesting that grammars GR361, GR353 and GR354 have an hierarchical
structure, even though it is spurious for this input.

The 'cost factor' of 300 seems to tilt the grammars heavily towards those that
minimise the number DATA_SYMBOLs. There is no sign of grammars with 'r u n s'
and 'w a l k s'. With a cost factor of 10, one of the grammars contains
'w a l k s'. With a cost factor of 2, the best two grammars are:

GRAMMAR GR178

Grammar GR178, derived from grammar GR138
and alignment ID1088 (< #55 m a r y w a l k s >)*1

G = 432.87, E = 36.41, score = 469.29.

Grammar patterns:

ID677 (< #1 j o h n r u n s >)*1

ID678 (< #7 j o h n w a l k s >)*1

ID679 (< #2 m a r y r u n s >)*1

ID685 (< #55 m a r y w a l k s >)*1

GRAMMAR GR211

Grammar GR211, derived from grammar GR159
and alignment ID1143 (< #6 < 49 47 1 #3 m a r y > < 2 48 #56 w a l k s > >)*1

G = 328.93, E = 160.76, score = 489.69.

Grammar patterns:

ID681 (< 49 47 1 #3 m a r y >)*2

ID682 (< 73 47 1 #4 j o h n >)*2

ID683 (< 2 #5 r u n s >)*2

ID700 (< 2 48 #56 w a l k s >)*2

ID727 (< #69 < 1 > < 2 > >)*4

With a cost factor of 3, the best 5 grammars are:

GRAMMAR GR209

Grammar GR209, derived from grammar GR159
and alignment ID1143 (< #6 < 49 47 1 #3 m a r y > < 2 48 #56 w a l k s > >)*1

G = 423.18, E = 160.76, score = 583.94.

Grammar patterns:

ID681 (< 49 47 1 #3 m a r y >)*2

ID682 (< 73 47 1 #4 j o h n >)*2

ID683 (< 2 #5 r u n s >)*2

ID700 (< 2 48 #56 w a l k s >)*2

ID727 (< #69 < 1 > < 2 > >)*4

GRAMMAR GR210

Grammar GR210, derived from grammar GR160
and alignment ID1143 (< #6 < 49 47 1 #3 m a r y > < 2 48 #56 w a l k s > >)*1
```

G = 423.18, E = 160.76, score = 583.94.

Grammar patterns:

ID681 (< 49 47 1 #3 m a r y >)*2

ID682 (< 73 47 1 #4 j o h n >)*2

ID683 (< 2 #5 r u n s >)*2

ID684 (< #6 < 1 > < 2 > >)*4

ID700 (< 2 48 #56 w a l k s >)*2

GRAMMAR GR195

Grammar GR195, derived from grammar GR163
and alignment ID1150 (< #57 < 49 47 1 #3 m a r y > < 2 48 #56 w a l k s > >)*1

G = 459.49, E = 157.59, score = 617.08.

Grammar patterns:

ID681 (< 49 47 1 #3 m a r y >)*2

ID682 (< 73 47 1 #4 j o h n >)*2

ID683 (< 2 #5 r u n s >)*2

ID700 (< 2 48 #56 w a l k s >)*2

ID702 (< #57 < 47 > < 48 > >)*2

ID727 (< #69 < 1 > < 2 > >)*4

GRAMMAR GR196

Grammar GR196, derived from grammar GR164
and alignment ID1150 (< #57 < 49 47 1 #3 m a r y > < 2 48 #56 w a l k s > >)*1

G = 459.49, E = 157.59, score = 617.08.

Grammar patterns:

ID681 (< 49 47 1 #3 m a r y >)*2

ID682 (< 73 47 1 #4 j o h n >)*2

ID683 (< 2 #5 r u n s >)*2

ID684 (< #6 < 1 > < 2 > >)*4

ID700 (< 2 48 #56 w a l k s >)*2

ID702 (< #57 < 47 > < 48 > >)*2

GRAMMAR GR178

Grammar GR178, derived from grammar GR138
and alignment ID1088 (< #55 m a r y w a l k s >)*1

G = 621.38, E = 36.41, score = 657.79.

Grammar patterns:

ID677 (< #1 j o h n r u n s >)*1

ID678 (< #7 j o h n w a l k s >)*1

ID679 (< #2 m a r y r u n s >)*1

ID685 (< #55 m a r y w a l k s >)*1

Here, the grammar which repeats the original sentences comes well down the
list instead of at the top of the list.

A little experimentation shows that a cost factor of 10 is, with this input,
at or near the 'tipping point' between grammars in which 'r u n', 'w a l k'
and 's' predominate and those where 'r u n s' and 'w a l k s' are dominant.
Likewise, a cost factor of 2 is at or near the tipping point between grammars
in which the original four sentences are the best grammar and those where
the inclusion of 'w a l k s' and 'r u n s' is best.

%89 2/2/05

FURTHER NOTES ON SP71, V 7.10

With this input:

```
[
    [
        (w e r u n)
        (w e w a l k)
        (t h e y r u n)
        (t h e y w a l k)
        (w e r u n f a s t)
        (w e r u n s l o w l y)
        (w e w a l k f a s t)
        (w e w a l k s l o w l y)
        (t h e y r u n f a s t)
        (t h e y r u n s l o w l y)
        (t h e y w a l k f a s t)
        (t h e y w a l k s l o w l y)
    ]
    [
    ]
]
```

the best grammar produced is:

GRAMMAR GR2174

Grammar GR2174, derived from grammar GR2054
and alignment
ID7695 (< #2 < 1 #1 t h e y > < 2 #4 w a l k > < 136 3 #7 s l o w l y > >)*1

G = 1521.47, E = 364.02, score = 1885.49.

Grammar patterns:

ID6289 (< 2 #4 w a l k >)*6

ID6290 (< 1 #6 w e >)*6

ID6291 (< 2 #8 r u n >)*6

ID6292 (< 1 #1 t h e y >)*6

ID6293 (< 3 #5 f a s t >)*4

ID6294 (< 136 3 #7 s l o w l y >)*4

ID6377 (< #193 < 1 > < 2 > >)*6

ID6385 (< #194 < 1 > < 2 > < 3 > >)*8

and Old patterns derived from the best 2 grammars after cleaning and tidying:

OLD PATTERNS DERIVED FROM THE BEST 2 GRAMMARS AT THE END OF PATTERN ID12, PHASE 2:

ID7792 (< 2 #4 w a l k >)*6

ID7793 (< 1 #6 w e >)*6

ID7794 (< 2 #8 r u n >)*6

ID7795 (< 1 #1 t h e y >)*6

ID7796 (< 3 #5 f a s t >)*4

ID7797 (< 3 #7 s l o w l y >)*4

ID7798 (< #2 < 1 > < 2 > >)*6

ID7799 (< #3 < 1 > < 2 > < 3 > >)*8

This is close to the 'right' answer except that the program has failed to detect
that ID7798 is embedded in ID7799. And all the frequency values look right
except for the frequency of ID7798, which should be 4.

%90 3/2/05

FURTHER NOTES ON SP71, V 7.10

At least part of the reason that the program does not find intermediate

structure in the example from %89 can be seen from the way it learns from
a parsing produced for the fifth New pattern, the first of the New
patterns that has three words instead of two:

Learning from alignment ID442:

```
0            w e         r u n   f a s t 0
             | |         | | |
1      < 1 #4 w e >      | | |          1
       | |       |       | | |
2      | |       | < 2 #5 r u n >       2
       | |       | | |         |
3 < #2 < 1       > < 2         > >      3
```

Patterns:

ID447 (< #34 f a s t >)*1

No patterns found for row 1.

No patterns found for row 2.

No patterns found for row 3.

At present, the program sees that all three Old patterns are fully matched
and fails to realise that ID447 has not yet been fitted into an abstract
pattern. A new version of SP71 (v 7.11) will be started to explore
how this can be solved.

%91 6/2/05

NOTES ON SP71, V 7.11

This version attempts to take account of unmatched portions of New that are
outside the main alignment by extending the existing learning mechanisms. This
dose not result in the anticipated hierarchical structure and has been fiddly
and complicated to program.

A neater and simpler idea may be to return to the learning processes in v 7.10
and then take account of unmatched portions of New that are outside the main
alignment (or alignments) at the stage when the program forms combinations of
alignments. Here is an outline of what may work:

1 Modify the parsing process so that one alignment can only proceed from one
cycle to the next if, in addition to its having all its CONTENTS symbols matched,
all the New symbols within the scope of the alignment form a coherent hit
sequence with no gaps between the New hit symbols. With this rule in place,
one can be confident that learning that involves patterns *within* the scope
of an alignment will only apply to the last-added Old pattern in an alignment,
which is always the bottom pattern. There is no need to test whether higher-level
Old patterns might be involved in learning.

2 With regard to unmatched New symbols that are *outside* the scope of any
alignment, the program forms these into new encoded patterns and then creates
an abstract pattern to tie together the newly-created patterns and references
to the lowest-level patterns of the one or more alignments that have been formed.
This technique can be applied to composite alignments as well as single alignments.

These things will be done in v 7.12.

%92 8/2/05

NOTES ON SP71, V 7.12

This has now been augmented as described in %92. But the program is still failing to
find hierarchical structure in grammars although it does form hierarchical
structures during learning. Here are some examples with this input:

```
[
    [
        (w e r u n)
        (w e w a l k)
        (t h e y r u n)
        (t h e y w a l k)
        (w e r u n f a s t)
    ]
    [
    ]
]
```

1 Learning from alignment ID454:

```
0            w e           r u n   f a s t 0
```

```
             | |           | | |
1      < 1 #4 w e >         | | |          1
       | |       |         | | |
2      | |       | < 2 3 #5 r u n >        2
       | |       | |   |         |
3 < #6 < 1       > <   3         > >      3
```

Part patterns from New:

ID462 (< #34 f a s t >)*1

Context symbol 24 added to pattern ID439 (< 24 #6 < 1 > < 3 > >)*4

Context symbol 25 added to pattern ID462 (< 25 #34 f a s t >)*1

ID463 (< #35 < 24 > < 25 > >)*1

This example shows how the system can learn hieararchical structures. But for some reason that is not yet entirely clear, it does not recognise these hierarchical structures in the grammars that it forms.

2 The aggregation method also gives results like this:

Learning from alignment ID114:

```
0 t h e y r u n   0
          | | |
1 < 2 #4  r u n > 1
```

Part patterns from New:

ID118 (< #15 t h e y >)*1

Context symbol 9 added to pattern ID118 (< 9 #15 t h e y >)*1

Context symbol 10 added to pattern ID105 (< 10 2 #4 r u n >)*1

ID119 (< #16 < 9 > < 10 > >)*1

3 And the method for processing an Old pattern gives results like this:

Learning from alignment ID116:

```
0 t h e y           r u n     0
                    | | |
1           < 2 #4 r u n >    1
            | |          |
2 < #5 < 1 > < 2          > > 2
```

Part patterns from New:

CONTENTS symbols of pattern ID120 match those of
pre-existing pattern ID118 (< 9 #15 t h e y >)*1

Part patterns for Old pattern in row 2 (alignment ID116):

ID121 (< 1 >)*1

ID122 (< 2 >)*1

Context symbol 1 added to pattern ID118 (< 1 9 #15 t h e y >)*1

Single reference pattern ID121 is deleted.

Single reference pattern ID122 is deleted.

CONTENTS symbols of pattern ID124 match those of
pre-existing pattern ID106 (< #5 < 1 > < 2 > >)*2

It would be useful if the abstract pattern created with ID114 was the same as as the abstract pattern created with ID116.

4 Again, the aggregation method gives different results in two cases where it should give the same results:

Learning from alignment ID224:

```
0 t h e y w a l k   0
          | | | |
1 < 3 #3  w a l k > 1
```

Part patterns from New:

```
CONTENTS symbols of pattern ID236 match those of
pre-existing pattern ID216 (< 1 4 #1 t h e y >)*1

Start of aggregate_part_patterns_and_alignments().

Context symbol 14 added to pattern ID216 (< 14 1 4 #1 t h e y >)*1

Context symbol 15 added to pattern ID212 (< 15 3 #3 w a l k >)*1

ID237 (< #23 < 14 > < 15 > >)*1

End of aggregate_part_patterns_and_alignments().

and

Learning from alignment ID225:

0          t h e y w a l k 0
           | | | |
1 < 1 4 #1 t h e y >       1

Part patterns from New:

CONTENTS symbols of pattern ID244 match those of
pre-existing pattern ID212 (< 15 3 #3 w a l k >)*1

Start of aggregate_part_patterns_and_alignments().

Context symbol 16 added to pattern ID216 (< 16 14 1 4 #1 t h e y >)*1

Context symbol 17 added to pattern ID212 (< 17 15 3 #3 w a l k >)*1

ID245 (< #24 < 16 > < 17 > >)*1

End of aggregate_part_patterns_and_alignments().

5 This example is problematic:

Learning from alignment ID455:

0        w e                      r u n   f a s t 0
         | |                      | | |
1 < 1 #4 w e >                    | | |          1
                                  | | |
2                        < 2 3 #5 r u n >         2
                         | |           |
3              < #2 < 4 > < 2           > >       3

Part patterns from New:

CONTENTS symbols of pattern ID465 match those of
pre-existing pattern ID462 (< 25 #34 f a s t >)*1

Start of process_old_pattern().

Part patterns for Old pattern in row 3 (alignment ID455):

ID466 (< 4 >)*1

ID467 (< 2 >)*1

Single reference pattern ID466 is deleted.

Single reference pattern ID467 is deleted.

Context symbol 26 added to pattern ID462 (< 26 25 #34 f a s t >)*1

Context symbol 26 added to pattern ID470 (< 26 #36 >)*1

ID469 (< #37 < 4 > < 2 > < 26 > >)*1

End of process_old_pattern().

Questions and issues:

* The initial 'w e', with '< 1 #4 w e >' does not enter into learning at all but
the trailing 'f a s t' does enter into the learning process. There may be a case
for a rule that, in process_old_pattern(), the only New part patterns that are
involved are those that fall within the scope of the Old pattern (meaning that
leading and trailing unmatched portions of New would be excluded). 'Within the
scope of the Old pattern' means that there must be a matched New symbol to the
left and right of the given New part pattern and these matched New symbols must
fall within the scope of the Old pattern (but see 7 below).
```

* aggregate_part_patterns_and_alignments() deals with all New part patterns that
are outside the scope of any Old pattern which seems to confirm the rule just
described.

process_old_pattern() will be changed in accordance with this rule.

6 Question: why is the program not recognising hierarchical structure in the
grammars? The grammars are derived from FULL_A alignments so we need to look
at those. For pattern ID5, the best alignments are:

ID531: (< #2 < 32 4 27 1 #4 w e > < 28 2 3 #5 r u n > >)*1

ID534: (< 24 #6 < 32 4 27 1 #4 w e > < 28 2 3 #5 r u n > >)*1

ID594: (< #2 < 32 4 27 1 #4 w e > < 2 3 #3 w a l k > >)*1

ID596: (< 24 #6 < 32 4 27 1 #4 w e > < 2 3 #3 w a l k > >)*1

ID666: (< #2 < 1 4 #1 t h e y > < 28 2 3 #5 r u n > >)*1

ID667: (< 24 #6 < 1 4 #1 t h e y > < 28 2 3 #5 r u n > >)*1

ID738: (< #2 < 1 4 #1 t h e y > < 2 3 #3 w a l k > >)*1

ID739: (< 24 #6 < 1 4 #1 t h e y > < 2 3 #3 w a l k > >)*1

ID811: (< #39 < 32 4 27 1 #4 w e > < 28 2 3 #5 r u n > < 31 30 29 26 25 #34 f a s
t > >)*1

ID813: (< #37 < 32 4 27 1 #4 w e > < 28 2 3 #5 r u n > < 31 30 29 26 25 #34 f a s
t > >)*1

ID816: (< #40 < 32 4 27 1 #4 w e > < 28 2 3 #5 r u n > < 31 30 29 26 25 #34 f a s
t > >)*1

ID819: (< #38 < 32 4 27 1 #4 w e > < 28 2 3 #5 r u n > < 31 30 29 26 25 #34 f a s
t > >)*1

The hierarchical parsing is not present in the set of full alignments from which the
grammars are derived in spite of the fact that the necessary patterns are present
in Old when parsing for Phase 2 is done:

ID436: (< 2 3 #3 w a l k >)*2

ID437: (< 32 4 27 1 #4 w e >)*2

ID438: (< 28 2 3 #5 r u n >)*2

ID439: (< 24 #6 < 1 > < 3 > >)*4

ID440: (< 1 4 #1 t h e y >)*2

ID462: (< 31 30 29 26 25 #34 f a s t >)*1

ID463: (< #35 < 24 > < 25 > >)*1

Since the relevant parsings are not present, it is not surprising that the hierarchical
structure does not appear in the grammars.

The reason the hierarchical parsing does not appear is probably because the presence
in the Old patterns of patterns like:

ID469: (< #37 < 4 > < 2 > < 26 > >)*1

ID473: (< #38 < 27 > < 28 > < 29 > >)*1

ID479: (< #39 < 4 > < 2 > < 30 > >)*1

ID485: (< #40 < 1 > < 3 > < 31 > >)*1

means that there are too many opportunities to form the non-hierarchical parsings.
If more alignments were selected on each cycle, the hierarchical parsing should
come in.

Looking more closely, we find that the program has recognised pattern
ID439: (< 24 #6 < 1 > < 3 > >)*4 in the three-word sentence but the
resulting alignment:

```
0                       w e             r u n                       f a s 0
                        | |             | | |                       | | |
1         < 32 4 27 1 #4 w e >          | | |                       | | | 1
          |         |       |           | | |                       | | |
```

```
2         |         |        | < 28 2 3 #5 r u n >                    | | | 2
          |         |        | |      |          |                    | | |
3         |         |        | |      |          | < 31 30 29 26 25 #34 f a s 3
          |         |        | |      |          |
4 < 24 #6 <         1        > <      3          >                          4


0 t     0
  |
1 |     1
  |
2 |     2
  |
3 t >   3

4     > 4
```

is recognised as illegal and is deleted - so it cannot go on into cycle 3 and form part of an hierarchical parsing. This is the key reason why the program is not forming hierarchical parsings and not recognising hierarchical structures in its grammars.

If the last right bracket in row 4 were to be positioned immediately to the right of '< 28 2 3 #5 r u n >' in row 2, then the alignment would not be rejected as illegal and it could proceed to cycle 3. We need to look at why the last bracket is wrongly positioned and correct this.

When this fault is corrected, we get an hierarchical parsing like this:

NEW ALIGNMENT (PATTERN ID5 (ID5), CYCLE 3, PHASE 2)

ID914 : ID840 : ID468 : #2546 NSC = 5209.04, EC = 67.47, CR = 77.20, CD = 5141.57, Absolute P = 0.00

```
0                             w e              r u n                    0
                              | |              | | |
1               < 32 4 27 1 #4 w e >           | | |                    1
                |         |        |           | | |
2               |         |        | < 28 2 3 #5 r u n >                 2
                |         |        | |      |          |
3               |         |        | |      |          |   < 31 30 29 26 25 3
                |         |        | |      |          |   |             |
4       < 24 #6 <         1        > <      3          > > |             |  4
        | |                                                | |           |
5 < #35 < 24                                               > <           25 5


0     f a s t     0
      | | | |
1     | | | |     1
      | | | |
2     | | | |     2
      | | | |
3 #34 f a s t >   3
            |
4           |   4
            |
5           > > 5
```

and this alignment appears in the cumulative list of full alignments:

ID914: (< #35 < 24 #6 < 32 4 27 1 #4 w e > < 28 2 3 #5 r u n > > < 31 30 29 26 25 #34 f a s t > >)*1

But none of the grammars are recognising this hierarchical structure.

7 On reflection, the proposed restriction that process_old_pattern() should only consider part New patterns that have hit New symbols on either side within the scope of the Old pattern (6 above) would prevent the method dealing sensibly with alignments like this:

Learning from alignment ID11:

```
0      w e w a l k 0
       | |
1 < #1 w e r u n > 1
```

Part patterns from New:

ID16 (< #3 w a l k >)*1

Start of process_old_pattern().

```
Part patterns for Old pattern in row 1 (alignment ID11):

ID18 (< #4 w e >)*1

ID19 (< #5 r u n >)*1

Context symbol 1 added to pattern ID18 (< 1 #4 w e >)*1

Context symbol 2 added to pattern ID19 (< 2 #5 r u n >)*1

Context symbol 2 added to pattern ID16 (< 2 #3 w a l k >)*1

ID20 (< #6 < 1 > < 2 > >)*1

or this:

Learning from alignment ID116:

0 t h e y          r u n     0
                   | | |
1           < 2 #4 r u n >   1
            | |          |
2 < #5 < 1 > < 2         > > 2

Part patterns from New:

CONTENTS symbols of pattern ID120 match those of
pre-existing pattern ID118 (< 9 #15 t h e y >)*1

Start of process_old_pattern().

Part patterns for Old pattern in row 2 (alignment ID116):

ID121 (< 1 >)*1

ID122 (< 2 >)*1

Context symbol 1 added to pattern ID118 (< 1 9 #15 t h e y >)*1

Single reference pattern ID121 is deleted.

Single reference pattern ID122 is deleted.

CONTENTS symbols of pattern ID124 match those of
pre-existing pattern ID106 (< #5 < 1 > < 2 > >)*2

End of process_old_pattern().

There may still be a case for excluding part New patterns at the start or end
of the alignment that do not fall opposite a part Old pattern.

%93 9/2/05

FURTHER NOTES ON SP71, V 7.12

Finally, the program has formed a plausible grammar with hierarchical structure!!!
This is after process_old_pattern() was 'corrected' so that part New patterns which
are at the beginning or end and are not opposite a part Old pattern are considered
to be outside the scope of the alignment. This reduces the number of different
non-hierarchical abstract patterns and gives the hierarchical parsings to come
into view.

Here is the hierarchical grammar:

GRAMMAR GR196

Grammar GR196, derived from grammar GR121
and alignment ID807 (< #34 < 22 #6 < 27 4 24 1 #4 w e > < 25 2 3 #5 r u n > > < 26 23 #33 f a s t > >)*1

G = 10197.10, E = 274.21, score = 10471.31.

Grammar patterns:

ID451 (< 2 3 #3 w a l k >)*2

ID452 (< 27 4 24 1 #4 w e >)*3

ID453 (< 25 2 3 #5 r u n >)*3

ID454 (< 22 #6 < 1 > < 3 > >)*5

ID455 (< 1 4 #1 t h e y >)*2
```

ID477 (< 26 23 #33 f a s t >)*1

ID478 (< #34 < 22 > < 23 > >)*1

It is not given the lowest score for reasons that need looking at. The lowest score goes to:

GRAMMAR GR186

Grammar GR186, derived from grammar GR114
and alignment ID761 (< #2 < 27 4 24 1 #4 w e > < 25 2 3 #5 r u n > > < 26 23 #33 f a s t >)*1

G = 10154.65, E = 247.56, score = 10402.20.

Grammar patterns:

ID451 (< 2 3 #3 w a l k >)*2

ID452 (< 27 4 24 1 #4 w e >)*3

ID453 (< 25 2 3 #5 r u n >)*3

ID455 (< 1 4 #1 t h e y >)*2

ID461 (< #2 < 4 > < 2 > >)*5

ID477 (< 26 23 #33 f a s t >)*1

This balance may change when more three-word sentences are processed and the coding
advantage of the hierarchical grammar should show more clearly.

%94 10/2/05

FURTHER NOTES ON SP71, V 7.12

Possible reasons why the hierarchical grammar is not best in the examples in %93:

1 The best grammar has these scores: G = 10154.65, E = 247.56, score = 10402.20,
while the hierarchical grammar has these scores: G = 10197.10, E = 274.21, score = 10471.31.
In the hierarchical grammar, G is bigger, most likely because it has one extra pattern.
But the best grammar also has a better (smaller) value for E and the reason for that
is less obvious. Here are the two relevant alignments:

For the best grammar:

ID761 : ID750 : ID461 : #1925 NSC = 5262.41, EC = 55.44, CR = 94.93, CD = 5206.97,
Absolute P = 0.00

```
0                   w e             r u n               f a s t   0
                    | |             | | |               | | | |
1      < 27 4 24 1 #4 w e >         | | |               | | | |   1
       |    |            |          | | |               | | | |
2      |    |            | < 25 2 3 #5 r u n >           | | | |   2
       |    |            | |    |          |             | | | |
3      |    |            | |    |          |   < 26 23 #33 f a s t > 3
       |    |            | |    |          |
4 < #2 <    4            > <    2          > >                     4
```

Number of unmatched code symbols = 11

For the hierarchical grammar:

ID807 : ID765 : ID478 : #2037 NSC = 5262.41, EC = 55.85, CR = 94.22, CD = 5206.55,
Absolute P = 0.00

```
0                           w e             r u n               f a s 0
                            | |             | | |               | | |
1              < 27 4 24 1 #4 w e >         | | |               | | | 1
               |        |        |          | | |               | | |
2              |        |        | < 25 2 3 #5 r u n >           | | | 2
               |        |        | |      |         |             | | |
3              |        |        | |      |         |   < 26 23 #33 f a s 3
               |        |        | |      |         |   |    |
4      < 22 #6 <        1        > <      3         > > |    |           4
       | |                                             | |    |
5 < #34 < 22                                           > <    23          5


0 t     0
  |
1 |     1
  |
2 |     2
  |
```

```
3 t >   3
    |
4   |   4
    |
5   > > 5

Number of unmatched code symbols = 11

The number of unmatched ID symbols is the same in both cases so the difference in
E values must be down to differences in weightings and E values from other parsings.

The best grammar is derived from GR114 which has these values:
G = 7722.39, E = 137.71, score = 7860.10, while the hierarchical grammar is derived
from GR121 which has these values: G = 7728.49, E = 156.85, score = 7885.34. So
the advantage is also in the precursor grammars, which are:

GRAMMAR GR114, derived from grammar GR105 and alignment:
ID702 (< #2 < 1 4 #1 t h e y > < 2 3 #3 w a l k > >)*1

G = 7722.39, E = 137.71, score = 7860.10.

Grammar patterns:

ID452 (< 27 4 24 1 #4 w e >)*3

ID453 (< 25 2 3 #5 r u n >)*3

ID461 (< #2 < 4 > < 2 > >)*5

ID451 (< 2 3 #3 w a l k >)*2

ID455 (< 1 4 #1 t h e y >)*2

and

GRAMMAR GR121, derived from grammar GR109 and alignment:
ID703 (< 22 #6 < 1 4 #1 t h e y > < 2 3 #3 w a l k > >)*1

G = 7728.49, E = 156.85, score = 7885.34.

Grammar patterns:

ID452 (< 27 4 24 1 #4 w e >)*3

ID453 (< 25 2 3 #5 r u n >)*3

ID454 (< 22 #6 < 1 > < 3 > >)*5

ID451 (< 2 3 #3 w a l k >)*2

ID455 (< 1 4 #1 t h e y >)*2

The critical difference is between ID461 (< #2 < 4 > < 2 > >)*5 and
ID454 (< 22 #6 < 1 > < 3 > >)*5. ID454 has 2 ID symbols while ID461 has
only 1. Since these two patterns are equivalent, it would be useful to know
why the program has made two equivalent patterns and how it might be modified
to avoid this.

2 To avoid confusing changes in numbering, the tidy_up_code_symbols() method
has been excluded from the program. Also, copying of patterns when the two
best grammars are merged is suppressed. In this case, GR114 is:

GRAMMAR GR114, derived from grammar GR105 and alignment:
ID669 (< #16 < 1 9 #15 t h e y > < 10 2 #3 w a l k > >)*1

G = 7722.39, E = 137.71, score = 7860.10.

Grammar patterns:

ID17 (< 27 9 24 1 #4 w e >)*3

ID18 (< 25 10 2 #5 r u n >)*3

ID108 (< #16 < 9 > < 10 > >)*5

ID15 (< 10 2 #3 w a l k >)*2

ID107 (< 1 9 #15 t h e y >)*2

and GR121 is:

GRAMMAR GR121, derived from grammar GR109 and alignment:
ID670 (< 22 #6 < 1 9 #15 t h e y > < 10 2 #3 w a l k > >)*1
```

```
G = 7728.49, E = 156.85, score = 7885.34.

Grammar patterns:

ID17 (< 27 9 24 1 #4 w e >)*3

ID18 (< 25 10 2 #5 r u n >)*3

ID19 (< 22 #6 < 1 > < 2 > >)*5

ID15 (< 10 2 #3 w a l k >)*2

ID107 (< 1 9 #15 t h e y >)*2

So the two 'critical' patterns are ID108 (< #16 < 9 > < 10 > >)*5 and
ID19 (< 22 #6 < 1 > < 2 > >)*5. As before, the one leading to the hierarchical
grammar has two ID symbols while the one leading to the best grammar has
only one. Here are the origins of these two patterns:

ID19:

Learning from alignment ID10:

0      w e w a l k 0
       | |
1 < #1 w e r u n > 1

Part patterns from New:

ID15 (< #3 w a l k >)*1

Start of process_old_pattern().

Part patterns for Old pattern in row 1 (alignment ID10):

ID17 (< #4 w e >)*1

ID18 (< #5 r u n >)*1

Context symbol 1 added to pattern ID17 (< 1 #4 w e >)*1

Context symbol 2 added to pattern ID18 (< 2 #5 r u n >)*1

Context symbol 2 added to pattern ID15 (< 2 #3 w a l k >)*1

ID19 (< #6 < 1 > < 2 > >)*1

and ID108:

Learning from alignment ID103:

0 t h e y r u n   0
          | | |
1 < 2 #5  r u n > 1

Part patterns from New:

ID107 (< #15 t h e y >)*1

Start of aggregate_part_patterns_and_alignments().

Context symbol 9 added to pattern ID107 (< 9 #15 t h e y >)*1

Context symbol 10 added to pattern ID18 (< 10 2 #5 r u n >)*1

ID108 (< #16 < 9 > < 10 > >)*1

It is not obvious how to prevent the program forming equivalent abstract
patterns like these. But it should not matter on the broad scale because
grammars containing two equivalent patterns are likely to score worse
than ones with only one and so the latter kinds of grammar should win
overall.

RESULTS FOR SP71, V 7.12 WITH LARGER SAMPLE

With this input:

[
    [
        (w e r u n)
        (w e w a l k)
        (t h e y r u n)
```

```
        (t h e y w a l k)
        (w e r u n f a s t)
        (w e r u n s l o w l y)
        (w e w a l k f a s t)
        (w e w a l k s l o w l y)
        (t h e y r u n f a s t)
        (t h e y r u n s l o w l y)
        (t h e y w a l k f a s t)
        (t h e y w a l k s l o w l y)
    ]
    [
    ]
]

the best two grammars formed by the program are:

GRAMMAR GR1315, derived from grammar GR1215 and alignment:
ID6353 (< 39 #16 < 194 189 186 184 9 #15 t h e y > < 187 10 #3 w a l k > >
< 188 179 40 #54 s l o w l y >)*1

G = 12794.73, E = 712.79, score = 13507.52.

Grammar patterns:

ID22 (< 187 10 #3 w a l k >)*6

ID24 (< 9 #4 w e >)*6

ID25 (< 10 #5 r u n >)*6

ID114 (< 194 189 186 184 9 #15 t h e y >)*6

ID115 (< 39 #16 < 9 > < 10 > >)*12

ID451 (< 23 #33 f a s t >)*4

ID811 (< 188 179 40 #54 s l o w l y >)*4

GRAMMAR GR1306, derived from grammar GR1216 and alignment:
ID6420 (< #55 < 39 #16 < 194 189 186 184 9 #15 t h e y > < 187 10 #3 w a l k > >
< 188 179 40 #54 s l o w l y > >)*1

G = 12830.42, E = 694.16, score = 13524.59.

Grammar patterns:

ID22 (< 187 10 #3 w a l k >)*6

ID24 (< 9 #4 w e >)*6

ID25 (< 10 #5 r u n >)*6

ID114 (< 194 189 186 184 9 #15 t h e y >)*6

ID115 (< 39 #16 < 9 > < 10 > >)*12

ID451 (< 23 #33 f a s t >)*4

ID811 (< 188 179 40 #54 s l o w l y >)*4

ID812 (< #55 < 39 > < 40 > >)*4

The second-best grammar is hierarchical!

As before, the hierarchical grammar has a larger G because it has an extra
pattern (the G values are 12794.73 and 12830.42). But it has a smaller E
value, presumably because the hierarchical structure allows the New patterns
to be encoded more efficiently (the two values are 712.79 and 694.16). The
two scores (13507.52 and 13524.59) are close. These results are obtained
with a cost_factor of 100.00.

Increasing the size of the cost_factor to 1000 still results in the
hierarchical grammar coming second because the E values are the same and
the G values, although larger, have the same relative sizes.

Changes in cost_factor do not seem to put the hierarchical grammar at
the top. When the cost factor is reduced to 5, other less plausible
grammars are produced.

%95 10/2/05

FURTHER NOTES ON SP71, V 7.12
```

With this input:

```
[
    [
        (w e r u n f a s t)
        (w e r u n s l o w l y)
        (w e w a l k f a s t)
        (w e w a l k s l o w l y)
        (t h e y r u n f a s t)
        (t h e y r u n s l o w l y)
        (t h e y w a l k f a s t)
        (t h e y w a l k s l o w l y)
        (w e r u n)
        (w e w a l k)
        (t h e y r u n)
        (t h e y w a l k)
    ]
    [
    ]
]
```

the program creates these best three grammmars:

```
GRAMMAR GR5067, derived from grammar GR5009 and alignment:
ID12033 (< #249 < 193 #247 t h e > < 194 186 #235 y > >
< 181 179 150 149 #21 w a l k >)*1

G = 13747.92, E = 710.89, score = 14458.81.

Grammar patterns:

ID156 (< 181 179 150 149 #21 w a l k >)*6

ID158 (< 148 13 #22 w e >)*6

ID159 (< 150 149 #23 r u n >)*6

ID160 (< 15 #24 f a s t >)*4

ID921 (< 15 #71 s l o w l y >)*4

ID6776 (< #185 < 148 > < 149 > >)*12

ID10737 (< 194 186 #235 y >)*8

ID10752 (< 193 #247 t h e >)*6

ID10757 (< #249 < 193 > < 194 > >)*8

GRAMMAR GR5068, derived from grammar GR5010 and alignment:
ID12033 (< #249 < 193 #247 t h e > < 194 186 #235 y > >
< 181 179 150 149 #21 w a l k >)*1

G = 13747.92, E = 710.89, score = 14458.81.

Grammar patterns:

ID156 (< 181 179 150 149 #21 w a l k >)*6

ID158 (< 148 13 #22 w e >)*6

ID159 (< 150 149 #23 r u n >)*6

ID160 (< 15 #24 f a s t >)*4

ID921 (< 15 #71 s l o w l y >)*4

ID6782 (< #187 < 13 > < 150 > >)*12

ID10737 (< 194 186 #235 y >)*8

ID10752 (< 193 #247 t h e >)*6

ID10757 (< #249 < 193 > < 194 > >)*8

GRAMMAR GR5099, derived from grammar GR5017 and alignment:
ID12036 (< #185 < 182 180 178 148 13 #78 t h e y >
< 181 179 150 149 #21 w a l k > >)*1

G = 13716.25, E = 752.74, score = 14468.99.

Grammar patterns:
```

```
ID156 (< 181 179 150 149 #21 w a l k >)*6

ID158 (< 148 13 #22 w e >)*6

ID159 (< 150 149 #23 r u n >)*6

ID160 (< 15 #24 f a s t >)*4

ID921 (< 15 #71 s l o w l y >)*4

ID1951 (< 182 180 178 148 13 #78 t h e y >)*6

ID6776 (< #185 < 148 > < 149 > >)*12
```

Somewhat as expected, none of them are hierarchical (in the sense
that none of them have the three levels we are looking for).

It is a bit puzzling why 't h e y' is broken into two pieces in the
first two grammars but not in the third. It may have something to do
with the rather large number of ID symbols for ID1951 in the third
grammar compared with the relative few for the three patterns:

```
ID10737 (< 194 186 #235 y >)*8

ID10752 (< 193 #247 t h e >)*6

ID10757 (< #249 < 193 > < 194 > >)*8
```

Also, it is not entirely clear why the program is not finding plausible
hierarchical grammars because, in its learning phase, it is making the
right kinds of structures as can be seen with pattern ID9:

```
Learning from alignment ID6746:

0               w e              r u n           0
                | |              | | |
1       < 13 #22 w e >           | | |           1
        | |         |            | | |
2       | |         | < 77 14 #23 r u n >         2
        | |         | |    |           |
3 < #25 < 13        > <    14          > < 15 > > 3

Part patterns from New:

Start of process_old_pattern().

Part patterns for Old pattern in row 3 (alignment ID6746):

ID6754 (< #182 < 13 > < 14 > >)*1

ID6755 (< 15 >)*1

Context symbol 145 added to pattern ID6754 (< 145 #182 < 13 > < 14 > >)*1

Single reference pattern ID6755 is deleted.

ID6756 (< #183 < 145 > < 15 > >)*1

End of process_old_pattern().
```

and it is creating corresponding full alignments in Phase 2:

```
ID6911: (< #183 < 145 #182 < 148 76 146 13 #22 w e >
< 153 150 149 147 77 14 #23 r u n > > < 78 15 #24 f a s t > >)*1
```

%96 14/2/05

FURTHER NOTES ON SP71, V 7.12

As noted in %95, the program forms 'correct' hierarchical patterns when
the small sentences are presented after the larger ones. But it is still
failing to find hierarchical grammars. Here are some possible reasons.

A test input file has been prepared like this:

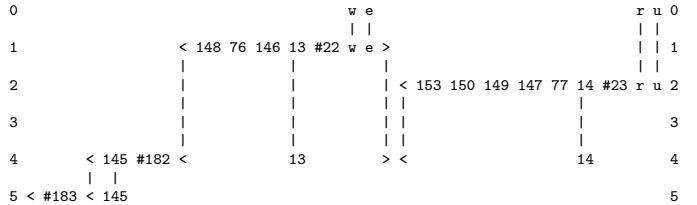
```
[
    [
        (w e r u n f a s t)
        (w e r u n s l o w l y)
        (w e w a l k f a s t)
        (w e w a l k s l o w l y)
        (t h e y r u n f a s t)
        (t h e y r u n s l o w l y)
```

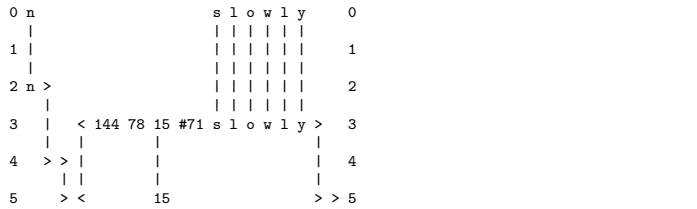
```
        (t h e y w a l k f a s t)
        (t h e y w a l k s l o w l y)
        (w e r u n)
    ]
    [
    ]
]
```

When we get to Phase 2 of the processing of the 9th pattern (’w e r u n’),
we get one ’correct’ full parsing like this:

SELECTED ALIGNMENT (PATTERN ID2 (ID9), CYCLE 3, PHASE 2)

ID7101: NSC = 6988.50, EC = 83.57, CR = 83.63, CD = 6904.94,
Absolute P = 6.98833677674e-026



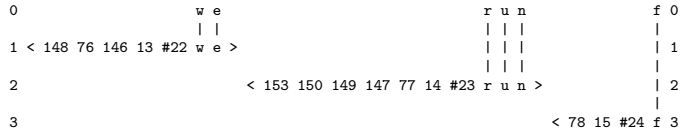


But the program fails to find the equivalent parsing for ’w e r u n f a s t’ even
though ’f a s t’ has the context symbol ’15’ in the pattern
ID160 (< #24 f a s t >)*1.

In broad terms, the reason seems to be that there are so many reasonably plausible
alternatives that the ’correct’ parsing is overwhelmed. Here are some indicative
parsings with the current parameters:

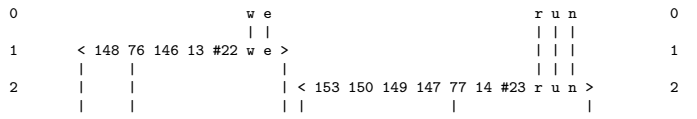
COMPOSITE ALIGNMENT (PATTERN ID1 (ID9), CYCLE 1, PHASE 2)

ID6820: NSC = 6301.56, EC = 83.20, CR = 75.74, CD = 6218.36,
Absolute P = 8.9900377241e-026



This composite alignment is a necessary precursor to the ’correct’ alignment.
But in the next cycle, the program finds alignments like these:

ID6831: NSC = 6301.56, EC = 73.10, CR = 86.20, CD = 6228.46,
Absolute P = 9.86909390439e-023



```
3       |     |               | |               |              | < 78 15 3
        |     |               | |               |              | | |
4 < #85 <     76              > <               77             > < 78    4


0     f a s t     0
      | | | |
1     | | | |     1
      | | | |
2     | | | |     2
      | | | |
3 #24 f a s t >   3
              |
4             > > 4
```

and

ID6834: NSC = 6301.56, EC = 74.89, CR = 84.15, CD = 6226.67,
Absolute P = 2.86242274375e-023

```
0                         w e                             r u n           0
                          | |                             | | |
1       < 148 76 146 13 #22 w e >                         | | |           1
        |           |         |                           | | |
2       |           |         | < 153 150 149 147 77 14 #23 r u n >         2
        |           |         | |                   |           |
3       |           |         | |                   |           | < 78 15 3
        |           |         | |                   |           | |    |
4 < #25 <           13        > <                   14          > <    15 4


0     f a s t     0
      | | | |
1     | | | |     1
      | | | |
2     | | | |     2
      | | | |
3 #24 f a s t >   3
              |
4             > > 4
```

and

ID6840: NSC = 6301.56, EC = 78.04, CR = 80.74, CD = 6223.52,
Absolute P = 3.20831971279e-024

```
0                           w e                             r u n         0
                            | |                             | | |
1       < 148 76 146 13 #22 w e >                           | | |         1
        |           |         |                             | | |
2       |           |         | < 153 150 149 147 77 14 #23 r u n >       2
        |           |         | |      |                          |
3       |           |         | |      |                          |   < 78 3
        |           |         | |      |                          |
4 < #187 <          13        > <     150                         > >     4


0       f a s t   0
        | | | |
1       | | | |   1
        | | | |
2       | | | |   2
        | | | |
3 15 #24 f a s t > 3

4                 4
```

and

ID6841: NSC = 6301.56, EC = 79.20, CR = 79.56, CD = 6222.36,
Absolute P = 1.43840603586e-024

```
0                           w e                             r u n         0
                            | |                             | | |
1       < 148 76 146 13 #22 w e >                           | | |         1
        |  |                  |                             | | |
2       |  |                  | < 153 150 149 147 77 14 #23 r u n >       2
        |  |                  | |          |                       |
3       |  |                  | |          |                       |   < 78 3
        |  |                  | |          |                       |
4 < #185 < 148                > <         149                      > >     4
```

```
0       f a s t   0
        | | | |
1       | | | |   1
        | | | |
2       | | | |   2
        | | | |
3 15 #24 f a s t > 3
```

but it fails to find the parsing which would be like ID6840 but with
ID6756 (< #183 < 145 > < 15 > >)*1 in the bottom row.

Possible solutions:

1 Increase the number of alignments that are selected on each cycle.

2 Find some way of reducing the creation of abstract patterns that are
distributionally equivalent or nearly so, such as:

ID162: (< #25 < 13 > < 14 > < 15 > >)*8
ID5108: (< #180 < 142 > < 143 > < 144 > >)*1
ID1965: (< #85 < 76 > < 77 > < 78 > >)*2

with

ID158: (< 148 76 146 13 #22 w e >)*4
ID1951: (< 142 76 13 #78 t h e y >)*4
ID156: (< 143 14 #21 w a l k >)*4
ID159: (< 153 150 149 147 77 14 #23 r u n >)*4
ID160: (< 78 15 #24 f a s t >)*4
ID921: (< 144 78 15 #71 s l o w l y >)*4

or

ID6754: (< 145 #182 < 13 > < 14 > >)*1
ID6763: (< #184 < 146 > < 147 > >)*1
ID6776: (< #185 < 148 > < 149 > >)*1

with the same patterns as above.

3 The number of equivalent or nearly-equivalent patterns may be reduced if the
program selects only *one* best grammar at the end of processing each
New pattern instead of two as at present.

4 Here is an example of how one of the near-equivalent abstract patterns
is created:

Learning from alignment ID6740:

```
0 w e         r u n   0
              | | |
1 < 77 14 #23 r u n > 1
```

Part patterns from New:

CONTENTS symbols of pattern ID6762 match those of
pre-existing pattern ID158 (< 13 #22 w e >)*4

Start of aggregate_part_patterns_and_alignments().

Context symbol 146 added to pattern ID158 (< 146 13 #22 w e >)*4

Context symbol 147 added to pattern ID159 (< 147 77 14 #23 r u n >)*4

ID6763 (< #184 < 146 > < 147 > >)*1

End of aggregate_part_patterns_and_alignments().

It is possible that it might not be necessary to create ID6763 if a search
could be made to see if there is an abstract pattern that fits this
composite pattern:

```
            w e                 r u n
            | |                 | | |
< 146 13 #22 w e >              | | |
                                | | |
                  < 147 77 14 #23 r u n >
```

5 When the number of alignments that are selected on each cycle is increased
and only one grammar is selected on each cycle, then the 'correct' parsings
are formed in Phase 2 of pattern ID9. But for some reason, the program is
still not forming 'correct' hierarchical grammars.

%97 21/2/05

```
FURTHER NOTES ON SP71, V 7.12

The program is producing identical grammars but with different scores.
For example, it produces:

GRAMMAR GR5262 (PATTERN ID9, PHASE 2)
derived from grammar GR5188 and alignment:
ID6312 (< #207 < 192 #206 < 195 193 13 #22 w e >
< 200 197 196 194 14 #23 r u n > > < 15
#68 s l o w l y > >)*1

G = 9810.91, E = 161.87, score = 9972.78.

Grammar patterns:

ID148 (< 195 193 13 #22 w e >)*5

ID149 (< 200 197 196 194 14 #23 r u n >)*5

ID150 (< 15 #24 f a s t >)*4

ID6034 (< 192 #206 < 13 > < 14 > >)*9

ID853 (< 15 #68 s l o w l y >)*4

ID6036 (< #207 < 192 > < 15 > >)*8

and

GRAMMAR GR5265 (PATTERN ID9, PHASE 2)
derived from grammar GR5184 and alignment:
ID6312 (< #207 < 192 #206 < 195 193 13 #22 w e >
< 200 197 196 194 14 #23 r u n > > < 15 #68 s l o w l y > >)*1

G = 9810.91, E = 157.75, score = 9968.65.

Grammar patterns:

ID148 (< 195 193 13 #22 w e >)*5

ID149 (< 200 197 196 194 14 #23 r u n >)*5

ID150 (< 15 #24 f a s t >)*4

ID6034 (< 192 #206 < 13 > < 14 > >)*9

ID6036 (< #207 < 192 > < 15 > >)*8

ID853 (< 15 #68 s l o w l y >)*4

The difference in scores is due to differences in E values and these are due to
differences in the alignments from which the grammars were derived.

GR5262 was derived from GR5188:

GRAMMAR GR5188 (PATTERN ID9, PHASE 2)
derived from grammar NULL and alignment:
ID6103 (< 192 #206 < 195 193 13 #22 w e > < 200 197 196 194 14 #23 r u n > >
< 15 #24 f a s t >)*1

G = 6087.19, E = 83.00, score = 6170.19.

Grammar patterns:

ID148 (< 195 193 13 #22 w e >)*5

ID149 (< 200 197 196 194 14 #23 r u n >)*5

ID150 (< 15 #24 f a s t >)*4

ID6034 (< 192 #206 < 13 > < 14 > >)*9

and GR5265 was derived from GR5184:

GRAMMAR GR5184 (PATTERN ID9, PHASE 2)
derived from grammar NULL and alignment:
ID6155 (< #207 < 192 #206 < 195 193 13 #22 w e > < 200 197 196 194 14 #23 r u n > >
< 15 #24 f a s t > >)*1

G = 6117.98, E = 78.87, score = 6196.85.

Grammar patterns:
```

```
ID148 (< 195 193 13 #22 w e >)*5

ID149 (< 200 197 196 194 14 #23 r u n >)*5

ID150 (< 15 #24 f a s t >)*4

ID6034 (< 192 #206 < 13 > < 14 > >)*9

ID6036 (< #207 < 192 > < 15 > >)*8

GR5188 is derived from an alignment (ID6103) that is contained within the alignment
from which GR5184 was compiled (ID6155).

During the process of compiling grammars, there seems to be a case for excluding
from consideration any alignment that is contained within another alignment for a
given pattern from New. This should help to simplify the process of compiling
grammars and should reduce the incidence of identical grammars derived via different
routes and identical grammars with different scores.

This will be attempted in v 7.13.

%98 23/2/05

NOTES ON SP71, V 7.13

The program is forming similar abstract patterns by alternative routes:

Learning from alignment ID130:

0      w e w a l k f a s t   0
       | |         | | | |
1 < #1 w e r u n   f a s t > 1

Part patterns from New:

ID169 (< #25 w a l k >)*1

Start of process_old_pattern().

Part patterns for Old pattern in row 1 (alignment ID130):

ID171 (< #26 w e >)*1

ID172 (< #27 r u n >)*1

ID173 (< #28 f a s t >)*1

Context symbol 14 added to pattern ID171 (< 14 #26 w e >)*1

Context symbol 15 added to pattern ID172 (< 15 #27 r u n >)*1

Context symbol 15 added to pattern ID169 (< 15 #25 w a l k >)*1

Context symbol 16 added to pattern ID173 (< 16 #28 f a s t >)*1

ID175 (< #29 < 14 > < 15 > < 16 > >)*1

and:

Learning from alignment ID1018:

0      w e w a l k s l o w l y   0
       | |         | | | | | |
1 < #2 w e r u n   s l o w l y > 1

Part patterns from New:

CONTENTS symbols of pattern ID1131 match those of
pre-existing pattern ID169 (< 15 #25 w a l k >)*1

Start of process_old_pattern().

Part patterns for Old pattern in row 1 (alignment ID1018):

CONTENTS symbols of pattern ID1133 match those of
pre-existing pattern ID171 (< 57 14 #26 w e >)*3

CONTENTS symbols of pattern ID1134 match those of
pre-existing pattern ID172 (< 15 #27 r u n >)*2

ID1135 (< #63 s l o w l y >)*1
```

```
Context symbol 59 added to pattern ID171 (< 59 57 14 #26 w e >)*3

Context symbol 60 added to pattern ID172 (< 60 15 #27 r u n >)*2

Context symbol 60 added to pattern ID169 (< 60 15 #25 w a l k >)*1

Context symbol 61 added to pattern ID1135 (< 61 #63 s l o w l y >)*1

ID1137 (< #64 < 59 > < 60 > < 61 > >)*1

These two abstract patterns then lead to two alternative hierarchical
structures via two different routes:

Learning from alignment ID9179:

0                   w e                       r u n            0
                    | |                       | | |
1       < 59 14 #26 w e >                     | | |            1
        | |             |                     | | |
2       | |             | < 193 152 60 15 #27 r u n >          2
        | |             | |         |               |
3 < #64 < 59            > <         60              > < 61 > > 3

Part patterns from New:

Start of process_old_pattern().

Part patterns for Old pattern in row 3 (alignment ID9179):

ID9202 (< #213 < 59 > < 60 > >)*1

ID9203 (< 61 >)*1

Context symbol 232 added to pattern ID9202 (< 232 #213 < 59 > < 60 > >)*1

Single reference pattern ID9203 is deleted.

ID9204 (< #214 < 232 > < 61 > >)*1

and:

Learning from alignment ID9180:

0                   w e                       r u n            0
                    | |                       | | |
1       < 59 14 #26 w e >                     | | |            1
        |    |          |                     | | |
2       |    |          | < 193 152 60 15 #27 r u n >          2
        |    |          | |            |            |
3 < #29 <    14         > <            15           > < 16 > > 3

Part patterns from New:

Start of process_old_pattern().

Part patterns for Old pattern in row 3 (alignment ID9180):

ID9207 (< #215 < 14 > < 15 > >)*1

ID9208 (< 16 >)*1

Context symbol 233 added to pattern ID9207 (< 233 #215 < 14 > < 15 > >)*1

Single reference pattern ID9208 is deleted.

ID9209 (< #216 < 233 > < 16 > >)*1

FURTHER NOTES

1 When we trace through the compiling of grammars, we find that 7 'correct' grammars
are created derived from alignments like: ID10561 (< #214 < 232 #213 < 192 151 1
< 238 231 153 61 16 #28 f a s t > >)*1

but the last of these does not proceed to be a basis for any later grammar because
of the GRAMMAR_LIMIT cut-off of 30. As the tree of alternative grammars expands,
the cut-off becomes progressively more severe. There may be a case for expressing
the cut-off as a percentage of the available grammars rather than an absolute number.

2 With the input sentences as they are now, the best grammar found is:

GRAMMAR GR13615 (PATTERN ID9, PHASE 2)
derived from grammar GR13435 and alignment:
ID10837 (< 239 234 #217 w > < 240 235 #219 e > <
```

```
G = 13663.67, E = 1203.49, score = 14867.16.

Grammar patterns:

ID169 (< 193 152 60 15 #25 w a l k >)*4

ID172 (< 245 242 237 193 152 60 15 #27 r u n >)*5

ID173 (< 238 231 153 61 16 #28 f a s t >)*4

ID1135 (< 243 229 194 61 #63 s l o w l y >)*4

ID9215 (< 239 234 #217 w >)*5

ID9216 (< 239 234 #218 t h >)*4

ID9217 (< 240 235 #219 e >)*9

ID9218 (< 241 236 #220 y >)*4

The reason for the fragmentation of 'w e' and 't h e y' seems to be that they
both contain 'e' and, with a cost factor of 100.0, it makes sense to share this
DATA_SYMBOL between the two words. This probably would not happen if we used
'w e' and 'y o u'.

After this change, the best grammar is:

GRAMMAR GR11489 (PATTERN ID9, PHASE 2)
derived from grammar GR11251 and alignment:
ID10042 (< 170 210 207 204 59 14 #26 w e > < 211 208 205 171 60 15 #27 r u n >)*1

G = 13097.82, E = 910.07, score = 14007.88.

Grammar patterns:

ID169 (< 171 60 15 #25 w a l k >)*4

ID171 (< 170 210 207 204 59 14 #26 w e >)*5

ID172 (< 211 208 205 171 60 15 #27 r u n >)*5

ID173 (< 209 201 16 #28 f a s t >)*4

ID1135 (< 206 199 172 61 #63 s l o w l y >)*4

ID2144 (< 207 204 170 59 14 #78 y o u >)*4

ID6480 (< #177 < 170 > < 171 > < 172 > >)*4

%99 24/2/05

FURTHER NOTES ON SP71, V 7.13

The program is forming 'correct' hierarchical structures and is incorporating
them into grammars, but these grammars are receiving relative high (poor) scores
and they are getting eliminated from the search before the final list of grammars
is formed. A likely reason for this is that the E value associated with each
full alignment is unnecessarily high. For example, compare this alignment:

ID8998: NSC = 5892.22, EC = 98.82, CR = 59.63, CD = 5793.40,
Absolute P = 1.78836529181e-030

0                                          w e                          0
                                           | |
1                  < 170 210 207 204 59 14 #26 w e >                    1
                   |                 |            |
2                  |                 |            | < 211 208 205 171 60 15 2
                   |                 |            | |                |
3                  |                 |            | |                |    3
                   |                 |            | |                |
4        < 202 #203 <                 59           > <                60   4
         |  |
5 < #204 < 202                                                          5


0     r u n                        s l o w l y     0
      | | |                        | | | | | |
1     | | |                        | | | | | |     1
      | | |                        | | | | | |
2 #27 r u n >                      | | | | | |     2
          |                        | | | | | |
3         |   < 206 199 172 61 #63 s l o w l y >   3
```

```
           |   |            |                |
4          > > |            |                |   4
             | |            |                |
5            > <            61               > > 5
```

with this one:

```
ID8936: NSC = 5892.22, EC = 99.06, CR = 59.48, CD = 5793.16,
Absolute P = 1.50971645409e-030

0                                  w e                             r u n   0
                                   | |                             | | |
1       < 170 210 207 204 59 14 #26 w e >                          | | |   1
        |                 |            |                          | | |
2       |                 |            | < 211 208 205 171 60 15 #27 r u n > 2
        |                 |            | |                 |               |
3       |                 |            | |                 |               | 3
        |                 |            | |                 |               |
4 < #64 <                 59           > <                 60              > 4


0                     s l o w l y     0
                      | | | | | |
1                     | | | | | |     1
                      | | | | | |
2                     | | | | | |     2
                      | | | | | |
3 < 206 199 172 61 #63 s l o w l y >   3
  |           |                  |
4 <           61                 > > 4
```

Both alignments have the same ID symbols except that the first one has '#204' and '#203' at the beginning whereas the first has only '#64' at the beginning. The symbol '#203' provides for the 'selection' of the hierarchical structure from alternatives that may fill the same slot. But ***there are no alternatives that may fill the same slot***! So it is not necessary to provide for anything other than '202' at the beginning of this pattern.

In general, we only need to provide 'discrimination' symbols when there are two or more alternatives for a given slot. The program will be modified so that patterns are only given unique ID symbols when there are two or more alternatives in a given slot or when it is a top level pattern. This should remove the cost penalty suffered by hierarchical parsings and hierarchical grammars.

%100

FURTHER NOTES ON SP71, V 7.13

With the modification described in the last section (and with the exclusion of composite alignments from learning) and with this input:

```
[
    [
        (w e r u n f a s t)
        (w e r u n s l o w l y)
        (w e w a l k f a s t)
        (w e w a l k s l o w l y)
        (y o u r u n f a s t)
        (y o u r u n s l o w l y)
        (y o u w a l k f a s t)
        (y o u w a l k s l o w l y)
        (w e r u n)
    ]
    [
    ]
]
```

The program now gives these three best grammars:

FINAL GRAMMARS (PATTERN ID9, PHASE 2):

GRAMMAR GR10066 (PATTERN ID9, PHASE 2)
derived from grammar GR9916 and alignment:
ID8856 (< 168 < 176 173 170 55 13 #23 w e > < 177 174 171 56 14 #17 r u n > >)*1

G = 12896.97, E = 765.94, score = 13662.91.

Grammar patterns:

ID161 (< 56 14 #18 w a l k >)*4

ID163 (< 176 173 170 55 13 #23 w e >)*5

```
ID164 (< 177 174 171 56 14 #17 r u n >)*5

ID165 (< 175 167 15 #144 f a s t >)*4

ID882 (< 172 165 57 #141 s l o w l y >)*4

ID1600 (< 173 170 55 13 #61 y o u >)*4

ID7531 (< 168 < 55 > < 56 > >)*9

GRAMMAR GR10097 (PATTERN ID9, PHASE 2)
derived from grammar GR9947 and alignment:
ID8853 (< 176 173 170 55 13 #23 w e > < 177 174 171 56 14 #17 r u n >)*1

G = 12866.99, E = 806.99, score = 13673.98.

Grammar patterns:

ID161 (< 56 14 #18 w a l k >)*4

ID163 (< 176 173 170 55 13 #23 w e >)*5

ID164 (< 177 174 171 56 14 #17 r u n >)*5

ID165 (< 175 167 15 #144 f a s t >)*4

ID882 (< 172 165 57 #141 s l o w l y >)*4

ID1600 (< 173 170 55 13 #61 y o u >)*4

GRAMMAR GR10070 (PATTERN ID9, PHASE 2)
derived from grammar GR9860 and alignment:
ID8856 (< 168 < 176 173 170 55 13 #23 w e > < 177 174 171 56 14 #17 r u n > >)*1

G = 12930.28, E = 747.96, score = 13678.24.

Grammar patterns:

ID161 (< 56 14 #18 w a l k >)*4

ID163 (< 176 173 170 55 13 #23 w e >)*5

ID164 (< 177 174 171 56 14 #17 r u n >)*5

ID165 (< 175 167 15 #144 f a s t >)*4

ID882 (< 172 165 57 #141 s l o w l y >)*4

ID1600 (< 173 170 55 13 #61 y o u >)*4

ID7531 (< 168 < 55 > < 56 > >)*9

ID7533 (< #146 < 168 > < 57 > >)*4

The hierarchical grammar now has a higher score and is the third in the final list
of grammars. There seems to be an anomaly in the frequencies of patterns: all of
them seem to be correct except for ID7533 which should be 9 instead of 4. This
needs checking out. If this error is corrected, it might be sufficient to put
the hierarchical grammar at the top of the list.

The reason for the low count is that ID7533 is used in only 4 FULL_A alignments.
The other 4 are ones like this:

ID5517: NSC = 5193.83, EC = 39.19, CR = 132.51, CD = 5154.64,
Absolute P = 1.58959743252e-012

0                            w e                 r u n          f a s t   0
                             | |                 | | |          | | | |
1             < 134 55 13 #23 w e >              | | |          | | | |   1
              |        |          |               | | |          | | | |
2             |        |          | < 141 56 14 #17 r u n >        | | | |   2
              |        |          | |        |            |        | | | |
3             |        |          | |        |            |   < 15 f a s t > 3
              |        |          | |        |            |   | |          |
4       < 132 <        13         > <        14           > > | |          | 4
        |  |                                                 | | |          |
5 < #119 < 132                                               > < 15         > 5


0   0

1   1
```

```
2   2

3   3

4   4

5 > 5
```

where the bottom pattern is derived like this:

```
Learning from alignment ID5334:

0                 y o u             w a l k               s l o w l y   0
                  | | |             | | | |               | | | | | |
1       < 55 13 #61 y o u >         | | | |               | | | | | |   1
        |    |           |         | | | |               | | | | | |
2       |    |           | < 56 14 #18 w a l k >         | | | | | |   2
        |    |           | |    |            |            | | | | | |
3       |    |           | |    |            |         < 57 s l o w l y > 3
        |    |           | |    |            |
4 < #19 <    13          > <    14           > < 15 > >                   4

Part patterns from New:

Start of process_old_pattern().

Part patterns for Old pattern in row 4 (alignment ID5334):

ID5351 (< < 13 > < 14 > >)*1

ID5352 (< 15 >)*1

Context symbol 132 added to pattern ID5351 (< 132 < 13 > < 14 > >)*1

Single reference pattern ID5352 is deleted.

ID5353 (< #119 < 132 > < 15 > >)*1

This is really spurious learning because alignment ID5334 is composite and ought to
be excluded from the learning process. The program will be modified to test each new
alignment to see if it is composite (not merely those that are created by
combine_alignments()) and all composite alignments will be excluded from learning.

After correction of two bugs, the two best grammars now found by the program are:

FINAL GRAMMARS (PATTERN ID9, PHASE 2):

GRAMMAR GR4349 (PATTERN ID9, PHASE 2)
derived from grammar GR4229 and alignment:
ID6911 (< 84 < 91 88 85 13 #23 w e > < 92 89 86 73 18 14 #17 r u n > >)*1

G = 12737.43, E = 862.79, score = 13600.23.

Grammar patterns:

ID168 (< 73 18 14 #18 w a l k >)*4

ID170 (< 91 88 85 13 #23 w e >)*5

ID171 (< 92 89 86 73 18 14 #17 r u n >)*5

ID172 (< 90 83 74 19 15 #77 f a s t >)*4

ID1420 (< 88 85 72 13 53 #44 y o u >)*4

ID5395 (< 87 81 #74 s l o w l y >)*4

ID5405 (< 84 < 13 > < 14 > >)*9

ID5407 (< #79 < 84 > < 15 > >)*4

GRAMMAR GR4350 (PATTERN ID9, PHASE 2)
derived from grammar GR4230 and alignment:
ID6911 (< 84 < 91 88 85 13 #23 w e > < 92 89 86 73 18 14 #17 r u n > >)*1

G = 12746.10, E = 862.79, score = 13608.89.

Grammar patterns:

ID168 (< 73 18 14 #18 w a l k >)*4

ID170 (< 91 88 85 13 #23 w e >)*5
```

```
ID171 (< 92 89 86 73 18 14 #17 r u n >)*5

ID172 (< 90 83 74 19 15 #77 f a s t >)*4

ID174 (< #19 < 13 > < 14 > < 15 > >)*4

ID1420 (< 88 85 72 13 53 #44 y o u >)*4

ID5395 (< 87 81 #74 s l o w l y >)*4

ID5405 (< 84 < 13 > < 14 > >)*9

With this input:

[
    [
        (w e r u n f a s t)
        (w e r u n s l o w l y)
        (w e w a l k f a s t)
        (w e w a l k s l o w l y)
        (y o u r u n f a s t)
        (y o u r u n s l o w l y)
        (y o u w a l k f a s t)
        (y o u w a l k s l o w l y)
        (w e r u n)
        (w e w a l k)
        (y o u r u n)
        (y o u w a l k)
    ]
    [
    ]
]

the two best grammars are:

FINAL GRAMMARS (PATTERN ID12, PHASE 2):

GRAMMAR GR8561 (PATTERN ID12, PHASE 2)
derived from grammar GR8411 and alignment:
ID12472 (< 134 84 < 150 147 85 13 #44 y o u > < 151 148 14 #18 w a l k > >)*1

G = 12172.42, E = 965.01, score = 13137.43.

Grammar patterns:

ID171 (< 151 148 14 #18 w a l k >)*6

ID173 (< 150 147 91 85 13 #23 w e >)*6

ID174 (< 92 86 14 #17 r u n >)*6

ID175 (< 152 133 15 #77 f a s t >)*4

ID177 (< #19 < 13 > < 14 > < 15 > >)*4

ID1423 (< 150 147 85 13 #44 y o u >)*6

ID5398 (< 149 131 87 #74 s l o w l y >)*4

ID5408 (< 134 84 < 13 > < 14 > >)*12

GRAMMAR GR8562 (PATTERN ID12, PHASE 2)
derived from grammar GR8412 and alignment:
ID12472 (< 134 84 < 150 147 85 13 #44 y o u > < 151 148 14 #18 w a l k > >)*1

G = 12163.07, E = 991.34, score = 13154.41.

Grammar patterns:

ID171 (< 151 148 14 #18 w a l k >)*6

ID173 (< 150 147 91 85 13 #23 w e >)*6

ID174 (< 92 86 14 #17 r u n >)*6

ID175 (< 152 133 15 #77 f a s t >)*4

ID1423 (< 150 147 85 13 #44 y o u >)*6

ID5398 (< 149 131 87 #74 s l o w l y >)*4

ID5408 (< 134 84 < 13 > < 14 > >)*12
```

ID10483 (< #110 < 134 > < 15 > >)*4

Here, the second one is hierarchical - but ID5398 is not encompassed within
the scope of ID10483.

%101 28/2/05

FURTHER NOTES ON SP71, V 7.13

In the last set of results in %100, the frequency of pattern ID10483, at 4,
seems to be still too low. If all the three-word sentences were parsed by
this pattern, the frequency should be 8.

The reason for this apparent anomaly is that, in Phase 2, 4 of the 8 three-word
sentences are indeed parsed using ID10483 but the other 4 are parsed like this:

ID11593: NSC = 6346.42, EC = 75.44, CR = 84.13, CD = 6270.98,
Absolute P = 1.95691232938e-023

```
0                            y o u                 r u n                 0
                             | | |                 | | |
1          < 150 147 85 13 #44 y o u >              | | |                 1
           |           |           |                | | |
2          |           |           | < 92 86 14 #17 r u n >               2
           |           |           | |       |            |
3          |           |           | |       |            |   < 149 131 87 3
           |           |           | |       |            |
4 < 134 84 <           13          > <       14           > >             4


0     s l o w l y   0
      | | | | | |
1     | | | | | |   1
      | | | | | |
2     | | | | | |   2
      | | | | | |
3 #74 s l o w l y > 3

4                   4
```

Although composite alignments like this one have been excluded from the learning
process in Phase 1, they are still used in Phase 2. The obvious answer, to exclude
them from Phase 2, is probably wrong because composite alignments that are also
FULL_A represent legitimate analyses for the purpose of compiling grammars. Indeed,
some of them have already been excluded by the recently-introduced rule that
excludes parsings that are part of larger parsings and there seems to be a case
for dropping this rule because it may exclude legitimate analyses of the New
patterns and thus distort the final choice of grammars (this will be done).

Question: In the two best grammars, GR8561 and GR8562, why is not 's l o w l y'
marked with the category '15' and thus encompassed within the scope of the
hierarchical pattern ID10483?

The reason seems to be that, in alignments like this:

SELECTED ALIGNMENT (PATTERN ID2, CYCLE 1, PHASE 1)

ID16: NSC = 2008.32, EC = 3.58, CR = 0.00, CD = 2004.73,
Absolute P = 0.0833333333333

```
0      w e r u n     s l o w l y 0
       | | | | |     |
1 < #1 w e r u n f a s t >       1
```

the letter 's' that appears in both 'f a s t' and 's l o w l y' prevents the
system finding the 'correct' analysis that would put those two words in the
same category.

In a more realistic set of sentences, where a diversity of contexts might mean
that one or other of these two words would be assigned to a category and then
the other one would be assigned to that category later, should get round this
problem because one or other of these words would appear 'opposite' a context
symbol rather than a specific word - and so the 'spurious' matching of the
two 's's would not occur.

With this reduced set of sentences, we may need to adopt a fudge: to choose
words where the above effect does not occur, eg substituting 'w e l l'
for 's l o w l y'.

%102 1/3/05

FURTHER NOTES ON SP71, V 7.13

```
Probably the main reason why this version of the program produces results
that are not totally 'correct' is that it lacks any ability to generalise.
SNPR was successful because it could generalise in two different ways and it
was able to correct over-generalisations. The two methods for generalisation
are:

1 Abstract structures containing disjunctive *classes* of entities could be
concatenated as well as specific structures. This implies that all the different
combinations of class members would be legal even though only a subset of
those combinations had been observed.

2 When a new disjunctive class is formed, all syntagmatic structures that
contain any member of that class are modified so that the specific member
is replaced by a pointer to the class.

Overgeneralisations were corrected by keeping track of the the usage of every
member of every disjunctive class ***in every one of the different contexts in
which it occurs***. If a specifi entity fails to occur within a given class
***in a given context***, then a new class is created for that context which
excludes the given entity.

It seems likely that SP71 needs comparable abilities although they will probably
not be exact analogues. In particular, the mechanisms for rebuilding were very
cumbersome and something more streamlined is needed.

Here are some examples where an ability to generalise might help the
SP71 program:

Learning from alignment ID5360:

0      w e r u n              0
       | | | | |
1 < #2 w e r u n s l o w l y > 1

Part patterns from New:

Start of process_old_pattern().

Part patterns for Old pattern in row 1 (alignment ID5360):

CONTENTS symbols of pattern ID5397 match those of
pre-existing pattern ID5359 (< #73 w e r u n >)*1

ID5398 (< s l o w l y >)*1

Context symbol 80 added to pattern ID5359 (< 80 #73 w e r u n >)*1

Context symbol 81 added to pattern ID5398 (< 81 s l o w l y >)*1

Unique ID symbol #74 added to pattern ID5398 (< 81 #74 s l o w l y >)*1

Context symbol 81 added to pattern ID5400 (< 81 >)*1

Unique ID symbol #75 added to pattern ID5400 (< 81 #75 >)*1

ID5399 (< #76 < 80 > < 81 > >)*1

and:

Learning from alignment ID5361:

0      w e r u n          0
       | | | | |
1 < #1 w e r u n f a s t > 1

Part patterns from New:

Start of process_old_pattern().

Part patterns for Old pattern in row 1 (alignment ID5361):

CONTENTS symbols of pattern ID5403 match those of
pre-existing pattern ID5359 (< 80 #73 w e r u n >)*1

CONTENTS symbols of pattern ID5404 match those of
pre-existing pattern ID175 (< 74 19 15 f a s t >)*4

Context symbol 82 added to pattern ID5359 (< 82 80 #73 w e r u n >)*1

Context symbol 83 added to pattern ID175 (< 83 74 19 15 f a s t >)*4

Unique ID symbol #77 added to pattern ID175 (< 83 74 19 15 #77 f a s t >)*4
```

```
Context symbol 83 added to pattern ID5400 (< 83 81 #75 >)*1

ID5405 (< #78 < 82 > < 83 > >)*1

In these two examples, 'w e r u n' is assigned to context class 80 in the
first case and to context class 82 in the second. Also, 's l o w l y' and
'f a s t' are assigned to different context classes even though they can
both occur immediately to the right of 'w e r u n'.

A possible way forward is to generalise in a manner that is similar to the
second of the two methods in SNPR except that generalisation would be done
by creating new abstract patterns and leaving the original abstract patterns
in the grammar. Then correction of over-generalisations should occur in the
process of compiling alternative grammars: patterns that are too general,
and therefore inefficient for the encoding of New patterns, should be excluded
from the best grammars. This method of correcting overgeneralisations should
be much less cumbersome than the method used in program SNPR.

These ideas will be attempted in SP71, v 8.0.

FURTHER THOUGHTS

As it stands, SP71, v 7.13 already has an ability to generalise. This can be
seen by comparing the results from this input:

[
    [(j o h n r u n s)
    (j o h n w a l k s)
    (m a r y r u n s)
    (m a r y w a l k s)
    ]
    [
    ]
]

where the best grammar found is:

GRAMMAR GR802 (PATTERN ID4, PHASE 2)
derived from grammar GR516 and alignment:
0ID1190 (< #41 < 47 44 41 39 36 4 #29 m a r y > < 45 42 5 #9 w a l k >
< 46 43 38 6 #5 s > >)*1

G = 10239.28, E = 320.40, score = 10559.68.

Grammar patterns:

ID8 (< 46 43 38 6 #5 s >)*4

ID47 (< 45 42 5 #9 w a l k >)*2

ID49 (< 39 4 #28 j o h n >)*2

ID50 (< 45 37 5 #8 r u n >)*2

ID53 (< #10 < 4 > < 5 > < 6 > >)*4

ID476 (< 47 44 41 39 36 4 #29 m a r y >)*2

ID876 (< #41 < 44 > < 45 > < 46 > >)*2

and the results with this input:

[
    [(j o h n r u n s)
    (j o h n w a l k s)
    (m a r y r u n s)
    ]
    [
    ]
]

where the best grammar found is:

FINAL GRAMMARS (PATTERN ID3, PHASE 2):

GRAMMAR GR269 (PATTERN ID3, PHASE 2)
derived from grammar GR154 and alignment:
ID731 (< #10 < 36 4 2 29 27 #29 m a r y > < 37 5 #8 r u n >
< 38 6 3 #5 s > >)*1

G = 9792.21, E = 206.25, score = 9998.46.

Grammar patterns:
```

```
ID7 (< 38 6 3 #5 s >)*3

ID46 (< 5 #9 w a l k >)*1

ID48 (< 31 27 12 4 #28 j o h n >)*2

ID49 (< 37 5 #8 r u n >)*2

ID52 (< #10 < 4 > < 5 > < 6 > >)*3

ID475 (< 36 4 2 29 27 #29 m a r y >)*1

The generalisation seems to arise from alignments like this:

Learning from alignment ID462:

0 m a r y           r u n           s     0
                    | | |           |
1            < 5 #8 r u n >         |     1
             | |          |         |
2            | |          | < 6 3 #5 s >   2
             | |          | | |       |
3 < #10 < 4 > < 5         > < 6       > > 3

Part patterns from New:

CONTENTS symbols of pattern ID496 match those of
pre-existing pattern ID475 (< 2 29 27 #29 m a r y >)*1

Start of process_old_pattern().

Part patterns for Old pattern in row 3 (alignment ID462):

ID497 (< 4 >)*1

ID498 (< < 5 > < 6 > >)*1

Context symbol 4 added to pattern ID475 (< 4 2 29 27 #29 m a r y >)*1

Single reference pattern ID497 is deleted.

Context symbol 35 added to pattern ID498 (< 35 < 5 > < 6 > >)*1

ID500 (< #36 < 4 > < 35 > >)*1

End of process_old_pattern().

Here, 'm a r y' is assigned to the context class '4' and this implies
that the sentence 'm a r y w a l k s' is legal even though it has not
been observed.

%103 13/3/05

NOTES ON SP71, V 8.0

A residual problem from v 7.9 is that, with the set of New patterns
containing short and long sentences, the program fails to recognise that
ID5398 (< 149 131 87 #74 s l o w l y >)*4 belongs in the same class as
ID175 (< 152 133 15 #77 f a s t >)*4 at the end of the 'long' sentence
pattern. This problem may be solved by introducing (more) generalisation
but in any case it is worth looking at the output file in detail to see
why the program is failing in this way.

'f a s t' is first isolated like this:

Learning from alignment ID130:

0      w e w a l k f a s t   0
       | |         | | | |
1 < #1 w e r u n   f a s t > 1

Part patterns from New:

ID171 (< w a l k >)*1

Start of process_old_pattern().

Part patterns for Old pattern in row 1 (alignment ID130):

ID173 (< w e >)*1

ID174 (< r u n >)*1
```

```
ID175 (< f a s t >)*1

Context symbol 13 added to pattern ID173 (< 13 w e >)*1

Context symbol 14 added to pattern ID174 (< 14 r u n >)*1

Unique ID symbol #17 added to pattern ID174 (< 14 #17 r u n >)*1

Context symbol 14 added to pattern ID171 (< 14 w a l k >)*1

Unique ID symbol #18 added to pattern ID171 (< 14 #18 w a l k >)*1

Context symbol 15 added to pattern ID175 (< 15 f a s t >)*1

ID177 (< #19 < 13 > < 14 > < 15 > >)*1

But 's l o w l y' is first isolated like this:

Learning from alignment ID5360:

0      w e r u n              0
       | | | | |
1 < #2 w e r u n s l o w l y > 1

Part patterns from New:

Start of process_old_pattern().

Part patterns for Old pattern in row 1 (alignment ID5360):

CONTENTS symbols of pattern ID5397 match those of
pre-existing pattern ID5359 (< #73 w e r u n >)*1

ID5398 (< s l o w l y >)*1

Context symbol 80 added to pattern ID5359 (< 80 #73 w e r u n >)*1

Context symbol 81 added to pattern ID5398 (< 81 s l o w l y >)*1

Unique ID symbol #74 added to pattern ID5398 (< 81 #74 s l o w l y >)*1

Context symbol 81 added to pattern ID5400 (< 81 >)*1

Unique ID symbol #75 added to pattern ID5400 (< 81 #75 >)*1

ID5399 (< #76 < 80 > < 81 > >)*1

It is not clear why '81' is present in < 81 #74 s l o w l y > at this stage
but absent from the pattern as it appears in the second-best grammar at
the end of processing.

Let us trace through the exact development of 'f a s t'.

After it is first isolated as '15 f a s t >' it becomes:

< 19 15 f a s t >
< 23 19 15 f a s t >
< 61 23 19 15 f a s t >
< 63 61 23 19 15 f a s t >
< 19 15 f a s t > (due to cleaning)
< 74 19 15 f a s t >
< 79 74 19 15 f a s t >
< 74 19 15 f a s t > (cleaning)
< 83 74 19 15 #77 f a s t >
and so on.

It seems that the only thing that would link 'f a s t' to 's l o w l y' is
that they both have a disjunctive relation with a null pattern derived
from alignments like this:

Learning from alignment ID5361:

0      w e r u n          0
       | | | | |
1 < #1 w e r u n f a s t > 1

with the null pattern '< 83 81 #75 >'

and

Learning from alignment ID5360:
```

```
0     w e r u n             0
      | | | | |
1 < #2 w e r u n s l o w l y > 1
```

with the null pattern '< 81 #75 >'.

Here, the context symbol '81' provides a link between 'f a s t' and 's l o w l y'.

If the short sentences occur before the long ones, it should be easier to find the link between 'f a s t' and 's l o w l y' because the short sentences would have given rise to abstract patterns and these would be used in the analysis of the longer sentences. This would mean that there would be more possibilities for 'f a s t' and 's l o w l y' to occur in the same context (because the context would be abstract, not specific, and would therefore apply to more cases).

Question: why is the program not forming alignments like this:

```
    w e r u n s l o w l y
    | | | | |
< #1 w e r u n f a s t >
```

The probable answer is that it is forming alignments like this:

```
    w e r u n    s l o w l y
    | | | | |    |
< #1 w e r u n f a s t >
```

and so it is missing the 'correct' alignment that would give rise to the 'correct' classification of 'f a s t' and 's l o w l y'.

This is indeed the case:

SELECTED ALIGNMENT (PATTERN ID2, CYCLE 1, PHASE 1)

ID16: NSC = 2008.32, EC = 3.58, CR = 0.00, CD = 2004.73,
Absolute P = 0.0833333333333

```
0     w e r u n    s l o w l y 0
      | | | | |    |
1 < #1 w e r u n f a s t >      1
```

The approximate metric for evaluating alignments should be biased against alignments that lead to fragmentation, but only if the cost factor is reasonably low so that the high weight of New symbols does not swamp everything.

Lowering the cost factor does not produce any grammar in which 'f a s t' and 's l o w l y' are classified together.

PROPOSALS FOR GENERALISATION

1 If an unmatched New pattern is opposite an unmatched reference in Old, then assign the context symbol of the reference to the New pattern. This is already implemented and already demonstrates generalisation with very simple four-sentence inputs.

2 When a new context symbol is assigned to a pattern, then assign the same context symbol to all patterns that have any other context symbol in common with the given pattern. This is rather 'massive' generalisation but the process of sifting and sorting should still enable us to arrive at plausible grammars.

Although a lot of patterns with a given context symbol will be irrelevant to that context, patterns like that should get zero frequency in the FULL_A parses of sifting and sorting and so they should disappear from view. This seems to be equivalent to the process of monitoring and rebuilding in SNPR but does not require the awkward process of tracing all uses of patterns in every different context.

%104 14/3/05

GENERALISATION (CONTINUED)

A question that arises relating to generalisation is "Why not generalise to the maximum possible extent and assign every pattern to every context?" It looks as if this extreme kind of over-generalisation should not prevent the system finding 'correct' grammars because all kinds of overgeneralisation should be removed during the sifting and sorting phase. Probably the main objection to doing this is that it would lead to each pattern having a context symbol for every context and this could become rather cumbersome.

Another question is "How do the forms of generalisation proposed under 1

and 2 in %103 compare with the kinds of generalisation used in SNPR?"

The first kind of generalisation, already implemented in SP71, is similar
to the way SNPR achieves generalisation by concatenation of patterns. In
both cases, the effect is to create a structure that may generate sentences
that have not been observed. This can be justified in terms of MLE principles
if the effect is an overall reduction in (G + E).

The second form of generalisation in SNPR meant, for each new class, looking
for 'inline' members of the class and converting them into the class. This
appears to be more conservative than what is proposed under '2' above but
this should not matter.

Question: "What exactly is needed in the case of 'f a s t' and 's l o w l y'?"
The argument, before is that '81' provides the link between the two, via the
null pattern. So the procedure should be:

1 For any given pattern to which a given context symbol is added (eg 83
added to '< 81 #75 >'), make a list of the other context symbols in the
given pattern (excluding unique ID symbols). For further additions to
the list, avoid duplicates.

2 For all the patterns in Old and for all the context symbols that have been
listed, add the given context symbol to any pattern that contains one of the
listed context symbols. Always check for duplicates. For example, add '83' to
any pattern that contains '81'. This will ensure that 'f a s t' and
's l o w l y' both get to have the context symbol '83'.

SP71, v 8.0 will be adapted as described under '2' in %103.

%105 15/3/05

NOTES ON SP71, V 8.0

With generalisation in place as described in %104, the two best grammars
found are:

GRAMMAR GR12783 (PATTERN ID12, PHASE 2)
derived from grammar GR12573 and alignment:
ID16538 (< 123 122 < 122 104 90 #49 y o u > < 123 105 91 52 #18 w a l k > >)*1

G = 12784.30, E = 1032.17, score = 13816.48.

Grammar patterns:

ID179 (< 123 105 91 52 #18 w a l k >)*6

ID181 (< 122 104 90 51 #23 w e >)*6

ID182 (< 123 105 91 52 #17 r u n >)*6

ID183 (< 157 153 149 147 145 124 #92 f a s t >)*4

ID1005 (< 157 153 149 147 145 124 #89 s l o w l y >)*4

ID1574 (< 122 104 90 #49 y o u >)*6

ID7842 (< 123 122 < 104 > < 105 > >)*12

GRAMMAR GR12754 (PATTERN ID12, PHASE 2)
derived from grammar GR12544 and alignment:
ID16537 (< 129 < 122 104 90 #49 y o u > < 123 105 91 52 #18 w a l k > >)*1

G = 12814.70, E = 1021.99, score = 13836.69.

Grammar patterns:

ID179 (< 123 105 91 52 #18 w a l k >)*6

ID181 (< 122 104 90 51 #23 w e >)*6

ID182 (< 123 105 91 52 #17 r u n >)*6

ID183 (< 157 153 149 147 145 124 #92 f a s t >)*4

ID1005 (< 157 153 149 147 145 124 #89 s l o w l y >)*4

ID1574 (< 122 104 90 #49 y o u >)*6

ID7842 (< 123 122 < 104 > < 105 > >)*12

ID10883 (< 129 < 122 > < 123 > >)*12

The first one is sensible enough, providing an abstract pattern for the short sentences but not for the longer ones. But the second one is weird: the abstract pattern ID7842 is referenced *twice* in the abstract pattern ID10883. This demonstrates a hierarchy with more than 2 levels but it is not true to the original sentences. However, ID10883 also describes the two-word sentence pattern in a sensible way. It is derived like this:

Learning from alignment ID10852:

```
0                       y o u                     r u n           0
                        | | |                     | | |
1       < 122 104 90 #49 y o u >                  | | |           1
        |  |                  |                  | | |
2       |  |                  | < 123 105 91 52 #17 r u n >       2
        |  |                  | |  |                    |
3 < #112 < 122                > < 123                  > < 124 > > 3
```

Part patterns from New:

Start of process_old_pattern().

Part patterns for Old pattern in row 3 (alignment ID10852):

ID10883 (< < 122 > < 123 > >)*1

ID10884 (< 124 >)*1

Context symbol 129 added to pattern ID10883 (< 129 < 122 > < 123 > >)*1

Single reference pattern ID10884 is deleted.

ID10885 (< #117 < 129 > < 124 > >)*1

The apparently absurd double referencing of an abstract pattern (ID7842) within ID10883 is almost certainly due to inappropriate generalisations.

Here is how ID7842 is first created:

Learning from alignment ID7814:

```
0                      w e                 w a l k           0
                       | |                 | | | |
1       < 104 90 51 #23 w e >              | | | |           1
        |  |               |                | | | |
2       |  |               | < 105 91 52 #18 w a l k >        2
        |  |               | |  |                   |
3 < #96 < 104              > < 105                 > < 106 > > 3
```

Part patterns from New:

Start of process_old_pattern().

Part patterns for Old pattern in row 3 (alignment ID7814):

ID7842 (< < 104 > < 105 > >)*1

ID7843 (< 106 >)*1

Context symbol 111 added to pattern ID7842 (< 111 < 104 > < 105 > >)*1

Single reference pattern ID7843 is deleted.

ID7844 (< #101 < 111 > < 106 > >)*1

Later, it is generalised to: < 123 122 111 < 104 > < 105 > >

Then it is cleaned to become: < 123 122 < 104 > < 105 > >

The apparent reason that '123' and '122' are not cleaned out is because they appear to the program to be functional. This is just an accident of the fact that they are the same as ID-symbols that are functional within the grammar.

Two questions arise:

* Why did the original 'correct' hierarchical form get weeded out?

* How to avoid the kind of spurious overgeneralisation just described?

%106 16/3/05

GENERALISATION IN SP71, V8.0

The 'spurious' overgeneralisation noted in %105 occurs here:

```
OLD PATTERNS AFTER GENERALISATION:

[patterns removed]

ID7842: (< 111 < 104 > < 105 > >)*1

ID7844: (< #101 < 111 > < 106 > >)*1

[patterns removed]

Learning from alignment ID7766:

0 w e         w a l k              0
              | | | |
1 < #79 y o u w a l k s l o w l y > 1

Part patterns from New:

CONTENTS symbols of pattern ID7881 match those of
pre-existing pattern ID181 (< 104 90 51 #23 w e >)*5

Start of process_old_pattern().

Part patterns for Old pattern in row 1 (alignment ID7766):

CONTENTS symbols of pattern ID7882 match those of
pre-existing pattern ID1574 (< 104 90 #49 y o u >)*4

CONTENTS symbols of pattern ID7883 match those of
pre-existing pattern ID179 (< 105 91 52 #18 w a l k >)*4

CONTENTS symbols of pattern ID7884 match those of
pre-existing pattern
ID1005 (< 119 115 113 112 110 108 106 92 53 #89 s l o w l y >)*4

Context symbol 122 added to pattern ID1574 (< 122 104 90 #49 y o u >)*4

Context symbol 122 added to pattern ID181 (< 122 104 90 51 #23 w e >)*5

Context symbol 123 added to pattern ID179 (< 123 105 91 52 #18 w a l k >)*4

Context symbol 124 added to pattern
ID1005 (< 124 119 115 113 112 110 108 106 92 53 #89 s l o w l y >)*4

Context symbol 124 added to pattern
ID7834 (< 124 119 115 113 112 110 108 #98 >)*1

ID7885 (< #112 < 122 > < 123 > < 124 > >)*1

End of process_old_pattern().

Patterns created from alignment ID7766 (with details):

ID7885 (< #112 < 122 > < 123 > < 124 > >)*1

< (0, 2.37, LB, BM), #112 (1, 4.00, UID, ID),
< (2, 2.37, LB, CN), 122 (3, 4.00, CS, CN),
> (4, 2.37, RB, CN), < (5, 2.37, LB, CN),
123 (6, 4.00, CS, CN), > (7, 2.37, RB, CN),
< (8, 2.37, LB, CN), 124 (9, 4.00, CS, CN),
> (10, 2.37, RB, CN), > (11, 2.37, RB, BM), frequency = 1.

OLD PATTERNS AFTER GENERALISATION:

[patterns removed]

ID7842: (< 123 122 111 < 104 > < 105 > >)*1

ID7844: (< 124 #101 < 111 > < 106 > >)*1

[patterns removed]

It is not clear why '123' and '122' are added to ID7842.

The reason turns out to be a simple programming error: the program
is checking all symbols in relevant patterns, not just the ID-symbols.

With this error corrected, the best grammar found by the program is:

GRAMMAR GR13400 (PATTERN ID12, PHASE 2)
derived from grammar GR13160 and alignment:
ID18148 (< 138 < 188 167 164 129 126 #49 y o u >
```

```
< 189 168 165 130 127 #18 w a l k > >)*1

G = 13866.91, E = 1459.57, score = 15326.49.

Grammar patterns:

ID179 (< 189 168 165 130 127 #18 w a l k >)*6

ID181 (< 188 167 164 129 126 #23 w e >)*6

ID182 (< 189 168 165 130 127 #17 r u n >)*6

ID183 (< 190 185 181 177 173 171 169 166 131 128 #86 f a s t >)*4

ID996 (< 190 185 181 177 173 171 169 166 131 128 #83 s l o w l y >)*4

ID1565 (< 188 167 164 129 126 #49 y o u >)*6

ID11245 (< 138 < 129 > < 130 > >)*12

ID11247 (< #114 < 138 > < 131 > >)*8

Apart from excess ID-symbols, this is a 'correct' hierarchical grammar and
the program now assigns 's l o w l y' to the same class as 'f a s t' and
assigns both of them to the last slot in the grammar pattern.

After cleaning and tidying, the best three grammars are:

GRAMMAR GR13460 (PATTERN ID12, PHASE 2)
G = 13641.23, E = 1459.57, score = 15100.81.

Grammar patterns:

ID18201 (< 2 #3 w a l k >)*6

ID18202 (< 1 #4 w e >)*6

ID18203 (< 2 #2 r u n >)*6

ID18204 (< 3 #7 f a s t >)*4

ID18205 (< 3 #6 s l o w l y >)*4

ID18206 (< 1 #5 y o u >)*6

ID18207 (< 4 < 1 > < 2 > >)*12

ID18208 (< #1 < 4 > < 3 > >)*8

GRAMMAR GR13461 (PATTERN ID12, PHASE 2)
G = 13641.23, E = 1459.57, score = 15100.81.

Grammar patterns:

ID18209 (< 2 #3 w a l k >)*6

ID18210 (< 1 #4 w e >)*6

ID18211 (< 2 #2 r u n >)*6

ID18212 (< 3 #7 f a s t >)*4

ID18213 (< 3 #6 s l o w l y >)*4

ID18214 (< 1 #5 y o u >)*6

ID18215 (< 4 < 1 > < 2 > >)*12

ID18216 (< #1 < 4 > < 3 > >)*8

GRAMMAR GR13462 (PATTERN ID12, PHASE 2)
G = 13677.53, E = 1457.35, score = 15134.88.

Grammar patterns:

ID18217 (< 5 2 #3 w a l k >)*6

ID18218 (< 4 1 #4 w e >)*6

ID18219 (< 5 2 #2 r u n >)*6

ID18220 (< 6 #7 f a s t >)*4
```

```
ID18221 (< 6 #6 s l o w l y >)*4

ID18222 (< 4 1 #5 y o u >)*6

ID18223 (< 3 < 1 > < 2 > >)*12

ID18224 (< #1 < 4 > < 5 > < 6 > >)*8
```

The first two are exactly the same and are the 'correct' hierarchical grammars.

The third is 'correct' but it has two separate patterns for two-word and three-word sentences and different classes for each.

None of these grammars are the same as the best grammar before cleaning and tidying and the reason for this is not entirely clear. The reason is simply that the grammars are copied prior to cleaning and tidying which gives them new ID numbers. This has now been corrected so that the original ID numbers are used instead of the ID numbers of the copied grammars.

%107 17/3/05

FURTHER NOTES ON SP71, V 8.0

With this input:

```
[
    [
        (w e r u n)
        (w e w a l k)
        (y o u r u n)
        (y o u w a l k)
        (w e r u n f a s t)
        (w e r u n s l o w l y)
        (w e w a l k f a s t)
        (w e w a l k s l o w l y)
        (y o u r u n f a s t)
        (y o u r u n s l o w l y)
        (y o u w a l k f a s t)
        (y o u w a l k s l o w l y)
    ]
    [
    ]
]
```

the best grammar is:

```
GRAMMAR GR13447 (PATTERN ID12, PHASE 2)
derived from grammar GR12907 and alignment:
ID12798 (< #72 < 173 168 164 162 160 158 140 86 84 74 67 #14 y o u >
< 170 165 141 87 85 75 #63 w a l k > > < 177 171 166 142 88 76 #60 s l o w l y >)*1

G = 14498.43, E = 1554.68, score = 16053.11.

Grammar patterns:

ID22 (< 173 168 164 162 160 158 140 86 84 74 67 #6 w e >)*6

ID23 (< 170 165 141 87 85 75 68 #3 r u n >)*6

ID117 (< 173 168 164 162 160 158 140 86 84 74 67 #14 y o u >)*6

ID994 (< 177 171 166 142 88 76 f a s t >)*4

ID1493 (< 177 171 166 142 88 76 #60 s l o w l y >)*4

ID2153 (< 170 165 141 87 85 75 #63 w a l k >)*6

ID2181 (< #72 < 84 > < 85 > >)*12

ID10325 (< 167 y o >)*6

ID10327 (< 169 u >)*6
```

and the second best grammar is:

```
GRAMMAR GR13328 (PATTERN ID12, PHASE 2)
derived from grammar GR12908 and alignment:
ID12793 (< #123 < 173 168 164 162 160 158 140 86 84 74 67 #14 y o u >
< 170 165 141 87 85 75 #63 w a l k > < 177 171 166 142 88 76 #60 s l o w l y > >)*1

G = 14544.41, E = 1521.74, score = 16066.15.
```

```
Grammar patterns:

ID22 (< 173 168 164 162 160 158 140 86 84 74 67 #6 w e >)*6

ID23 (< 170 165 141 87 85 75 68 #3 r u n >)*6

ID117 (< 173 168 164 162 160 158 140 86 84 74 67 #14 y o u >)*6

ID994 (< 177 171 166 142 88 76 f a s t >)*4

ID1493 (< 177 171 166 142 88 76 #60 s l o w l y >)*4

ID2153 (< 170 165 141 87 85 75 #63 w a l k >)*6

ID2181 (< #72 < 84 > < 85 > >)*12

ID8000 (< #123 < 140 > < 141 > < 142 > >)*8

ID10325 (< 167 y o >)*6

ID10327 (< 169 u >)*6

and none of the subsequent grammars capture the hierarchical structure.

A new version of the program, v 8.1, will be started to try to fix this problem.

The 'wrong' patterns ID10325 and ID10327 arise like this:

Learning from alignment ID10122:

0       y o    u w a l k s l o w l y   0
        | |    |        | | | | | |
1 < #102 y o u r u n      s l o w l y > 1

Part patterns from New:

CONTENTS symbols of pattern ID10323 match those of
pre-existing pattern ID2153 (< 165 141 87 85 75 #63 w a l k >)*5

Start of process_old_pattern().

Part patterns for Old pattern in row 1 (alignment ID10122):

ID10325 (< y o >)*1

ID10326 (< u r >)*1

ID10327 (< u >)*1

ID10328 (< n >)*1

CONTENTS symbols of pattern ID10329 match those of
pre-existing pattern ID1493 (< 166 142 88 76 #60 s l o w l y >)*3

Context symbol 167 added to pattern ID10325 (< 167 y o >)*1

Context symbol 168 added to pattern ID10326 (< 168 u r >)*1

Unique ID symbol #141 added to pattern ID10326 (< 168 #141 u r >)*1

Context symbol 168 added to pattern ID10313 (< 168 164 162 160 #137 >)*1

Context symbol 169 added to pattern ID10327 (< 169 u >)*1

Context symbol 170 added to pattern ID10328 (< 170 n >)*1

Unique ID symbol #142 added to pattern ID10328 (< 170 #142 n >)*1

Context symbol 170 added to pattern ID2153 (< 170 165 141 87 85 75 #63 w a l k >)*5

Context symbol 171 added to pattern ID1493 (< 171 166 142 88 76 #60 s l o w l y >)*3

ID10331 (< #143 < 167 > < 168 > < 169 > < 170 > < 171 > >)*1

However, these two 'wrong' patterns should not matter because the program
is also forming 'correct' patterns like
ID117 (< 173 168 164 162 160 158 140 86 84 74 67 #14 y o u >)*6
and ID2181 (< #72 < 84 > < 85 > >)*12 and
ID8000 (< #123 < 140 > < 141 > < 142 > >)*8.

A possible reason why hierarchical structure is not being recognised is that
there are too many abstract versions of the long sentence pattern and the
short pattern. Since different context symbols are used in each case, the
```

```
program does not discover that they match. But this should only matter if
the abstract patterns are being matched against each other. In fact, the
program should recognise that the short sentence pattern is part of the
longer one by direct parsing. Why is this not working correctly?

If we trace through the best grammars from the beginning, things look good
when we get to ID3:

GRAMMAR GR218 (PATTERN ID3, PHASE 2)
derived from grammar GR56 and alignment:
ID335 (< #5 < 1 11 9 #14 y o u > < 23 12 10 2 #3 r u n > >)*1

G = 7022.37, E = 171.53, score = 7193.90.

Grammar patterns:

ID20 (< 23 10 2 #4 w a l k >)*1

ID22 (< 11 9 6 3 1 #6 w e >)*2

ID23 (< 23 12 10 2 #3 r u n >)*2

ID24 (< #5 < 1 > < 2 > >)*3

ID115 (< 1 11 9 #14 y o u >)*1

but then go bad when we get to ID4:

GRAMMAR GR659 (PATTERN ID4, PHASE 2)
derived from grammar GR389 and alignment:
ID747 (< 36 y o > < 38 u > < 39 35 33 27 25 10 2 #4 w a l k >)*1

G = 7199.18, E = 310.22, score = 7509.40.

Grammar patterns:

ID20 (< 39 35 33 27 25 10 2 #4 w a l k >)*2

ID22 (< 37 32 30 28 26 24 9 1 #6 w e >)*2

ID23 (< 39 35 33 25 10 2 #3 r u n >)*2

ID119 (< #15 < 9 > < 10 > >)*4

ID468 (< 36 y o >)*2

ID470 (< 38 u >)*2

The two fragmented patterns at the bottom originate like this:

Learning from alignment ID403:

0       y o     u w a l k 0
        | |     |
1 < #13 y o u r u n >     1

Part patterns from New:

CONTENTS symbols of pattern ID466 match those of
pre-existing pattern ID20 (< 35 33 27 25 10 2 #4 w a l k >)*1

Start of process_old_pattern().

Part patterns for Old pattern in row 1 (alignment ID403):

ID468 (< y o >)*1

ID469 (< u r >)*1

ID470 (< u >)*1

ID471 (< n >)*1

Context symbol 36 added to pattern ID468 (< 36 y o >)*1

Context symbol 37 added to pattern ID469 (< 37 u r >)*1

Unique ID symbol #35 added to pattern ID469 (< 37 #35 u r >)*1

Context symbol 37 added to pattern ID443 (< 37 32 30 28 #30 >)*1

Context symbol 38 added to pattern ID470 (< 38 u >)*1
```

```
Context symbol 39 added to pattern ID471 (< 39 n >)*1

Unique ID symbol #36 added to pattern ID471 (< 39 #36 n >)*1

Context symbol 39 added to pattern ID20 (< 39 35 33 27 25 10 2 #4 w a l k >)*1

ID472 (< #37 < 36 > < 37 > < 38 > < 39 > >)*1

It seems that the fragmentation wins because of all the excess ID-symbols in
the 'good' patterns. For example, this alignment:

COMPOSITE ALIGNMENT (PATTERN ID4 (ID4), CYCLE 1, PHASE 2)

ID747: NSC = 4221.77, EC = 47.44, CR = 88.98, CD = 4174.32,
Absolute P = 5.22392068302e-015

0      y o        u                          w a l k   0
       | |        |                          | | | |
1 < 36 y o >      |                          | | | |   1
                  |                          | | | |
2            < 38 u >                        | | | |   2
                                             | | | |
3                    < 39 35 33 27 25 10 2 #4 w a l k > 3

has an EC value of 47.44, whereas this 'correct' alignment:

SELECTED ALIGNMENT (PATTERN ID4 (ID4), CYCLE 2, PHASE 2)

ID764: NSC = 4221.77, EC = 72.89, CR = 57.92, CD = 4148.88,
Absolute P = 1.14459818548e-022

0                                    y o u                          w a l 0
                                     | | |                          | | |
1       < 37 34 32 30 28 26 24 1 9 #14 y o u >                      | | | 1
        |    |                             |                        | | |
2       |    |                             | < 39 35 33 27 25 10 2 #4 w a l 2
        |    |                             | |    |
3 < #34 <    34                            > <    35                       3


0 k     0
  |
1 |     1
  |
2 k >   2
    |
3   > > 3

has an EC value of 72.89.

%108 21/3/05

FURTHER NOTES ON SP71, V 8.1

At present, the best grammar found (with the short sentences first) is:

GRAMMAR GR13447 (PATTERN ID12, PHASE 2)
derived from grammar GR12907 and alignment:
ID12797 (< #72 < 173 168 164 162 160 158 140 86 84 74 67 #14 y o u >
< 170 165 141 87 85 75 #63 w a l k > >
< 177 171 166 142 88 76 #60 s l o w l y >)*1

G = 14498.43, E = 1554.68, score = 16053.11.

Grammar patterns:

ID22 (< 173 168 164 162 160 158 140 86 84 74 67 #6 w e >)*6

ID23 (< 170 165 141 87 85 75 68 #3 r u n >)*6

ID117 (< 173 168 164 162 160 158 140 86 84 74 67 #14 y o u >)*6

ID993 (< 177 171 166 142 88 76 f a s t >)*4

ID1492 (< 177 171 166 142 88 76 #60 s l o w l y >)*4

ID2152 (< 170 165 141 87 85 75 #63 w a l k >)*6

ID2180 (< #72 < 84 > < 85 > >)*12

ID10324 (< 167 y o >)*6

ID10326 (< 169 u >)*6
```

```
and the likely reason for the fragmented patterns ID10324 and ID10326 is that they
come from a composite pattern that has a relatively good (low) score because the
'correct' alignments are weighed down by lots of CONTEXT_SYMBOLs as in the
first 6 patterns in the above grammar.

A dirty fix for this problem is simply to exclude composite alignments from being used
during the compiling of alternative grammars. In this case, the best grammar is:

GRAMMAR GR10426 (PATTERN ID12, PHASE 2)
derived from grammar GR10216 and alignment:
ID15051 (< #120 < 192 188 185 182 177 172 168 166 164 162 144 77 48 32 11 #14 y o u >
< 174 169 145 78 49 12 33 #4 w a l k > < 181 175 170 146 79 #56 s l o w l y > >)*1

G = 13260.03, E = 2167.45, score = 15427.48.

Grammar patterns:

ID20 (< 174 169 145 78 49 12 33 #4 w a l k >)*6

ID22 (< 192 188 185 182 177 172 168 166 164 162 144 77 48 32 11 #6 w e >)*6

ID23 (< 174 169 145 78 49 33 12 #3 r u n >)*6

ID115 (< 192 188 185 182 177 172 168 166 164 162 144 77 48 32 11 #14 y o u >)*6

ID863 (< 181 175 170 146 79 f a s t >)*4

ID885 (< 59 #42 < 48 > < 49 > >)*12

ID1490 (< 181 175 170 146 79 #56 s l o w l y >)*4

ID9235 (< #120 < 144 > < 145 > < 146 > >)*8

and none of the the other grammars capture the hierarchical structure in the
sentences although that kind of structure appears to have been learned because
of the context symbol in ID885 -- and an examination of the output file shows
that hierarchical structures are indeed learned.

Question: why do the hierarchical structures not survive to the end of all
learning? The answer to this question is not clear at present. It may have
something to do with the program forming multiple alternative abstract
patterns for the two and three word sentences and this somehow drowns out
the hierarchical structures.

At the end of the first four (short) sentences, the best grammar is:

GRAMMAR GR482 (PATTERN ID4, PHASE 2)
derived from grammar GR290 and alignment:
ID746 (< #28 < 37 34 32 30 28 26 24 1 9 #14 y o u >
< 39 35 33 27 25 10 2 #4 w a l k > >)*1

G = 7241.86, E = 395.01, score = 7636.87.

Grammar patterns:

ID20 (< 39 35 33 27 25 10 2 #4 w a l k >)*2

ID22 (< 37 32 30 28 26 24 9 1 #6 w e >)*2

ID23 (< 39 35 33 25 10 2 #3 r u n >)*2

ID115 (< 37 34 32 30 28 26 24 1 9 #14 y o u >)*2

ID401 (< #28 < 24 > < 25 > >)*4

This is combined with the second-best grammar to make the Old patterns for
the next cycle and, after cleaning, these are:

OLD PATTERNS DERIVED FROM THE BEST 2 GRAMMARS AT THE END OF PATTERN ID4, PHASE 2:

ID20 (< 25 10 #4 w a l k >)*2

ID22 (< 24 9 #6 w e >)*2

ID23 (< 25 10 #3 r u n >)*2

ID115 (< 24 9 #14 y o u >)*2

ID401 (< #28 < 24 > < 25 > >)*4

ID119 (< #15 < 9 > < 10 > >)*4
```

```
ID13 (< #1 w e r u n >)*1

ID15 (< #2 w e w a l k >)*1

ID108 (< #13 y o u r u n >)*1

ID377 (< #27 y o u w a l k >)*1
```

This shows how two equivalent abstract patterns are formed that describe the same abstract structure. We need somehow to merge these equivalent structures.

With ID5, which is the first of the three-word sentences, the program forms hierarchical structure like this:

```
Learning from alignment ID824 (PATTERN ID5, PHASE 1):

0                w e             r u n   f a s t 0
                 | |             | | |
1       < 24 9 #6 w e >          | | |          1
        | |          |           | | |
2       | |          | < 25 10 #3 r u n >        2
        | |          | | |            |
3 < #28 < 24         > < 25           > >       3

Part patterns from New:

CONTENTS symbols of pattern ID836 match those of
pre-existing pattern ID833 (< 41 f a s t >)*1

Start of aggregate_part_patterns_and_alignments().

Context symbol 42 added to pattern ID401 (< 42 #28 < 24 > < 25 > >)*4

Context symbol 43 added to pattern ID833 (< 43 41 f a s t >)*1

ID837 (< #40 < 42 > < 43 > >)*1

but the best grammar for the first 5 sentences is:

GRAMMAR GR936 (PATTERN ID5, PHASE 2)
derived from grammar GR726 and alignment:
ID1136 (< #43 < 55 51 48 46 24 9 #6 w e > < 49 47 25 10 #3 r u n >
< 50 45 43 41 f a s t > >)*1

G = 9887.79, E = 344.95, score = 10232.74.

Grammar patterns:

ID20 (< 47 25 10 #4 w a l k >)*2

ID22 (< 55 51 48 46 24 9 #6 w e >)*3

ID23 (< 49 47 25 10 #3 r u n >)*3

ID115 (< 51 48 46 24 9 #14 y o u >)*2

ID833 (< 50 45 43 41 f a s t >)*1

ID846 (< #42 < 46 > < 47 > >)*5

ID849 (< #43 < 48 > < 49 > < 50 > >)*1

This is non-hierarchical but the second-best grammar is hierarchical:

GRAMMAR GR997 (PATTERN ID5, PHASE 2)
derived from grammar GR757 and alignment:
ID1195 (< #40 < 42 #28 < 55 51 48 46 24 9 #6 w e > < 49 47 25 10 #3 r u n > >
< 50 45 43 41 f a s t > >)*1

G = 9882.83, E = 378.71, score = 10261.54.

Grammar patterns:

ID20 (< 47 25 10 #4 w a l k >)*2

ID22 (< 55 51 48 46 24 9 #6 w e >)*3

ID23 (< 49 47 25 10 #3 r u n >)*3

ID115 (< 51 48 46 24 9 #14 y o u >)*2

ID401 (< 42 #28 < 24 > < 25 > >)*5
```

ID833 (< 50 45 43 41 f a s t >)*1

ID837 (< #40 < 42 > < 43 > >)*1

The program has been run with just the first 5 patterns and the results are
essentially the same as before. Here are the critical details for the first
two grammars where all patterns are the same except the abstract patterns.

GRAMMAR GR936 (PATTERN ID5, PHASE 2)
G = 9887.79, E = 344.95, score = 10232.74.

ID839 (< #42 < 46 > < 47 > >)*5

< (fr = 103, 2.19, LB, BM), #42 (fr = 5, 6.55, UID, ID),
< (fr = 103, 2.19, LB, CN), 46 (fr = 10, 5.55, CS, CN),
> (fr = 103, 2.19, RB, CN), < (fr = 103, 2.19, LB, CN),
47 (fr = 10, 5.55, CS, CN), > (fr = 103, 2.19, RB, CN),
> (fr = 103, 2.19, RB, BM), frequency = 5.

ID842 (< #43 < 48 > < 49 > < 50 > >)*1

< (fr = 103, 2.19, LB, BM), #43 (fr = 1, 8.87, UID, ID),
< (fr = 103, 2.19, LB, CN), 48 (fr = 6, 6.29, CS, CN),
> (fr = 103, 2.19, RB, CN), < (fr = 103, 2.19, LB, CN),
49 (fr = 4, 6.87, CS, CN), > (fr = 103, 2.19, RB, CN),
< (fr = 103, 2.19, LB, CN), 50 (fr = 2, 7.87, CS, CN),
> (fr = 103, 2.19, RB, CN), > (fr = 103, 2.19, RB, BM), frequency = 1.

GRAMMAR GR968 (PATTERN ID5, PHASE 2)
G = 9882.83, E = 378.71, score = 10261.54.

ID112 (< 44 #15 < 9 > < 10 > >)*5

< (fr = 103, 2.19, LB, BM), 44 (fr = 6, 6.29, CS, ID),
#15 (fr = 5, 6.55, UID, ID), < (fr = 103, 2.19, LB, CN),
9 (fr = 10, 5.55, CS, CN), > (fr = 103, 2.19, RB, CN),
< (fr = 103, 2.19, LB, CN), 10 (fr = 10, 5.55, CS, CN),
> (fr = 103, 2.19, RB, CN), > (fr = 103, 2.19, RB, BM), frequency = 5.

ID833 (< #41 < 44 > < 45 > >)*1

< (fr = 103, 2.19, LB, BM), #41 (fr = 1, 8.87, UID, ID),
< (fr = 103, 2.19, LB, CN), 44 (fr = 6, 6.29, CS, CN),
> (fr = 103, 2.19, RB, CN), < (fr = 103, 2.19, LB, CN),
45 (fr = 2, 7.87, CS, CN), > (fr = 103, 2.19, RB, CN),
> (fr = 103, 2.19, RB, BM), frequency = 1.

Here, G is slightly smaller for the hierarchical grammar than for the
non-hierarchical -- as expected.
The two values are G = 9887.79 and G = 9882.83.

But E is quite a bit bigger for the non-hierarchical grammar than for the
hierarchical. The two values are E = 344.95 for the non-hierarchical and
E = 378.71 for the hierarchical grammar. The probable reason for this is
that ID112 (< 44 #15 < 9 > < 10 > >)*5 has *two* ID-symbols compared with
the one ID-symbol for ID839 (< #42 < 46 > < 47 > >)*5. This penalty applies
every time the two-word sentence pattern is used.

%109 22/3/05

FURTHER NOTES ON SP71, V 8.1

At present, the model calculates frequencies and costs of symbols at the
beginning of processing by looking at the symbols in New. This is really
a mistake because, at the beginning, the model does not 'know' about the
patterns in New. It would make more sense to set the frequency of every
symbol to 1 at the beginning and let the model build up its knowledge of
symbol frequencies from their occurrencies in Old. The model will be
adapted to do this (now done).

Regarding the last conclusion in %108, that there are too many ID-symbols
in the pattern that represents intermediate structure, the model needs to
be modified so that, if a context symbol is added to a pattern which has
a unique ID symbol that is only required as a top-level reference, then
that ID-symbol may be removed (because the new context symbol serves
instead).

What seems to be needed is a function that is applied before Phase 2 that
scans the patterns in Old looking for ID-symbols that have no function
either as a top-level identifier or as an identifier that is used within
one or more other patterns or as discrimination symbols. These should be
removed. This function will be added to the program.

RESULTS WITH A PRELIMINARY VERSION OF find_and_remove_unnecessary_ID_symbols()

This preliminary version does not attempt to identify ID-symbols that are
needed as discrimination symbols.

Now, with N_GRAMMARS == 2, the best two grammars are non-hieachical:

GRAMMAR GR7935 (PATTERN ID12, PHASE 2)
G = 12205.32, E = 1233.21, score = 13438.53.

Grammar patterns:

ID20 (< 224 219 192 96 w a l k >)*6

ID22 (< 235 232 227 222 218 216 214 212 191 95 w e >)*6

ID23 (< 224 219 192 96 r u n >)*6

ID115 (< 235 232 227 222 218 216 214 212 191 95 y o u >)*6

ID820 (< 231 225 220 193 97 f a s t >)*4

ID1268 (< 231 225 220 193 97 s l o w l y >)*4

ID8091 (< #177 < 191 > < 192 > < 193 > >)*8

ID8116 (< 226 206 < 95 > < 96 > >)*12

and

GRAMMAR GR7906 (PATTERN ID12, PHASE 2)
G = 12205.25, E = 1233.77, score = 13439.02.

Grammar patterns:

ID20 (< 224 219 192 96 w a l k >)*6

ID22 (< 235 232 227 222 218 216 214 212 191 95 w e >)*6

ID23 (< 224 219 192 96 r u n >)*6

ID115 (< 235 232 227 222 218 216 214 212 191 95 y o u >)*6

ID820 (< 231 225 220 193 97 f a s t >)*4

ID1268 (< 231 225 220 193 97 s l o w l y >)*4

ID8116 (< 226 206 < 95 > < 96 > >)*12

ID10641 (< #201 < 218 > < 219 > < 220 > >)*8

but the third-best one is hierarchical:

GRAMMAR GR7965 (PATTERN ID12, PHASE 2)
G = 12239.78, E = 1240.22, score = 13480.00.

Grammar patterns:

ID20 (< 224 219 192 96 w a l k >)*6

ID22 (< 235 232 227 222 218 216 214 212 191 95 w e >)*6

ID23 (< 224 219 192 96 r u n >)*6

ID115 (< 235 232 227 222 218 216 214 212 191 95 y o u >)*6

ID820 (< 231 225 220 193 97 f a s t >)*4

ID1268 (< 231 225 220 193 97 s l o w l y >)*4

ID8091 (< #177 < 191 > < 192 > < 193 > >)*8

ID8116 (< 226 206 < 95 > < 96 > >)*12

ID8119 (< #187 < 206 > < 97 > >)*8

A possible reason for this is that ID8116 still has two ID symbols instead of
the 'correct' number of one. This is probably because the two are needed in
the complete set of Old patterns but they are not needed in any one grammar.
That both ID symbols are needed in the set of Old patterns is confirmed.

With N_GRAMMARS == 5, the best grammar is hierarchical and 'correct':

```
GRAMMAR GR8261 (PATTERN ID12, PHASE 2)
G = 12474.04, E = 661.18, score = 13135.23.

Grammar patterns:

ID20 (< 75 69 33 25 w a l k >)*6

ID22 (< 90 74 68 32 24 w e >)*6

ID23 (< 75 69 33 25 r u n >)*6

ID115 (< 90 74 68 32 24 y o u >)*6

ID805 (< 76 70 39 f a s t >)*4

ID1313 (< 76 70 39 s l o w l y >)*4

ID3099 (< 83 < 68 > < 69 > >)*12

ID3102 (< #84 < 83 > < 70 > >)*8

and so is the second-best grammar:

GRAMMAR GR8292 (PATTERN ID12, PHASE 2)
G = 12474.44, E = 661.99, score = 13136.44.

Grammar patterns:

ID20 (< 75 69 33 25 w a l k >)*6

ID22 (< 90 74 68 32 24 w e >)*6

ID23 (< 75 69 33 25 r u n >)*6

ID115 (< 90 74 68 32 24 y o u >)*6

ID412 (< 38 < 24 > < 25 > >)*12

ID805 (< 76 70 39 f a s t >)*4

ID809 (< #43 < 38 > < 39 > >)*8

ID1313 (< 76 70 39 s l o w l y >)*4

The third-best is non-hierarchical:

GRAMMAR GR8233 (PATTERN ID12, PHASE 2)
G = 12484.46, E = 656.98, score = 13141.44.

Grammar patterns:

ID20 (< 75 69 33 25 w a l k >)*6

ID22 (< 90 74 68 32 24 w e >)*6

ID23 (< 75 69 33 25 r u n >)*6

ID115 (< 90 74 68 32 24 y o u >)*6

ID805 (< 76 70 39 f a s t >)*4

ID1313 (< 76 70 39 s l o w l y >)*4

ID3086 (< #75 < 74 > < 75 > < 76 > >)*8

ID3099 (< 83 < 68 > < 69 > >)*12

After cleaning and tidying, the best grammar is:

GRAMMAR GR8321 (PATTERN ID12, PHASE 2)
G = 12368.93, E = 661.18, score = 13030.11.

Grammar patterns:

ID9764 (< 2 w a l k >)*6

ID9765 (< 1 w e >)*6

ID9766 (< 2 r u n >)*6

ID9767 (< 1 y o u >)*6
```

```
ID9768 (< 3 f a s t >)*4

ID9769 (< 3 s l o w l y >)*4

ID9770 (< 4 < 1 > < 2 > >)*12

ID9771 (< #1 < 4 > < 3 > >)*8

The second-best is:

GRAMMAR GR8322 (PATTERN ID12, PHASE 2)
G = 12369.33, E = 661.99, score = 13031.32.

Grammar patterns:

ID9772 (< 2 w a l k >)*6

ID9773 (< 1 w e >)*6

ID9774 (< 2 r u n >)*6

ID9775 (< 1 y o u >)*6

ID9776 (< 3 < 1 > < 2 > >)*12

ID9777 (< 4 f a s t >)*4

ID9778 (< #1 < 3 > < 4 > >)*8

ID9779 (< 4 s l o w l y >)*4

And the third-best is:

GRAMMAR GR8323 (PATTERN ID12, PHASE 2)
G = 12402.66, E = 656.98, score = 13059.64.

Grammar patterns:

ID9780 (< 4 2 w a l k >)*6

ID9781 (< 3 1 w e >)*6

ID9782 (< 4 2 r u n >)*6

ID9783 (< 3 1 y o u >)*6

ID9784 (< 5 f a s t >)*4

ID9785 (< 5 s l o w l y >)*4

ID9786 (< #1 < 3 > < 4 > < 5 > >)*8

ID9787 (< 6 < 1 > < 2 > >)*12

A repeat test shows that the program still obtains the 'correct' results when
the short sentences follow the long ones.

This version of the program will be frozen and a new version started for refinements.

%110 30/3/05

NOTES ON SP71, V 8.2 (BEFORE ANY CHANGES FROM V 8.1)

1 GENERALISATION

In order to test whether the program might overgeneralise with some kinds of
data, it was tested with this input:

[
    [
        (John loves Mary)
        (Mary loves John)
        (John is loved by Mary)
        (Mary is loved by John)
        (he loves her)
        (she loves him)
        (he is loved by her)
        (she is loved by him)
    ]
    [
    ]
]
```

It seemed possible that 'he', 'she', 'him' and 'her' might get lumped into
the same class as 'John' and 'Mary'. In that case, the grammar would
overgenelise and generate sentences like 'him loves she' etc.

The best grammar (after cleaning and tidying) is:

GRAMMAR GR6082 (PATTERN ID8, PHASE 2)
G = 7999.38, E = 2121.31, score = 10120.69.

Grammar patterns:

ID4579 (< 3 1 Mary >)*4

ID4580 (< 1 3 John >)*4

ID4581 (< 2 loves >)*4

ID4582 (< 2 is loved by >)*4

ID4583 (< #1 < 1 > < 2 > < 3 > >)*8

ID4584 (< 1 he >)*2

ID4585 (< 3 her >)*2

ID4586 (< 1 she >)*2

ID4587 (< 3 him >)*2

This grammar does not overgeneralise, mainly because it does not generalise
across the two contexts for 'John' and 'Mary' in the first place: it creates
separate context symbols for each context and does not recognise that
'John' and 'Mary' can appear in both contexts.

This needs more exploration.

2 INTERMEDIATE-LEVEL STRUCTURES

In order to test the program's ability to recognise other kinds of
intermediate-level structures, it has been run with this input:

```
[
    [
        (this boy loves that girl)
        (that boy loves this girl)
        (this girl loves that boy)
        (that girl loves this boy)
        (this boy hates that girl)
        (that boy hates this girl)
        (this girl hates that boy)
        (that girl hates this boy)
    ]
    [
    ]
]
```

The expectation is that the program should forma 'noun phrase' grouping
between the word level and the sentence level.

The best grammar (after cleaning and tidying) is this:

GRAMMAR GR1423 (PATTERN ID8, PHASE 2)
G = 5321.61, E = 383.32, score = 5704.93.

Grammar patterns:

ID5366 (< 8 5 2 11 hates >)*4

ID5367 (< 9 10 this boy >)*4

ID5368 (< 8 5 2 11 loves >)*4

ID5369 (< 7 12 that girl >)*4

ID5370 (< #3 < 10 > < 11 > < 12 > >)*2

ID5371 (< 6 1 that boy >)*4

ID5372 (< 4 3 this girl >)*4

ID5373 (< #4 < 1 > < 2 > < 3 > >)*2

ID5374 (< #1 < 4 > < 5 > < 6 > >)*2

ID5375 (< #2 < 7 > < 8 > < 9 > >)*2

Here, there is a complete failure to recognise the NP structure although the
program does pick out the noun phrases as discrete groupings.

What seems to be needed is a process of finding good partial matches and
learning new patterns applied to patterns like 'this boy' and 'this girl' so
that appropriate context classes and abstract sub-patterns can be formed.
This is a recursive application of the parsing and learning procedures.

This would clear up an anomaly in the current model where each new pattern has
to be compared with pre-existing patterns to see whether it already exists.
This is anomalous because it is an ad hoc matching and unification procedure
which is separate from the main procedures for matching and unification.

The program will be given version number 9.0 and re-organised something
like this:

```
SP71()
{
     1 Read a set of patterns into New. Each New pattern is normally a sentence
          expressed as a sequence of letters without spaces or punctuation
          between words. Old is initially empty.
     2 Make a preliminary measure of the frequencies of occurrence of the symbol
          types in the New patterns. From these frequencies, derive a bit cost
          for each symbol type using the Shannon-Fano-Elias method.
     3 While (there are unprocessed patterns in New)
     {
          PHASE 1: DERIVE PATTERNS FROM PARTIAL ALIGNMENTS.
               P1.1 Identify the first or next pattern from New as the 'current'
                    New pattern (CNP).
               P1.2 Parse the CNP by multiple alignment using the patterns that
                    are currently in Old.
               P1.3 If the best multiple alignment is partial or composite,
                    derive new patterns from it.
               P1.4 Apply P1.2, P1.3 and P1.4 recursively to the CONTENTS
                    symbols of each derived pattern.
               P1.5 Add the derived patterns to Old together with a copy of the
                    CNP to which 'identification' symbols have been added (ID-CNP).
          PHASE 2: COMPILE ALTERNATIVE GRAMMARS.
               [The same as v 8.1.]
     }
     4 Clean and tidy the best three grammars and print.
}
```

In this scheme, P1.2, P1.3 and P1.4 become an integrated function for parsing_and_learning().
To avoid anomalies, it must be totally self contained without any global variables that can
be changed from within the function. This includes the hit structure and the structures
for selecting amongst multiple alignments.

%111 31/3/05

NOTES ON SP71, V 9.0

On reflection, it seems that it is not necessary to make the hit structure and the other
structures used in recognition local to the recognition function - because they are all
cleared when the recognition function finishes and can then be used again in another
invocation of the function.

However, there are some difficulties in generalising the recognition functions for
other purposes:

* 'cycle' assumes cycles in the recognition of a New pattern. There could well be
some confusion if it were used for the recognition of other patterns.

* Adding ID symbols to the receptacle pattern is integrated with recognition.

* Updating of receptacle pattern is embedded in compare_patterns().

* print_pattern_cycle() assumes the processing of New patterns in two phases.

%112 22/4/05

NEW THINKING ON THE INTEGRATION OF PARSING AND LEARNING

The current problem can be summarised like this:

1 To avoid the creation of duplicate patterns during Phase 1, it is necessary to
match each new proposed pattern against existing patterns (in created_patterns
and old_patterns). If a match is found, the pre-existing pattern is used instead
of the proposed new pattern but a new context symbol is added to the pre-existing
pattern to reflect the new context in which the pattern may be found.

2 This matching of patterns seems to be ad hoc and poorly integrated with the
matching of patterns that is done during parsing. This suggests that, instead
of using a special-purpose matching function to check whether each proposed new
pattern already exists, it might be better to use the parsing process itself to
do this checking. This would reduce two matching procedures to one and thereby
simplify the program.

3 In order to create abstract structures representing the noun phrases in
sentences like "the boy loves the girl", it seems necessary to include the
learning process with the parsing of proposed new patterns. In other words,
the process of checking whether a proposed new pattern already exists would
be done by all the parsing and learning functions together, which is essentially
everything in Phase 1.

4 So Phase 1 would be made into a recursive function that could process a proposed
new pattern in such a way that it may itself propose new patterns which would be
processed by a recursive application of Phase 1, and this may propose new
patterns ... and so on.

5 Using a recursive function is not a problem in itself but it could prove very
complicated to manage the assignment of ID-symbols to patterns created at varying
depths of the recursion. There is a possible simplification by deferring the
application of the 'Phase 1' function to the stage when the initial application
of Phase 1 has finished. But the process is still intrinsically complex and difficult
to manage. It does look as if we need something that is simpler and 'cleaner' and
easier to manage.

Here is an alternative scheme that seems to have the advantage that it would
achieve an even fuller integration of parsing and learning:

1 Instead of doing learning when the parsing process has finished (ie it cannot
find any more Old patterns to add to the alignment, or an Old pattern is added
that does not have all its C-symbols matched), learning is applied after
combine_alignments() ***on every cycle*** of the parsing process.

2 At first sight, this is the same as what is done now, because parsing stops
as soon as a partially-matched Old pattern is added to an alignment and one might
think that no learning is possible unless there is a partially-matched Old pattern
in the alignment. The reasons that it is different from the current scheme are:

* In the proposed new scheme, learning of new patterns would happen on any cycle
where there were unmatched symbols in the New pattern. Given that learning in
each cycle would be done after combine_alignments(), there should be fewer occasions
when there were unmatched New symbols than in older versions of SP62 or in SP61
but it may still happen.

* If combine_alignments() produced a composite alignment, this would be a signal
for learning that would create an abstract pattern to tie the free-standing Old
patterns together.

* Proposed new patterns would be added to old_patterns immediately without
first putting them in created_patterns.

3 Instead of the basic operation being "parse with multiple cycles and then derive
new patterns from any 'good' partial alignments (with recursive application of the
parsing process)", the basic operation would be "compare two patterns and derive new
patterns from any 'good' partial matches (with recursive application of the
matching process)".

4 This is not a huge simplification but it should make the process easier to manage.
And it means that perception and learning are very closely integrated, as seems to be
the case in people (eg Broadbent's 1956 demonstration). It also means that there would
not be a process of parsing with a separate process of comparing proposed new patterns
with pre-existing patterns (as in current versions of SP71). There would be one single
process of comparing two patterns and deriving new patterns from partial matches -
although the use of combine_alignments() is a complication.

5 Although the basic operation is the comparison of two patterns, it should still be
possible to derive multiple alignments with multiple levels but those multiple alignments
would be the product of a combination of pattern matching and learning, not pattern
matching alone (as at present).

This new scheme will be attempted in SP71, v 9.0.

%113 25/4/05

FURTHER THOUGHTS ABOUT SP71, v 9.0

1 If learning is integrated with parsing, what will happen if the sentence can be
'fully' parsed by existing Old patterns? For a sentence like "the boy loves the
girl", the first cycle would produce a composite alignment something like this:

```
      the         boy         loves         the         girl
       |           |           |            |           |
< D 0 the >        |           |            |           |
                   |           |            |           |
            < N 0 boy >        |            |           |
                               |            |           |
                        < V 0 loves >       |           |
                                            |           |
                                     < D 0 the >        |
                                                        |
                                                 < N 1 girl >
```

Then, by learning, a new abstract pattern will be created something like this:

```
< A1 < D > < N > < V > < D > < N > >
```

Then this will be matched with pre-existing patterns and, we suppose, an exact match for the CONTENTS symbols will be found with a pattern something like this:

```
< A2 < D > < N > < V > < D > < N > >
```

So the program will 'realise' that there was no need for the A1 pattern and will use A2 instead and delete A1.

This seems clumsy and calls into question the idea of applying the learning process on every cycle of the recognition process.

Another question is why learning should produce a pattern like the A1 pattern above and not something like this:

```
< A1a < D 0 > < N 0 > < V 0 > < D 0 > < N 1 > >
```

And another question is whether the process of matching A1 with A2 would produce alignments like this:

```
< A1 < D > < N > < V > < D > < N > >
                    | | | | | |
                < A2 < D > < N > < V > < D > < N > >
```

from which the noun-phrase intermediate level of structure may be derived.

From the above, it looks as if there is a case for sticking to the current arrangement whereby parsing cycles are done until no more non-partial alignments can be found and use alignments with partial matches as the basis for learning.

2 Regarding the current process of looking for an exact match between proposed new patterns and pre-existing patterns, we may replace this with the process for finding good full or partial matches between two patterns.

*** There is no need to worry about taking this through more than one cycle, because when two patterns are matched, there are only two possible outcomes: there is either a full match or a partial match. If there is a full match, then the program need merely use the pre-existing pattern instead of the newly-created pattern. If there is a partial match, this can be used for further learning.

It is true that this process may be recursive if a partial match leads to the creation of proposed new patterns, but it is somewhat less complex than if we need to worry about multiple recognition cycles.

%114

%115

References

[1] J. G. Wolff. *Unifying Computing and Cognition: the SP Theory and Its Applications*. CognitionResearch.org, Menai Bridge, 2006. ISBNs: 0-9550726-0-3 (ebook edition), 0-9550726-1-1 (print edition). Distributors, including Amazon.com, are detailed on bit.ly/WmB1rs.