SIMPLICITY AND POWER

Some Unifying Ideas in Computing

Gerry Wolff

School of Electronic Engineering Science, University of Wales (Bangor), Dean Street, Bangor, Gwynedd, LL57 1UT, UK. +44 248 351151 ext 2691

Issue: 1.1

20 September 1988

This article is to appear in the Computer Journal

1. DOCUMENT CONTROL

1.1 Changes History

Issue 0.1 to 0.9 - earlier drafts each with limited circulation.

Issue 1.0 (26 April 1988) - issue 0.9, has been carefully revised in the light of comments from anonymous referees and from colleagues.

Issue 1.1 (20 September 1988) - the document has been reformatted using Framemaker, and small changes have been made to the text in the light of comments from an anonymous referee .

1.2 Changes forecast

None. This version of the document has been accepted for publication in the Computer Journal.

2. ABSTRACT

The article develops the conjecture that the organization and use of all kinds of formal system, knowledge structure or computing system may usefully be seen in terms of the management of **redundancy** in information.

Every formal system, knowledge structure or computing system has two key dimensions - **simplicity** (complexity or size) and expressive or descriptive **power** - and there is a **trade-off** between them. The balance between simplicity and power corresponds to a balance between **OR relations** and **AND relations** in the system.

The **efficiency** of the system (the ratio of power to size) depends on the extraction of redundancy from the system. Key mechanisms for the extraction of redundancy are **pattern matching** and **search** (by **hill climbing** or equivalent mechanism) for the greatest possible **unification** of patterns.

These principles are the basis of a proposed new language and associated computing machine, called SP, which combines simplicity with high expressive power. SP is a Prolog-like pattern-matching system well suited to high levels of parallelism in processing.

In SP, the boundary between ‘knowledge engineering’ and other kinds of information engineering breaks down. In SP there is the potential for full integration of artificial intelligence, software engineering and other aspects of computing - with Shannon-Weaver information theory as a unifying framework. SP also offers a bridge between ‘connectionist’ and ‘symbolic’ views of computing.

SP, and the principles on which it is based, have potential applications in several areas of computing including:

- Logic and logical inference.
- Formal specification of computing systems.
- Inductive learning.
- Pattern recognition.
- Object-oriented design and conceptual modelling: class-inclusion relations, part-whole relations, inheritance of attributes - and the integration of these constructs. SP supports intensional and extensional descriptions of classes and the creation of entity-relationship models.
- The representation of knowledge in expert systems and databases; information retrieval and content addressable memory.
- Probabilistic inference and reasoning with uncertain and incomplete knowledge.
- The representation of plans and the automatic generation of plans.

- Specification of the organization of natural languages including ‘context sensitive’ features.
- Software re-use and configuration management.
- The integration of diverse kinds of knowledge which is required in many applications including integrated project support environments; facilitation of translations between computer languages and between one form of knowledge and another.

3. INTRODUCTION

There is a Babel of languages, formalisms and representational systems in computing and a diversity of concepts. Some of this variety is useful but much of it is not. There is a need to rationalise and integrate computing ideas and to achieve a corresponding simplification of the subject.

Simplicity in itself is not enough. We could, for example, propose the very simple theory that all (digital) computing is about storing and manipulating bits. This theory is too simple to be illuminating. We need a view of computing which is simple but which also has explanatory and expressive power.

This article proposes some principles which can lay some claim to providing that desirable marriage of simplicity and power. The principles are embodied in a proposed new computing language and associated computing machine called ‘SP’. The name SP is mnemonic for ‘Simplicity’ and ‘Power’; it is also mnemonic for ‘Syntagmatic’ and ‘Paradigmatic’, two concepts from taxonomic linguistics which figure prominently in the system.

SP is a Prolog-like pattern-matching system well suited to high levels of parallelism in processing. The justification for creating yet another language is the expectation that it will lead to an overall simplification of the field.

In SP, the boundary between ‘knowledge engineering’ and other kinds of information engineering breaks down. In SP there is potential for the full integration of artificial intelligence, software engineering and other aspects of computing - with Shannon-Weaver information theory as a unifying framework. SP also offers a bridge between ‘connectionist’ and ‘symbolic’ views of computing.

The principles and the system have potential applications in several areas of computing including:

- Logic and logical inference.
- Formal specification of computing systems.
- Inductive learning.
- Pattern recognition.
- Principles of object-oriented design and conceptual modelling: class-inclusion relations, part-whole relations, inheritance of attributes - and the integration of these constructs. SP supports intensional and extensional descriptions of classes and the creation of entity-relationship models.
- The representation of knowledge in expert systems and databases; information retrieval and content addressable memory.
- Probabilistic inference and reasoning with uncertain and incomplete knowledge.
- The representation of plans and the automatic generation of plans.

- Specification of the organization of natural languages including ‘context sensitive’ features.
- Software re-use and configuration management.
- The integration of diverse kinds of knowledge which is required in many applications including integrated project support environments; facilitation of translations between computer languages and between one form of knowledge and another.

4. SIMPLICITY AND POWER IN GRAMMARS AND COMPUTING SYSTEMS

The ideas to be described derive most immediately from work on the inductive learning of grammars and from an analysis of cognitive development (see, for example, Wolff, 1982, 1987; an overview is presented in Wolff, 1988). I don’t intend to discuss these fields in detail - merely to sketch the ideas which are relevant here.

From this work on inductive learning has emerged the conjecture that knowledge representation and inductive learning may both usefully be seen in terms of the management of **redundancy** in information. The idea has now been generalized to the conjecture that: **the organization and use of all kinds of formal system, knowledge structure or computing system may usefully be seen in terms of the management of redundancy in information.**

Key ideas in the management of redundancy in a knowledge structure are:

- The detection of redundancy by **pattern matching** and
- The reduction of redundancy in the structure by **searching** for the greatest possible **unification** of patterns (using **hill climbing** or related process).

As I shall try to show, the extraction of redundancy by pattern matching and unification of patterns can provide a unified view of several concepts in computing. Since the concept of redundancy is part of Shannon-Weaver information theory, information theory is a foundation for the theory to be described.

The idea of searching for economical structures has figured in research on ‘neural computing’ (see, for example, Rumelhart & McClelland, 1986) and, indeed, in research on cluster analysis and numerical taxonomy. This article presents a view of computing which is not tied to any particular architecture (such as neural networks) and which is in several other respects different from work on neural computing or cluster analysis. In many ways it is complementary to these fields. In particular, it may be seen as a bridge between the purely connectionist and ‘sub-symbolic’ views of computing associated with research on simulated neural nets and the more traditional ‘symbolic’ views of computing.

4.1 Redundancy, Structure and Inductive Reasoning

Before we proceed to examine these ideas, there are some general points to be made about redundancy:

- **Redundancy means Repetition of Information.** Although this is not always obvious, redundancy in information (and thus all kinds of structure - see below) always means repetition of information. Repetition of information always means redundancy.

- **Redundancy and Structure.** The concept of redundancy (in the Shannon-Weaver sense) is closely related to the concept of **structure**. A body of information which has no redundancy is entirely random; it is ‘white noise’ with a maximum of entropy; it has no structure. The structure of a body of information may be equated with the patterns of redundancy in it. Thus any discussion of the structure or organization of a body of information may be mirrored by a corresponding discussion about the redundancy in the information.

- **Redundancy and Inductive Reasoning.** Whatever the particular reasons for storing and manipulating information (in computing systems, on paper, or in our heads) the underlying reason is always to make **predictions**. The principle of inductive reasoning - that the past is a guide to the future - is the basis of all kinds of natural or artificial cognition.

  Inductive reasoning depends on repetition of information: patterns of information which have repeated in the past are assumed to repeat in the future. Given that repetition of information also means redundancy, we can see that inductive reasoning (and thus all kinds of natural and artificial cognition) depends on the existence of redundancy in the world.

The idea that the storage and manipulation of information is always about prediction may seem obscure if we think of some hum-drum computing task like the storage and processing of accounts. But the accounts of a company (or any other organization) are only interesting in relation to the future: Should there be changes in how the company is managed? Should the receiver be called in? Should we buy more shares in the company or sell the ones we have got?

A world without redundancy would not permit prediction. It would lack any structure; there would be no point in storing or manipulating information because that information would provide no means of anticipating the future.

These ideas have a bearing on the philosophical problem of finding a rational basis for inductive reasoning but it would take us too far afield to discuss this interesting question here.

In the following subsections I discuss the significance of redundancy in more detail using grammatical inference (inductive learning) as a way of introducing the main concepts.

4.2 Grammatical Inference

Grammatical inference is a process of discovering, inducing or constructing a grammar from a body of ‘raw’ data - typically a string of characters or a set of strings of characters. The grammar which is inferred from the string or strings may be seen as a means of succinctly describing those strings.

There are always many grammars which are compatible with a given body of raw data, in the sense that they can ‘generate’ that body of data; some of these alternative grammars are, in some sense, ‘better’ than others. The inference problem is to find the ‘best’ grammar or, more realistically, to find one which is ‘good enough’.

Two key measures of ‘goodness’ are: the compactness, size or **simplicity** of a grammar; and its usefulness, expressiveness or **power** for describing data. The words with emphasis have been chosen deliberately to show the parallel with the need for both simplicity and power in how we think about computing.

Consider Figure 1. The ‘raw’ data at the top of the figure may be represented by a number of alternative grammars. In the first, labelled "Primitive Grammar 1", we simply give a label ‘1’ to the whole string of data. This primitive grammar is not at all compact: apart from the label, it is exactly the same size as the original data. But it is very ‘powerful’ in the sense that it may be used to represent the original data succinctly in other contexts by means of the symbol ‘1’.

Please insert Figure 1 about here

The same raw data may also be represented by "Primitive Grammar 2" in which each type of letter is represented by a digit. If rules are selected repeatedly from this primitive grammar it can ‘generate’ the original data. This grammar is very simple and compact - but it is not at all powerful: a description of the original data using the grammar is as big as the original data (see Figure 1).

These two primitive grammars represent extremes of a **trade-off** between simplicity and power. Between the two extremes lies a whole range of grammars covering a whole range of combinations of simplicity and power. The problem of grammatical inference is to find grammars, like the two ‘well-structured’ grammars in Figure 1, in which that combination is at or near a maximum.

The first well-structured grammar in Figure 1 is just as expressive (powerful) as the first primitive grammar (because the original data may be represented with the single symbol ‘1’) but it is more compact (simple) and thus represents an improvement over the first primitive grammar.

The second well-structured grammar in Figure 1 is as simple as the second primitive grammar but it is more powerful because the grammar may be used to encode the original data in a more economical form than with the second primitive grammar (as shown in Figure 1).

As a contrast to the two well-structured grammars, consider the two badly structured grammars in Figure 1. The first badly structured grammar is as powerful as the first well-structured grammar but it is less compact. We can say that it is relatively ‘inefficient’ compared with the first well-structured grammar because it has a relatively poor ratio of power to size.

The second badly structured grammar is as simple as the second well-structured grammar but it is less powerful. Again, its efficiency is low compared with the corresponding well-structured grammar. There are many possible inefficient grammars like this which should be rejected by the grammar induction process.

4.3 Grammars and Other Systems

There is a more than superficial analogy between these example grammars and other kinds of formal system. The analogy extends to cognitive and computing systems in general - hardware, software or both:

- Every system has a **size** measurable in terms of the amount of information (bits) needed to specify the system.

- Every system has more or less **power**, comparable with the expressive power of the example grammars. For example, a 'bare' computing machine, without software, is a relatively poor tool for doing any particular kind of task, eg processing accounting data and producing accounting reports. The addition of an operating system makes it more powerful in this sense and the further addition of an accounting package makes it even better. 'Power' in this context means much the same as 'usefulness' or 'functionality' and seems to be equivalent at some level of abstraction to the notion of expressive power introduced in last section.

In addition to the dimensions of simplicity and power seen in grammars and in computing systems, there is the concept of **efficiency** (the ratio of power to size) which we saw in connection with grammars and which seems also to apply to other systems. Efficiency in this discussion means 'doing a lot with relatively little'. It corresponds with the intuitive notion of 'elegance' or 'prettiness' of design.

A method of calculating *simplicity* (size), *power* and *efficiency* of grammars is described in the Appendix.

4.3.1 An Abstract Space for Grammars and Computing Systems

Figure 2 summarizes the ideas introduced so far. It is a set of graphs representing an abstract 'space' within which any grammar, formal system, knowledge structure or computing system may be placed. Each structure is a point in the space. The 'x' axis records the size of a structure; the 'y' axis records its descriptive power. These graphs represent imaginary data but are similar to unpublished graphs obtained from program SNPR - the program for grammatical inference which is described in Wolff (1982).

Please insert Figure 2 about here

Each of the curves in the figure represents the trade-off between size and power. Each of the higher curves represents a set of well-structured, *efficient* systems with good ratios of power to size. The lower curves are for poorly-structured, inefficient systems with low ratios of power to size.

Where a structure should be positioned up or down a curve depends on the relative importance of size and power and this will vary with the application. However, in all circumstances, the aim should be to maximize efficiency - the ratio of power to size. In grammatical inference, we need to find or create grammars which fall on or near the line marked '1' in the figure. In the design of computing systems we need to create systems which fall on or near that line.

No system can have more descriptive power than the raw data to which it relates. Hence the 'ceiling on power' in Figure 2.

4.4 An Analogy

It may be helpful at this point to describe an everyday analogy for the relationships shown in Figure 2. The analogy is fairly accurate but not completely so.

The relationship between size and power (in the senses intended here) is similar to the relationship between cost and value in the purchase of goods or services.

The trade-off between size (simplicity) and power is like the dimension from 'down market' to 'up market' which is so familiar when we are buying things. For a given type of commodity we may choose to go for lower cost and less value or we may choose to pay more for something which is higher quality or in some other way more valuable.

Of course, we all know that you do not always 'get what you pay for'. Independent of the up-market/down-market dimension is the notion of 'value for money'. Whether we are buying at the low or high ends of the market, a given sum of money can yield either poor value or good value. The notion of value for money is analogous to the notion of efficiency introduced earlier.

The cost and value analogy should not be stretched too far. Value, in particular, can be subjective and relative to a particular buyer. By contrast, the notions of size, power and efficiency can in principle be measured precisely in terms of information theory and without reference to subjective judgements.

4.5 Constraint and Freedom, Power and Simplicity, AND Relations and OR Relations

The trade-off between simplicity and expressive power in a grammar corresponds with a balance between freedom and constraint in the grammar. The second primitive grammar in Figure 1 provides the freedom to generate an infinite range of strings using the basic symbols. The first primitive grammar in Figure 1 is constrained to one string of symbols.

'Freedom' in this context corresponds to OR relations and 'constraint' corresponds to AND relations in the grammar. The trade-off between compactness and expressiveness is mirrored by the balance between OR relations and AND relations in a grammar.

These ideas also map onto basic information theory: 'freedom' means choice which means low information content; 'constraint' means reduced choice and raised information content.

Figure 3 illustrates the way the dimensions of ‘freedom-constraint’, ‘simplicity-complexity’ and degrees of ‘functionality’ are related. At the top of the diagram is a ‘raw’ computing machine representing relative simplicity, a high degree of freedom but low functionality. Below this are layers of organization representing decreasing freedom but increasing complexity and increasing functionality.

raw machine	freedom/simplicity/low functionality
microcode	
assembler	
high level language	
application program	constraint/complexity/high functionality

Figure 3

4.6 Efficiency and Redundancy Extraction

The efficiency of our two well-structured grammars was achieved by the **extraction of redundancy** in the original data and a corresponding compression of those data. Extraction of redundancy appears to be the key to achieving a favourable ratio of simplicity to power.

4.6.1 Avoid Repetition of Information

The basic idea in extracting redundancy from information is to avoid storing information more than once when once will do. Here are three ways in which this can be achieved:

- **Method 1.** In the two well-structured grammars in Figure 1, the patterns **A,B,C,D** and **P,Q,R**, which repeat themselves irregularly in the raw data, are stored once and accessed via pointers or references from inside the grammar in the case of the first well-structured grammar and from outside the grammar in the case of the second.

- **Method 2.** These two patterns:

  A,B,C,D,P,E,F,G,H
  A,B,C,D,Q,E,F,G,H

may be reduced to **A,B,C,D,x,E,F,G,H**, where **x --> P | Q**. Alternatively, the two patterns may be represented as **A,B,C,D(P | Q)E,F,G,H**.

- **Method 3.** Where a pattern or type of pattern repeats in a sequence of contiguous instances, this may be reduced to a single instance with some indication that it repeats. A sequence like **A,A,A,A,A,A,A,A,A,A** may be reduced to a recursive rule or it may be treated as ‘iteration’.

The first of these examples illustrates the principle of **structure sharing** as a means of reducing or eliminating redundancy in data: a repeating pattern of information is stored once and accessed via **references** from the several contexts in which it occurs. **A major motivation for using references**

in computing is to achieve the extraction of redundancy from data by means of structure sharing. A commonplace example is the use of named procedures and functions in computer programs, and calls to those procedures or functions from diverse contexts.

4.7 Pattern Matching and the Hill-Climbing Search for Efficient Structures

The three methods of extracting redundancy from information, described above, all depend on **pattern matching** and **unification** of patterns. Where a pattern of information repeats itself, the repetition can be avoided by creating one copy to replace all instances of the pattern: the replicated patterns are **unified**.

In most realistically large bodies of data there is a very large number of alternative ways in which patterns may be matched and unified. In these circumstances, the set of unifications to choose is the one which maximizes **C** where:

$$C = \mathrm{SUM}_{i=1}^{i=n} f_i . s_i$$

f is the frequency of pattern **i** in a body of raw data, **s** is its size (in bits or some equivalent measure) and **n** is the number of different kinds of pattern.

The patterns **A,B,C,D** and **P,Q,R** used in the two well-structured grammars in Figure 1 are better according to this measure than the patterns **Q,R,A,B** and **C,D,P** used in the two badly structured grammars.

There is no algorithmic way of ensuring that C is always maximized. Except for trivially simple cases, it is always necessary to use the kinds of **search** techniques familiar in artificial intelligence (most notably, **hill climbing**). These techniques cannot guarantee a perfect solution but they provide a practical means of finding at least an approximation to the desired goal.

An example of the application of hill-climbing to redundancy reduction by pattern matching and unification is described in detail in Wolff (1982). Winston (1984, Ch. 4) provides a useful introduction to the concept including the the idea of ‘local peaks’ and methods of avoiding them. In the context of computing with simulated neural networks, the technique of ‘simulated annealing’ has proved to be an effective means of reducing the chances of the system getting stuck on local peaks (see, for example, Hinton & Sejnowski, 1986).

4.8 De-referencing as Pattern Matching and Unification

As described above, when two or more patterns have been unified, references are often used to mark the contexts in which the original patterns occurred. A pattern may be brought back into its original context by **de-referencing** its identifier. **But de-referencing is itself an example of pattern matching and unification of patterns**. To achieve de-referencing, a match must be found for the reference elsewhere in the structure - and the two matching patterns must be unified.

A good example to illustrate this point is the way a named procedure in a computer program may be ‘called’ from several different contexts. The procedure name is associated with the body of the procedure in one part of the program. A reference to the procedure appears wherever the procedure is needed in other parts of the program. A ‘call’ to the procedure means searching for the name which matches the reference and unifying them.

Memory ‘pointers’ and memory ‘addresses’ may also be seen in these terms. Accessing an area of memory means searching for the address which matches a pointer and unifying them. The search process in this case is rather simple but it is search, nevertheless.

4.9 Modelling the World

As we have seen, there is a close connection between the concept of structure and the concept of efficiency. Efficient grammars are well-structured grammars and they are also the grammars which capture the redundancy (or structure) in the raw data.

When these ideas are generalized to other kinds of computing system, we find ourselves in familiar territory: a computing system is well structured to the extent that it reflects (‘captures’, ‘models’) the structure of its inputs and of its outputs. This principle has been propounded most persuasively by Michael Jackson (1975). It figures in the use of ‘entity-relationship models’ as a basis for the design of commercial software systems (LBMS, 1986). The principle is also an important part of the philosophy of object-oriented design (see, for example, Birtwistle et al., 1979; Cook, 1986).

The principle of modelling a computer system on the structure of the material it has to deal with is well recognized but the significance of this principle is, perhaps, not fully appreciated.

Normally it is justified on the grounds that it facilitates thinking about design and it makes it easier to modify designs. Both these justifications are sound but there is the additional, and related, reason that it is a means of maximizing the functionality of the system for a given cost in the complexity of the system (or minimizing complexity for a given functionality). Systems designed to conform to this principle are more efficient (in the sense defined above) than otherwise.

In general, systems which are ‘efficient’ are ones whose structure reflects or models the ‘natural’ structures in the external data. ‘Natural structures’ are patterns in the data which are ‘coherent’. Patterns are ‘coherent’ when they repeat frequently in the data. Natural structures in this sense include such things as ‘objects’, ‘entities’, ‘relationships’ and, as we shall see later, ‘classes’ of these things.

4.10 Useful and Useless Redundancies

The idea that ‘elegance’ or ‘prettiness’ in design are closely bound up with the extraction of redundancy in information is not in conflict with the well-known uses of redundancy in computing systems to increase speed of processing or the reliability of systems or both.

For example, it is normal practice in computing to create ‘backup’ or security copies of information as a protection against losing the information altogether. A backup copy is redundant in the sense that it replicates the original data. Here, the redundancy between the original and its copy is useful.

By contrast, redundancy within each copy arising from poor structuring of the information is not useful and should be minimized. ‘Management of redundancy’ means, to a large extent, the removal of **unnecessary** redundancy or complication in a structure.

The use of references or identifiers to achieve structure sharing is an interesting example of useful redundancy. To achieve this effect, the identifier must appear in association with what it identifies and it must also appear in each of the contexts where the material is referenced. In this case a small amount of redundancy is introduced into a knowledge structure as a means of achieving a relatively large reduction in redundancy: structure sharing only makes sense if the material to be shared is bigger than the identifier used to achieve the sharing.

4.11 Summary

Here is a summary of the ideas introduced so far:

1. Fundamental principles for the organization of formal systems and computing systems may be seen in a relatively transparent form in the organization of simple grammars.

2. Any grammar, other formal system, knowledge structure or computing system may be seen as a means of succinctly describing information which is external to the system - inputs, outputs or both.

3. Any system has a place in a space defined by two dimensions: **simplicity** (or size or complexity - meaning the information content of the system) and expressive or descriptive **power** (meaning the effectiveness with which the system can represent or manipulate its external information).

4. There is a **trade-off** between simplicity and power.

5. The balance between simplicity and power in a system corresponds to the balance between **OR relations** and **AND relations** in the system.

6. The **efficiency** of a system is the ratio of power to size.

7. Efficiency is obtained by capturing or extracting the **redundancy** in the external information.

8. Efficiency in this sense seems to correspond to the intuitive notions of ‘good structure’ in a system, ‘elegance’ or ‘prettiness’ in design. A well-structured system reflects or models the structure of its external data: objects or entities, relationships and classes.

9. The basic principle of **redundancy extraction** is to avoid recording a pattern more than once when once will do. Often this means the use of the principle of **structure sharing** and this means the introduction of **references** between one structure and another.

10. Redundancy extraction may be achieved more generally by **pattern matching** and the **search** (by **hill climbing** or equivalent process) for as much **unification** as possible. Notions of ‘reference’ and ‘de-referencing’ may be subsumed by these notions.

5. ORGANIZING PRINCIPLES

A main proposition from the last section is that any system may be placed anywhere in the space of possible systems by the application of two basic organizing principles:

- AND relations.
- OR relations.

Another main proposition from the last section is that *efficiency* (in the sense previously defined) may be achieved by the extraction of *redundancy* from information. Important mechanisms in this connection are *structure sharing*, *referencing* and *de-referencing*. More generally, redundancy extraction may be achieved by *pattern matching* and the *search* (by *hill climbing* or equivalent process) for the greatest possible *unification* of patterns.

These principles appear to have a fundamental significance in the organization and use of formal systems, knowledge structures and computing systems of all kinds.

Some of the importance of AND relations and OR relations is recognized in Michael Jackson's principle (1975) that any sequential file of data may be described in terms of sequence, selection and iteration; and likewise for any program using that file as data (see also Bohm & Jacopini, 1966). In general, however, the fundamental significance of the principles which have been described in previous sections seem not to be widely recognized.

To get a feel for the generality of the organizing principles we can briefly survey some of the ways in which they currently appear in computing.

5.1 AND Relations

Here are some examples of AND relations in computing:

- Sequences of statements in a program.
- The relationship between **a** and **b** in **If a then b else c** statements.
- The relationship between the fields in a table in a relational database.
- The sequence of structures in the body of a Prolog Horn clause.
- The relationship between the head and the body of a Prolog Horn clause.
- The relationship between an identifier and what it identifies in a wide variety of computing systems.
- The relationship between a structure in the syntax of any computing language and the semantics of that structure.
- The relationship between the (formal or actual) items in a parameter list for a function or procedure.
- The relationship between the fields of a Pascal record or COBOL record or C struct.

AND relations are often treated as being ordered; probable exceptions in the examples just given are fields in a relational table, fields in records or structs and the relationship between a syntactic structure and its semantics.

5.2 OR Relations

Examples of OR relations in computing:

- The relationship between **b** and **c** in **If a then b else c** statements.
- Case statements.
- The relationship between items in a Pascal enumerated type.
- The relationship between Horn clauses in a Prolog procedure.
- The relationship between the rows in a table in a relational database.
- The relationship between items in a C union and the relationship between the variants in a Pascal ‘record with variants’.

5.3 Identifiers and References

The types of identifier for an object (and thus the means by which it may be referenced) include:

1. The position of the object in (actual or virtual) memory.
2. The position of the object in a ‘naming space’. ‘Scoping rules’ may be used so that a given name may be used in more than one context.
3. A ‘path’ through a tree or network (eg Unix files or the ‘dot’ notation in Pascal). Path names may be abbreviated if the ‘scoping’ assumption is made that the current context is the default context.
4. The contents of the structure as in ‘content addressable memory’.

The use of identifiers in conjunction with AND relations and OR relations results in the trees and networks which are so widespread in computing: hierarchical directories of files, the structure of function or procedure calls in a typical computer program, plex structures in a network database, the structure of ‘include’ files in a typical C program or system, the structure of a non-overwriting filestore like ADAM (Peeling, Morison and Whiting, 1984).

The first three methods of identification may be seen as being ‘content addressable’ if the information used to identify an object is seen as being a **part** of that object.

The extreme case of identifying an object by its constituents is where the whole object stands for itself. This may seem an idiosyncratic, perhaps meaningless, view of the notion of identification but, if we do not allow this notion, we are forced into arbitrary distinctions which lead to unwelcome complications in our view of the subject.

As an illustration of the idea of an object standing for itself, each word in this text may be seen as an AND group of references to its constituent letters. Each letter is an object which stands for (is a reference to) itself. Creating special identifiers for each letter is less economical than using the letters directly.

5.4 Pattern Matching, Hill Climbing and Unification

Pattern matching is a recognized technique in computing (see, for example, Winston, 1977, Ch. 14) but the possibility that all kinds of computing may be seen as the extraction of redundancy by pattern matching and hill climbing search for the best possible unification seems not to be generally recognised: the section on pattern matching is omitted from the second edition of Winston's book (1984) and the topic does not even get an entry in the index.

From the following examples we can see some of the range of applications of pattern matching in existing systems and, to a lesser extent, search for good unifications:

- Resolution theorem proving: pattern matching and unification are basic mechanisms.
- De-referencing of identifiers may be seen as a form of pattern matching and unification.
- Parsing is clearly an application of pattern matching and unification. Many parsers are designed to deliver only one 'correct' parsing of any given input string. More sophisticated parsers will deliver alternative parses and distinguish 'good' ones from 'bad' ones.
- Information may be retrieved from databases by pattern matching (query-by-example).
- The creation of human-like capabilities in pattern recognition clearly requires pattern matching.
- Inductive learning is largely a matter of pattern matching allied to search mechanisms designed to seek the greatest possible unification of patterns (Wolff, 1982).

6. THE SP LANGUAGE

The syntax of the SP language, shown in Figure 4, is extremely simple. Since it represents a distillation of existing ideas in computing, there should be no surprise that the language is similar in certain respects to existing languages.

SP owes its greatest debt to Prolog but differs from Prolog in interesting ways:

- There is nothing in SP equivalent to the distinction in Prolog between the head and the body of a Horn clause. SP deals only in 'patterns'.
- The rules for matching and unification are significantly different from Prolog.
- Indeterminacy in the results of a computation is expressed in SP by creating a disjunctive structure rather than creating a set of alternative results not incorporated in any structure. This is significant in relation to inductive learning.

- In SP, variables do not have explicit names. A variable may be given a name by associating the variable with the name inside a structure, eg (Clementine,_). In this way, the connection between variables and their names is recognized as being an AND relation and is brought within the small range of basic constructs in the language without any *ad hoc* principle.
- The fact that variables are not explicitly named means that there is nothing in the language like the principle in Prolog that repeated instances of a named instantiated variable within one Horn clause refer to the same structure. This has a bearing on how ‘context sensitive’ structures are represented as will be described in section 7.4.

Although this will be less obvious, the language also owes a debt to object-oriented languages like Simula, Smalltalk and LOOPS. As I shall show in section 7.3, it supports the notions of classes and sub-classes, inheritance of attributes and part-whole relations. SP differs from all current OO languages in being significantly simpler and, in a theoretical sense, more ‘clean’.

The Lisp-like qualities of the language will be obvious.

6.1 The Syntax

```
Object --> Ordered-AND-object | Unordered-AND-object |
                                OR-object | Simple-object ;
Ordered-AND-object --> ‘(’ , body , ‘)’ ;
Unordered-AND-object --> ‘[’ , body , ‘]’ ;
OR-object --> ‘{’ , body , ‘}’ ;
body --> b | NULL ;
b --> Object , body ;
Simple-object --> symbol | ‘_’ ;
symbol --> character , s ;
s --> symbol | NULL ;
character --> ‘a’ | ... | ‘z’ | ‘0’ | ... | ‘9’ ;
```

Figure 4. The syntax for SP

Between any two simple objects which are immediately contiguous there must be at least one space, comma or new line. Otherwise, as an aid to readability, zero or more spaces, commas or new lines may be used in any desired combination between objects.

A ‘symbol’ may be a single character or a string of characters. In terms of the definition of SP, a string is an Ordered-AND-object of characters. The reason for having ‘symbol’ as a distinct construct (and not using the term ‘string’) is that symbols, unlike Ordered-AND-objects, will be treated as atomic.

To be practical, the SP language would need such constructs as ‘integer’ and ‘real’. The assumption here is that such concepts and the operations which relate to them (addition, multiplication etc) may be defined in SP and need not be supplied as primitives. Justification of this claim is beyond the scope of this paper.

6.2 The Semantics of SP: How it is Intended to Work

This section describes the possible ways in which the language may be made to work - ie to do some computing. A formal definition of the semantics has not yet been attempted.

The meanings of the types of object should be reasonably clear from their names:

- An Ordered-AND-object (OAO) represents a sequence: a collection of structures where the order of the structures is significant. An OAO represents what, in taxonomic linguistics, used to be called ‘syntagmatic’ relations.
- An Unordered-AND-object (UAO) represents a ‘bag’ of structures: the order of structures in the bag is not significant. Any given item may be repeated in a UAO but, for reasons which will become clear, any such repetition is likely to be removed as computing proceeds.
- An OR-object (ORO) represents a collection of items from which one or more may be selected. It represents an ‘inclusive’ OR relation amongst its constituents. An ORO represents what, in taxonomic linguistics, used to be called ‘paradigmatic’ relations.
- Simple-objects are the ‘atoms’ of the language.
- The ‘_’ character represents a ‘variable’, ie a place holder to which an object may be ‘assigned’. Like Prolog, values are assigned to variables by unification of patterns. Like Prolog and unlike most conventional programming languages, the value of a variable may not be changed once it has been assigned. In general, the language works in a ‘non-overwriting’ mode: non-redundant information is never destroyed in the course of computing.

The symbols ‘->’ and ’...’ which appear later in the article are not part of the SP language. ‘->’ is used to show when one SP object gives rise to another in the course of computing. ‘...’ is used in SP examples to show where other information would go in a more fully developed description.

6.2.1 Computing

To create ‘results’ using SP, the ‘data’ and the ‘program’ must be associated. This is done by joining them together within an AND object. Quote marks have been used for ‘results’, ‘data’ and ‘program’ because these distinctions are not recognized in the system and a uniform notation is used for them all.

All computing is done by **searching** within the composite program-data object to find patterns which **match** each other; when patterns do match they may be **unified**. The search space is usually very large; where there is recursion, the search space is infinite.

Pattern matching may be ‘free’ or ‘constrained’:

- SP in ‘free’ mode lives in an impractical world where all possible matches are sought and all possible unifications are delivered. This version of SP will not be discussed.

- SP in 'constrained' mode has restrictions on the kinds of match which may be sought or the kinds of unification which may be delivered.

There is a variety of ways in which constraint may be applied in the SP system and varying degrees of constraint - and there can be corresponding varieties of the SP language:

1. **'Broad' SP**. The most general kind of constraint relies on the notion of 'goodness' or 'economy' in matching and unification. The aim is always to achieve as much unification as possible or, equivalently, to reduce the size of an SP structure as much as possible.

   There is no algorithmic way to ensure that the maximum possible amount of unification is always achieved. Any practical version of 'broad' SP will have to use **hill-climbing** search techniques of the kind described in Wolff (1982). These may reliably find more or less 'good' unifications but cannot guarantee that the best will always be found. To cope with big search spaces in a practical way, Broad SP also demands high levels of parallel processing in pattern matching and unification.

   Like chess programs, versions of Broad SP may vary in how deeply the search tree is explored before decisions are made. Alternatively, the depth of exploration in Broad SP may be controlled by a parameter.

   Broad versions of SP may be regarded as the canonical forms of SP. All other versions may be regarded as approximations to the ideal.

2. **'Narrow' SP**. Superficially, the reliance of the most general constrained version of SP on search techniques seems to mean that computing with SP is different from the clockwork nature of conventional computing. However, on a 'simplicity and power' view, **all** computing systems rely on search techniques of some kind even when this is not explicitly recognized. For example, any practical parser on a conventional computer embodies a 'biggest is best' principle to help it find the 'correct' parsing from amongst the many 'incorrect' parses which the grammar may otherwise allow. The clockwork nature of Prolog derives from constraints applied to how pattern matching may be done and what will count as 'true'.

   SP may be constrained to work more like Prolog or a conventional programming language (eg Pascal, C or functional languages like Lisp) by being more restrictive about how the search for matching patterns is done. Here are three of the more obvious kinds of restriction:

   1. The search for matches for the constituents of an object may be done always in **left-to-right sequence**. This reduces the range of possible matches which can be found and gives the language a 'procedural' flavour. If this restriction is applied only to OAOs then these objects become 'processes' and UAOs represent 'concurrency' amongst their constituents.

   2. **Nearest is dearest**. When the system seeks a match for any given object it will try other objects in order of their 'distance' from the given object in the total structure of objects rather in the manner of scoping rules in languages like Pascal.

3. **All-or-nothing matching**. In most computer languages, the search for matching patterns is greatly reduced by insistence on all-or-nothing matching, particularly for identifiers. This kind of restriction is implied by the atomic nature of the ‘symbol’ construct of SP. This feature should, perhaps, be removed in any ‘broad’ version of SP.

These and other kinds of restriction on pattern matching may be regarded as ways of reducing the size of the space of possible unifications which the system has to search and thus making the search tractable within the limits on processing power imposed by currently available hardware. The penalty of these restrictions is that they limit the range of problems which the computing system may solve.

6.2.2 Rules for Matching and Unification

Here are the rules required for pattern matching and unification in SP:

1. **Matching simple objects**. Simple objects which are not variables and which match (eg A and A) give a copy of either object as the unification.

2. **Matching with variables**. An SP variable matches any object. In all cases, including when a variable is matched with another variable, the result of unifying a variable with another object (or objects) is a copy of the other object(s). For example, [A,_] matches [A,B] giving [A,B]; [A,_] matches with [A,B,C] giving [A,B,C].

3. **Matching two OAOs**. Two OAOs may match completely or partly. In both cases, unification is possible:

    1. When one or more constituents of a OAO match one or more constituents of another OAO, **in the same order**, then each constituent of the one OAO may be unified with the corresponding constituent of the other. Obviously order is irrelevant when only one constituent from each OAO is involved.

        Notice that ordinal position is not significant. For example, A and B in (X,A,B) may be unified with A and B in (A,Y,B).

    2. When two OAOs match only partly, the resulting unification contains all the unmatched constituents from both OAOs - in the appropriate position relative to the constituents which have been unified. These unmatched constituents are formed into one or more OROs, each in an appropriate position in the unified pattern. For example, the unification of (X,Y,A,B) and (P,Q,R,A,B) will give ({(X,Y)(P,Q,R)}A,B).

    3. Where alternative partial unifications are possible then the one giving the greatest amount of unification is preferred (unless there is a tie in which case an arbitrary choice is made). For example, if unification were sought between (A,B,C,P,D,E,F) and (A,B,C,Q,D,E,F) then (A,B,C{P,Q}D,E,F) would be better than ({(A,B,C,P)(A,B,C,Q)}D,E,F).

It is normal for there to be many alternative possible unifications. A more realistic example appears in section 7.1. The 'best' set of unifications is the one which reduces the size of the structure by the greatest amount. As we have seen, there is no algorithmic method which is guaranteed to find the best set of unifications in all cases. An SP 'engine' will depend on hill climbing search techniques or something equivalent.

4. **Matching with UAOs**. The rules for matching a UAO against another UAO or an OAO are the same as for matching two OAOs except that ordering constraints do not apply.

5. **Matching with OROs**. Two rules apply:

    1. An ORO matches any other object if one of its constituents matches that object. The resulting unification is a copy of the given object. For example, {A,B,C} matched with A gives A.

    2. An ORO matches another ORO if one or more of its constituents can be matched with one or more constituents of the other ORO. The unification is an ORO whose constituents are the result of unifying the constituents which match, unless there is only one such unified pair of constituents in which case the result of unifying the two OROs is that unification. For example, the unification of {A,B,C} and {B,C,D} is {B,C}. The unification of {A,B,C} and {C,D,E} is C.

       In general, the effect of unifying two OROs is to form the intersection of the two sets of constituents.

The rules described in this section are not fully developed - more work is needed in this area. There is a case, in future work, for introducing 'weights' on SP objects to reflect (absolute and contextual) frequencies of objects in the raw data. The metric which evaluates alternative unifications will need to accommodate these weights.

The introduction of weights is likely to lead to some modification in the rules just described, particularly the rules for matching and unification with OROs. Two OROs may be unified by taking the union of their constituents (rather than the intersection) but assigning higher weights to items which are in the intersection.

7. APPLICATIONS

In this section I shall illustrate the workings of SP and the principles on which it is based using examples from several areas of computing.

7.1 Parsing with a Simple Grammar

This first example, showing how SP may be used to represent a simple grammar and parse a simple 'sentence', may at first sight seem remote from the mainstream of computing. The justification for first illustrating the workings of SP in this way is the belief that many aspects of computing may be seen in these terms.

In many ways SP has the organization of simple phrase structure grammars. In section 7.4 I will show how SP overcomes the limitations of such grammars and has the power to handle ‘context sensitive’ features in representations of knowledge.

```
[
(S(NP,_)(VP,_))
(NP{john,mary})
(VP(V,_)(NP,_))
(V{loves,hates})
]
```

Figure 5. A grammar written in SP

Figure 5 shows a little grammar written in SP. In this and later examples of grammars, the distinction between upper and lower case letters has no formal significance; it merely helps one to see which objects are serving as ‘non-terminal symbols’ and which represent ‘text’. To forestall misunderstanding, it should be stressed that SP is **not** a re-write system. The matching and unification mechanisms in SP may imitate the effect of a re-write system but they are more general.

The example grammar may be used to parse a sentence by associating the sentence with the grammar inside a UAO as shown in Figure 6.

```
[(john,loves,mary)
[
(S(NP,_)(VP,_))
(NP{john,mary})
(VP(V,_)(NP,_))
(V{loves,hates})
] ]
```

Figure 6. Association of a sentence with a grammar

Consider, first of all, a ‘top down’ parsing, where the processes of matching and unification are driven by the ‘top level’ rule in the grammar: (S(NP,_)(VP,_)). Taking the components of this object in turn, left to right, the system tries to unify as much as possible of this object with other objects in the ‘universe’ of objects in the combined sentence and grammar. The same applies to any new objects created by unification. The sequence of matchings and unifications is shown here:

```
1  (S(NP,_)(VP,_))

2  (S(NP{john,mary})(VP,_))
   /* By unification of (NP,_) with (NP{john,mary}) */

3  (S(NP,john){(VP,_)(loves,mary)})
   /* By unification of {john,mary} with ‘john’ in (john,loves,mary) */

4  (S(NP,john){(VP(V,_)(NP,_))(loves,mary)})
   /* By unification of (VP,_) with (VP(V,_)(NP,_)) */

5  (S(NP,john){(VP(V{loves,hates})(NP,_))(loves,mary)})
   /* By unification of (V,_) with (V{loves,hates}) */

6  (S(NP,john)(VP(V,loves){(NP,_)(mary)}))
   /* By unification of {loves,hates} with ‘loves’ in (loves,mary) */

7  (S(NP,john)(VP(V,loves){(NP{john,mary})(mary)}))
   /* By unification of (NP,_) with (NP{john,mary}) */

8  (S(NP,john)(VP(V,loves)(NP,mary)))
   /* By unification of {john,mary} with ‘mary’ */
```

Figure 7. Unification in a ‘top down’ parsing of the sentence in Figure 3 using the grammar in Figure 3

Some comments on Figure 7:

- In stage 2 we see the use of the variable ‘_’.
- In stage 3 we see how an ORO, {john,mary}, is reduced to a singleton, ‘john’, when one of its constituents matches that singleton.
- Also in stage 3 we see how SP forms a disjunction when two matched patterns both contain unmatched components. After ‘john’ within (john,loves,mary) has been unified with {john,mary} within (S(NP{john,mary})(VP,_)), the residues of the two larger objects are formed into an ORO: {(VP,_)(loves,mary)}.

Figure 8 shows how the system may work in a 'bottom up' mode, where unification starts with the sentence rather than the top rule of the grammar.

1 (john,loves,mary)

2 ((NP,john),loves,mary)
/* By unification of 'john' in (john,loves,mary) with (NP{john,mary}) */

3 (S(NP,john){(VP,_)(loves,mary)})
/* By unification of (NP,john) with (S(NP,_)(VP,_)). There is a conflict here because (NP,john) may also be unified with (VP(V,_)(NP,_)) giving ((VP(V,_)(NP,john))(loves,mary)). In this example, this second 'bad', alternative will not be followed through. This kind of conflict is discussed in the text. */

4 (S(NP,john){(VP,_)((V,loves),mary)})
/* By unification of (V{loves,hates}) with 'loves' in (loves,mary) */

5 (S(NP,john){(VP,_)(VP(V,loves){(NP,_),mary})})
/* By unification of ((V,loves),mary) with (VP(V,_)(NP,_)) */

6 (S(NP,john)(VP(V,loves){(NP,_),mary}))
/* By unification of (VP,_) with (VP(V,loves){(NP,_),mary}) */

7 (S(NP,john)(VP(V,loves){(NP,_),(NP,mary)}))
/* By unification of 'mary' with (NP{john,mary}) */

8 (S(NP,john)(VP(V,loves)(NP,mary)))
/* By unification of (NP,_) with (NP,mary) */

Figure 8. Unification in a 'bottom up' parsing of the sentence in Figure 6 using the grammar in Figure 6

In Figure 8 we see an example, at stage 3, of how rival unifications can arise. They represent alternative solutions to the problem in hand. In principle, the system may be allowed to compute all possible alternative solutions. For most purposes, it will be necessary to nip off any branch of the search tree if it seems to be leading in an unhelpful direction. In terms of the theory on which SP is based, 'unhelpful' means uneconomical. If the 'bad' path is followed at stage 3 of Figure 8 it will lead to this parsing:

((VP(V{loves,hates})(NP,john)),loves,(VP(V{loves,hates})(NP,mary)))

No further unification is possible without violating the order constraints on the sentence. This solution is clearly much less economical than the 'correct' parsing and may be rejected on those grounds.

7.2 Logic and Logical Inference

SP has potential as a medium for expressing logical propositions and making logical inferences. This is perhaps not surprising, given the significance in the language of the simple logical relations, AND and inclusive OR.

Here is a familiar and elementary example:

All men are mortal.
Socrates is a man.
Therefore Socrates is mortal.

In first-order predicate calculus this may be expressed as:

```
FORALL x is-member-of man . mortal(x)
AND Socrates is-member-of man
=> mortal(Socrates)
```

In SP, the same premises and the inference may be expressed like this:

```
[[mortal[man,_]]
[man,Socrates]]
-> [mortal[man,Socrates]]
```

Remember that '->' is not part of SP. It simply shows what is produced in the course of computing. In the first line, the pattern [man,_] matches any UAO which has 'man' as a constituent object. Thus it may be read as "all men" or "any man". The underscore symbol removes the need for a universal quantifier rather in the way that Skolemization may be used to remove universal quantifiers in predicate calculus. (Existential quantifiers are not needed either because any given object may be named.)

The conjunction of 'mortal' with [man,_] gives the pattern [mortal[man,_]]. This represents "All men are mortal". Likewise, [man,Socrates] represents "Socrates is a man". When these two patterns are matched and unified, the result is [mortal[man,Socrates]] which may be read as "Socrates is a man and he is mortal".

Thus SP does not derive [mortal,Socrates] directly from the two original propositions but creates a pattern with which [mortal,Socrates] will unify. In this way, the proposition "Socrates is mortal" is validated.

The problems mentioned earlier in finding matches between patterns and in differentiating between 'good' and 'bad' matches, suggest that SP is too uncertain a medium on which to base logical inference. But it is already known that every formal system which is expressive enough to be useful is also 'incomplete' in the sense that there are truths which are expressible in the system but which cannot be proved within the system. The uncertainty of matching and unification in infinite search spaces may provide a reason for the existence of incompleteness in formal systems.

7.2.1 ‘True’ and ‘False’

SP has no explicit concepts of ‘true’ or ‘false’. These concepts, which are primitives in other views of computing may be seen as emergent properties of the SP system.

If every SP object is regarded as some kind of statement (it is, without doubt, a body of information), then it may be regarded as true if it matches some other object, much in the way that a Prolog Horn clause is regarded as true if a match can be found for it within a set of clauses.

SP differs from Prolog, of course, in that it allows partial matching. Correspondingly, it supports the notion of degrees of truth. In general, truth in an SP system will be modelled by the metrics which guide the search for ‘good’ structure within the system. There is no space here to discuss fully the ramifications of these ideas.

7.3 Object-Oriented Design and Entity-Relationship Models

This section discusses a set of inter-related ideas associated with object-oriented (OO) design and with the creation of conceptual models.

The distinctive features of OO systems (eg Simula, Smalltalk and LOOPS) are:

- That all data structures and procedural code are organized as **objects**. An object is an association of data structures with the procedural code (‘functions’ or ‘methods’) with which those data structures may be manipulated. In most systems, data structures are accessed **only** by means of ‘messages’ sent to object methods and may thus be protected from modification in illegal ways.

- Objects in an OO system may be grouped into **classes** (including super-classes and sub-classes) in the same way that biologists group animals and plants. This has the psychological advantage that groupings may be set up corresponding to the way we naturally think of these things.

- A (related) advantage is that any structure (‘property’ or ‘attribute’) in an object may be recorded just once at the appropriate level of generality and it may then be **inherited** by the lower levels. This saves space; by allowing redundancy to be minimized, it helps avoid problems of inconsistency in design and facilitates the modification of designs.

In the more sophisticated OO systems (eg LOOPS), cross classification is possible with the possibility of a class having multiple inheritance of attributes from two or more higher level classes. A structure of interlocking hierarchies like this is sometimes called a ‘heterarchy’.

In OO languages, classes serve as models for the creation of object ‘instances’. This is similar to the way in which types serve as models for the assignment of values to variables in other languages.

In most OO systems, classes are themselves objects which are instances of ‘metaclasses’.

An idea which is complementary to the notion of class inclusion relations is the notion of **part-whole relations**. This idea tends not to be explicitly recognized in OO systems but is there nonetheless in the kinds of groupings and sub-groupings of objects which can be formed. Part-whole relations form hierarchies and heterarchies as AND trees and AND networks.

Related to OO systems are the kinds of ‘entity-relationship’ (ER) models which are the stock-in-trade of systems analysis (LBMS, 1986). Such models typically identify significant ‘entities’ in the domain being modelled, ‘attributes’ of those entities and named ‘relationships’ between entities.

Although they do not usually figure in discussions of OO systems and ER models, three other concepts will be mentioned here which relate to classes and categories. A class may be defined **extensionally** by listing all the members of the class. It may also be defined **intensionally** by describing the properties of the class. The third notion is that many of the classes we regularly use in everyday life are **polythetic**. What this means is that there need be no single attribute which is found in every example of the class. This property of natural categories has proved puzzling to theorists who try to characterize classes in terms of **defining** characteristics. Polythetic classes need not have any defining characteristics.

All the concepts described in the foregoing may be accommodated by the concept of an ‘object’ in SP: ‘class’, ‘super-class’ and ‘sub-class’, ‘metaclass’, ‘instance’, ‘function’, ‘method’, ‘message’, ‘class-hierarchy’ and ‘heterarchy’, ‘part-whole hierarchy’ and ‘heterarchy’, ‘entity’, ‘attribute’, ‘relationship’, ‘extensional definition’, ‘intensional definition’ and ‘polythetic class’.

7.3.1 Class-inclusion Relations, Part-whole Relations and Inheritance of Attributes

Figure 9 shows how class-inclusion relations, part-whole relations and inheritance of attributes may be integrated.

```
[person [name,_]
        ((head ((eyes ...)(nose ...) ...))
         (body ...)(legs ...))
        [eats ...][sleeps ...][breaths ...] ...
         [profession
               {[tinker ...]
                 [tailor ...]
                   ...
               }]
         [gender
                 {[male ...]
                   [female ...]
                  }]]
```

Figure 9. The integration of class-inclusion relations, part-whole relations and inheritance of attributes

The sets of three dots (‘...’) show where other information would go in a more fully developed description. This structure shows the class ‘person’ as having the attributes ‘head’, ‘body’, ‘legs’, ‘eats’, ‘sleeps’ etc. These are the **parts** of which the concept of a person is composed; parts may have sub-parts down to any level.

‘Persons’ have sub-classes ‘tinker’, ‘tailor’ etc and they are also cross-classified as ‘male’ or ‘female’. Any number of levels is possible in this kind of classification scheme.

Notice how part-whole relations equate with AND relations, while class-inclusion relations equate with OR relations.

An ‘instance’ of a person may be represented as:

[person [name,Tom],_,[profession[tinker,_]][gender[male,_]]]

or even more succinctly as:

[_,[_,Tom],_,[_,[tinker,_]][_,[male,_]]]

Either pattern will unify with the class schema giving a full description of ‘Tom’ something like this:

```
[person [name,Tom]
        (head ((eyes ...)(nose ...) ...))
         (body ...)(legs ...)
          (eats ...)(sleeps ...)(breaths ...) ...
          [profession [tinker ...]]
          [gender [male ...]]]
```

Pattern matching and unification provides a mechanism by which attributes may be **inherited** by an instance (or sub-class) in the sense understood in object-oriented design. In SP, unlike most other OO systems, the concepts of class-inclusion relation, and inheritance of attributes are **integrated** with the concept of part-whole structure.

7.3.2 Methods and Messages

Narrow SP has potential as a ‘procedural’ language although the details will not be pursued here. An SP object may contain constituent objects corresponding to ‘methods’ in other OO languages. For example, a class ‘person’ containing the method ‘eat’ may be represented something like this:

[person, [name,_],[eat[food,_], ...], ...]

The first set of three dots corresponds to the details of how eating is done. The second set represents the many other attributes of the class ‘person’. A particular person may be represented as, for example:

[person, [name,Mary],_]

or, more succinctly, [person,Mary,_].

If Mary is to eat something then one can send a ‘message’ to the object which represents her something like this:

[person,Mary,[eat,ham-sandwich],_]

Thus, in SP, both ‘methods’ and ‘messages’ may be treated as objects, not essentially different from other kinds of knowledge. One advantage of treating ‘methods’ and ‘messages’ uniformly with other kinds of knowledge is that ‘inheritance’ may be exploited within them. A ‘method’ may be inherited by other objects and it may itself inherit objects from elsewhere. Likewise for ‘messages’. Knowledge engineering systems like KEE go some way down this path.

7.3.3 Class, Meta-class, Instance and the Evolution of Classes

Most OO systems make a sharp distinction between ‘classes’ and ‘instances’. Classes serve as templates for the creation of instances but instances may not be templates for anything. The structure of classes is established by a designer before the system runs and remains fixed while the system runs. As it runs it may create new instances of classes dynamically but not new classes. Since classes, in most systems, are regarded as objects, this means that they must be instances of something. To avoid classes being instances of classes (which would violate the sharp distinction between instances and classes) it is necessary to introduce the concept of ‘meta-class’. Classes may then be instances of metaclasses. There is an infinite regress because meta-classes must be instances of meta-meta classes - and so on (see Ungar & Smith, 1987).

In SP, the concepts of ‘instance’, ‘class’ and ‘meta-class’ are merged. There is no attempt to create ‘strict’ hierarchies or avoid Russell’s paradox. The advantage of this breaking down of the distinction between classes and instances is the removal of unnecessary rigidities in the system: any object may serve as a template for the creation of other objects (ie any object may serve as a class) and any object may be created dynamically as the system runs. In other words, the structure of classes of the system may evolve as the system runs.

Many people find it difficult to abandon the distinction between classes and instances. The reason seems to be that the things which, in everyday life, are conventionally regarded as instances of classes (eg individual people) are highly salient, coherent concepts. The fact that an individual person (dog, cat, table, chair) is a very highly coherent concept should not disguise the fact that such things are in fact collections of more primitive percepts. In short, they are classes of visual, auditory and tactile images. Something which would conventionally be regarded as an instance, eg ‘Mary’, may be specialised into such concepts as ‘Mary-as-wife-and-mother’, ‘Mary-as-magistrate’ etc. These are sub-classes of the class ‘Mary’.

7.3.4 Entity-Relationship Models

Representing the relationship between two or more entities in SP is no different from representing the structure of a given entity. The relationship may be expressed as an AND relation between the relationship identifier and identifiers of the entities to be related. For example,

```
[employs,John,[Harry,Mary],_]
```

in the context of this schema:

```
[employs
        [employer,_]
        [employees,_]
          ...]
```

expresses the idea that ‘John’ employs ‘Harry’ and ‘Mary’. The ellipsis represents other information (eg company law) that may be associated with the relationship between employers and employees.

7.3.5 Polythetic Classes

As already mentioned, most ‘natural’ categories are polythetic, meaning that there need not be any single attribute which is found in all examples of the class. We can recognize something as, say, a cat when any one (or perhaps more) of its distinctive features is missing or replaced by something else. This property of natural categories is puzzling if one assumes that there must be defining characteristics for classes, but it can be accommodated quite easily by SP.

As a simple example, the SP object ({A,B}{C,D}{E,F}) represents the class of three letter strings comprising ACE, ACF, ADE, ADF, BCE, BCF, BDE and BDF. No single attribute (letter) is found in every example of the class. In general, polythesis is a reflection of disjunction (OR relations) in the organization of knowledge.

7.4 ‘Context Sensitive’ Power in SP

The expressive power of systems for constructing context-free grammars is limited. Systems of this type (eg BNF) cannot, for example, adequately represent the syntax and semantics of most natural languages.

Although SP resembles BNF, it can succinctly represent structures in natural language - such as ‘discontinuous dependencies’ in syntax - which are beyond the scope of BNF. SP has at least the expressive power of ‘context-sensitive’ systems like Definite Clause Grammars (DCGs; Pereira & Warren, 1980).

In the French sentence *Les plumes sont vertes* there are two overlapping sets of discontinuous dependencies. They are shown here by underscores:

> *<u>Les</u> plume<u>s</u> <u>sont</u> verte<u>s</u>*
>
> *Les <u>plumes</u> sont verte<u>s</u>*

In the first case, the features of the sentence which make it plural are marked. All these parts must agree even though they may be separated by any amount of intervening structure. Likewise, in the second case, the parts of the sentence which express feminine gender must agree.

In the DCG formalism, which is a notational variant of Prolog, these dependencies would be expressed throughout the grammar in a number of rules, including the highest level rule shown here:

```
s(Num,Gen,s(NP,VP)) --> np(Num,Gen,NP),vp(Num,Gen,VP).
```

In this rule, ‘Num’ and ‘Gen’ are variables recording number and gender, respectively. Agreement is assured because of the ‘meta’ rule of Prolog that repeated examples of an instantiated variable within a clause all refer to a single structure. If the first instance of ‘Num’ in the rule is instantiated to ‘singular’ then all the others are too. Likewise for ‘plural’. The ‘Gen’ variable behaves in the same way for ‘masculine’ and ‘feminine’.

With SP, the same effect is achieved without any need for the meta rule. Figure 10 shows a small grammar which can generate the example sentence amongst others. The ‘top level’ rule is simpler than the top level rule in the DCG formalism because it does not attempt to record discontinuous dependencies at this level. The grammar as a whole is significantly smaller than the equivalent DCG grammar.

```
[
(S(NP,_)(VP,_))
(NP(D,_)(N,_))
(VP(V,_){(A,_)((P,_)(NP,_))})
(P{sur,sous, ...})
(V{(SING{est, ...})(PL{sont, ...})})
(D{(SING{(FEM{une,la, ...})(MASC{un,le, ...})})(PL{les, ...})})
(N(NS,_)(SUF1,_))
(NS{(FEM{plume, ...})(MASC{papier, ...})})
(SUF1{(SING,0)(PL,s)})
(A(AS,_)(SUF2,_)(SUF1,_))
(AS{noir,vert, ...})
(SUF2{(FEM,e)(MASC,0)})
(_,D,SING,_,N,_,SUF1,SING,_,V,SING,_{(A,_,SUF1,SING,_)_})
(_,D,PL,_,N,_,SUF1,PL,_,V,PL,_{(A,_,SUF1,PL,_)_})
(_,D,_,FEM,_,N,_,FEM,_{(A,_,SUF2,FEM,_)_})
(_,D,_,MASC,_,N,_,MASC,_{(A,_,SUF2,MASC,_)_})
]
```

Figure 10. A fragment of French syntax written in SP

Discontinuous dependencies for ‘singular’, ‘plural’, ‘feminine’ and ‘masculine’ forms are expressed by the last four rules in the grammar, in that order. The ‘_’ symbol, which will match arbitrarily large amounts of structure, appears wherever there is structure which is irrelevant to the dependencies being expressed. In this way the dependencies can be shown directly and plainly as OAOs.

7.5 Pattern Recognition and Inductive Learning

Perhaps the most interesting aspect of SP is its potential for pattern recognition, for inductive learning and for their integration with other areas of computing.

Computing in SP means searching for patterns which match each other and unifying them; where alternative unifications are possible, which is almost always the case, then the SP system must choose the ‘best’ one, meaning the one which allows the greatest amount of unification and a correspondingly large simplification of structure.

This and the way SP treats partial matching means that a working SP system is likely to be applicable to practical problems of pattern recognition. It is likely to have the kind of flexibility long recognized in human pattern recognition: the ability to recognize whole patterns from partial patterns and to recognize patterns despite distortion or fragmentation.

The subject of the second paragraph in this section was computing. But essentially the same thing can be said about inductive learning. The inductive learning of a grammar, for example, can be achieved by processes which at heart are pattern matching and unification, with the selection of ‘good’ (meaning economical) structures in preference to ‘bad’ ones (see Wolff, 1982, 1987, 1988).

Here is a very simple example to illustrate how an SP system may achieve inductive learning:

```
[
(John,runs)
(Mary,runs)
(John,walks)
(Mary,walks)
]
```

would be reduced by the SP system to:

```
({John,Mary}{runs,walks})
```

This is functionally equivalent to a grammar which can ‘generate’ the four sentences from which it was derived.

To infer grammars like those in Figures 5 and 10, the SP system would need to be able to create new identifiers for structures (eg NP in (NP{john,mary}) and VP in (VP(V,_)(NP,_)) and introduce them at appropriate points.

The potential applications of a system with inductive learning capabilities include: automation of the process of building natural language grammars; automation of the process of ‘knowledge acquisition’ for expert systems; optimization of database structures; inductive learning of structures for pattern recognition; automatic programming.

The inductive properties of SP give the added dimension of adaptability to pattern recognition applications. A pattern recognition system based on SP is likely to have the capability to learn new general schemas from unprocessed input. It has long been recognized that much of the power of human pattern recognition in, for example, speech recognition, lies in the dynamic adaptation of the system to changes in the input. Pattern recognition and inductive learning are intimately related.

7.6 Other Applications and Attributes

There is no space in this article to discuss fully all the possible applications of SP. This section will briefly describe some other areas where SP may be applied and properties of SP which may prove useful.

- **Configuration Management**. In software development and many other applications of computers there is a need to control the ‘versions’ or ‘variants’ of programs, documents and other information objects. Most such objects come in parts and sub-parts and each such part or sub-part may come in more than one version. The main problem is to keep track of the required combinations of versions and parts. A related problem is to keep track of associations between objects as, for example, between each item of source code and its corresponding object code.

Many of the ideas described in section 7.3 - on the representation of class-inclusion relations, part-whole relations and ‘relationships’ between entities - may be applied to problems of configuration management.

There is scope here for the **integration** of configuration classes with other classes in a software system: the creation of a new version of a body of software would mean the creation of new sub-class of the appropriate class in the system.

- **Software Re-Use**. A significant problem in the software industry is the difficulty of incorporating existing software in new developments and the consequent waste of development effort. This difficulty arises from three main sources:

  - The diversity of languages and formalisms used in software development makes integration difficult.

  - Even when old and new software are written in the same language, many programming languages have only limited means of integrating old with new. In particular, most programming languages lack the OO mechanisms of inheritance which can greatly facilitate the integration of old software with new.

  - There are difficulties in identifying and retrieving software to serve a given purpose.

  SP may alleviate these problems on all three fronts:

  - It has potential as a ‘broad-spectrum’ language with a wide range of applications. It has potential as a standard for the organization of software systems and may thus promote the integration of old software with new.

  - Through the inheritance mechanism which is an implicit part of its organization, SP, like other OO languages, facilitates integration.

  - Since SP has potential as a language for information retrieval (see below) it may prove useful in the identification of existing software to serve a given purpose.

- **Rules for Expert Systems, Probabilistic Inference and Reasoning with Uncertain and Incomplete Knowledge**. ‘Production rules’ of the kind commonly used in expert systems may be expressed directly in SP. For example, a rule to express the common observation that "Clouds (probably) mean rain" may be expressed as

  [[cloud,{black,white}]{rain,hail,snow}]

  If this rule is matched with [[cloud,black],_] representing the question "What do black clouds mean?", the object [[cloud,black]{rain,hail,snow}] will be returned showing that rain is a possibility.

The rule may work backwards: from the existence of rain may be inferred the certainty of there being clouds. In general SP supports the retrieval of complete patterns from partial patterns. It will also support forwards and backwards chaining.

It should be clear from this example that SP can support the drawing of 'probabilistic' inferences. It would be natural in making these inferences to utilize the 'weights' mentioned earlier, reflecting the frequencies of objects in the raw data.

Uncertainty can be represented directly in SP using OROs. Gaps in knowledge may be represented with variables ('_'). The design of SP accommodates uncertainty and incomplete knowledge without *ad hoc* provision.

- **The Representation of Plans and Automatic Planning**. There is a good correspondence between the main constructs of SP and those commonly recognised in project planning:

  - An OAO may be used to represent a **sequence** of activities.

  - A UAO may be used to represent activities which are **independent** of each other, where their ordering is not important. This corresponds to the slightly innaccurate use of the term *parallel* in project planning. 'Parallel' activities may be performed in parallel or they may be performed in some arbitrary sequence, depending on the resources available and the required timescales.

  - An ORO may be used to represent activities which are **alternatives** in a plan. This is not very common in ordinary projects but the concept is recognised in the term 'contingency planning'.

  What is missing from SP as a medium for expressing plans is any explicit concept of 'iteration'. However, an equivalent effect may be achieved by means of recursion.

  It is not possible to be very precise at this stage, but it seems likely that the inferential processes embodied in the semantics of SP will lend themselves to the kind of 'automatic planning' which figures in some commercially available systems to support project planning and which has been the subject of extensive research (see, for example, Tate, 1985).

- **Database Organization, Information Retrieval and Content Addressable Memory**. SP provides a framework for organizing information in databases which is similar to, but not identical with the relational model. It would be rash at this stage to claim that SP were superior to the relational model (although it may be) but it clearly has potential in this area.

  Amongst the potential benefits of an SP database are:

  - Integration of database constructs with 'programming' and software design constructs.

  - Integration of databases with expert systems.

    - The economy and integration which may be achieved by the use of a single language both for structuring the database and for querying it. The example, above, of querying an expert system rule base illustrates the potential of SP as a flexible ‘query-by-example’ means of accessing information in a database.

    - An SP system may retrieve information using special keys or identifiers but it also provides the functionality of **content addressable memory**.

    - By virtue of the pattern matching and unification mechanisms, an SP database may have capabilities for automatic normalisation of the database and removal of redundancies.

- **Software Design and the Formal Specification of Computing Systems**. The way SP supports OO constructs has already been discussed at length. The benefits of these constructs in software design is in the creation of software which is easy to understand, to modify and to maintain. However, there are other potential benefits of SP in software development.

  The simple and regular nature of SP, its close relation to logic (discussed above), and its complete independence of any machine-oriented concepts, means that it should be regarded as a specification language and not a ‘programming’ language in the conventional sense.

  Like Prolog, it has potential as an ‘executable’ specification language. An SP specification may run immediately without the need for the time consuming and error-prone processes of ‘refining’ a specification to become an ‘implementation’ or ‘program’ followed by ‘verification’ that the ‘program’ conforms to the specification. Of course, it does not eliminate the need to **validate** the specification against the user’s requirements.

  Much software design entails the recreation of routines for pattern matching and unification in a variety of guises. By providing powerful and ‘universal’ methods of processing patterns, SP can save much re-invention of the wheel.

- **Translation between computer languages and porting of software**. SP has potential application in the translation between computer languages which is needed from time to time. SP may also facilitate the porting of software from one machine to another (since porting of software may be regarded as a form of translation).

  In any kind of translation (including translation between natural languages) there are advantages in using an ‘interlingua’ or base form as an intermediary between languages. The main reason for this is that the number of translators which need to be developed may be reduced: for all combinations of N languages the number of two-way translators required is N! without an interlingua but only N with an interlingua.

  If SP does indeed capture the essentials of computing in the way which has been claimed then it should be a good choice as an interlingua for translation between existing computer languages.

- **Integration of diverse kinds of knowledge**. A common problem in computing is the need for integration - the need to have simple and uniform methods of storing and using diverse kinds of information. In the preparation of documents, for example, there is a need for an integrated approach to the storage and manipulation of text and diagrams. In databases to support computer-aided design and in integrated project support environments (IPSEs) there is a similar need to be able to integrate the varied kinds of knowledge which are used.

  SP has potential to provide the general format which is needed for the uniform storage and use of diverse kinds of knowledge.

  It may not always be convenient to map every body of knowledge to the SP syntax. In these cases, SP can serve a useful purpose as a framework within which 'alien' bodies of knowledge may be stored. It can, for example, be used to show (in a UAO) the association between a program written in, say, COBOL, and its corresponding compiler.

- **The user interface**. The usability of any system depends, in part, on simplicity in its organization and consistency across a wide range of applications. The simple syntax of SP and the scope of its semantics can mean a simple user interface which is consistent in a variety of applications.

8. CONCLUSION

SP and the theory on which it is based has potential to integrate and rationalize diverse concepts in computing. The basic conjecture is that the organization and use of all kinds of formal system, knowledge structure or computing system may usefully be seen in terms of the management of redundancy in information.

In this article I have tried to show with examples how a few carefully chosen basic constructs may illuminate a wide range of concerns in computing. No attempt has been made to show how SP would apply to such things as numerical computing, the representation and use of (2D and 3D) geometric information, concepts associated with the interface between a computer and its human user, or the representation and use in computational form of concepts of time.

There is a substantial programme of work needed to explore the adequacy or otherwise of SP constructs in the kinds of areas mentioned. If SP and its underlying theory are sound then they should have something useful to say about areas like these. It should be possible to create 'higher level' constructs from the basic constructs in SP to meet the needs of particular domains.

It may be that the basic constructs in SP will need to be modified or augmented. However, in all circumstances, we should resist the temptation to postulate new basic constructs in an *ad hoc* manner. In accordance with the best scientific traditions, we should try to achieve descriptive and explanatory power with a small range of theoretical devices. In short, we should aim in our thinking for a favourable combination of simplicity and power.

9. ACKNOWLEDGEMENTS

Earlier drafts of this article have been circulated quite widely and several people have made useful comments. I am particularly grateful to Simon Tait of Praxis Systems plc with whom I have discussed these ideas extensively. I am also grateful for helpful comments from James Davenport of the University of Bath; Tim Denvir, Anthony Hall, Trevor King, Paul Newman, Martyn Ould and Lynn Robinson of Praxis Systems plc; Robert Kowalski of Imperial College, London; Peter Van Peborg of Plessey Research; James Bond of the Central Electricity Generating Board, Bristol; and Simon Jones of the School of Electronic Engineering Science, University of Wales (Bangor).

10. REFERENCES

Birtwistle G M, Dahl O-J, Myhrhaug B & Nygaard K (1979). *Simula Begin*, New York: Van Nostrand Reinhold.

Bohm C & Jacopini G (1966). Flow diagrams, Turing machines, and languages with only two formation rules. *Communications of the ACM 9(5)*, 366-371.

Cook S (1986). Languages and object-oriented programming. *Software Engineering Journal 1(2)*, 73-80.

Hinton G E & Sejnowski T J (1986). Learning and relearning in Boltzmann machines. Chapter 7 in D E Rumelhart and J L McClelland (Eds), Vol I, pp 282-317.

Jackson M A (1975). *Principles of Program Design*. London: Academic Press.

Rumelhart D E & McClelland J L (Eds)(1986). *Parallel Distributed Processing*, Vols I and II, Cambridge Mass.: MIT Press.

Structured Systems Analysis (1986). Learmonth & Burchett Management Services, London.

Peeling N E, Morison J D & Whiting E V (1984). ADAM: an abstract database machine. Report No. 84007, Royal Signals and Radar Establishment, Malvern, UK.

Pereira F C N & Warren D H D (1980). Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence 13*, 231-278.

Tate A (1985). A review of knowledge-based planning techniques. In M Merry (Ed), Expert Systems 85: Proceedings of the Fifth Technical Conference of the British Computer Society Specialist Group on Expert Systems, Cambridge: Cambridge University Press, 89-112.

Ungar D & Smith R B (1987). Self: the Power of Simplicity. Proceedings of the Conference on Object Oriented Programming Systems and Languages (OOPSLA '87), 227-241.

Winston P H (1984). *Artificial Intelligence*, Reading, Mass.: Addison-Wesley, Second Edition. (1977: First Edition).

Wolff J G (1982). Language acquisition, data compression and generalization. *Language & Communication 2(1)*, 57-89.

Wolff J G (1987). Cognitive development as optimization. In L Bolc (Ed), *Computational Models of Learning*, Heidelberg: Springer-Verlag, 161-205.

Wolff J G (1988). Learning syntax and meanings through optimization and distributional analysis. In Y Levy, I M Schlessinger and M D S Braine (Eds), *Categories and Processes in Language Acquisition*, New York: Lawrence Erlbaum, 179-215.

11. APPENDIX: THE CALCULATION OF SIZE, POWER AND EFFICIENCY OF GRAMMARS

The size ($\mathbf{s_r}$), in bits, of a body of raw data may be calculated as

$$\mathbf{s_r = n_r \cdot c_r}$$

where:

- $\mathbf{n_r}$ is the number of symbols used in the raw data.
- $\mathbf{c_r}$ is the 'cost' (in bits) of each symbol calculated as

  $$\mathbf{c_r = \log_2 t_r}$$

  rounded up, where $\mathbf{t_r}$ is the number of symbol types used in the raw data.

The size ($\mathbf{s_e}$), in bits, of a body of data after encoding by a grammar may be calculated as

$$\mathbf{s_e = n_e \cdot c_e}$$

where:

- $\mathbf{n_e}$ is the number of symbols required to describe the encoded data using the grammar.
- $\mathbf{c_e}$ is the 'cost' (in bits) of each symbol calculated in the same way as $\mathbf{c_r}$.

The size ($\mathbf{s_g}$), in bits, of a grammar may be calculated as

$$\mathbf{s_g = n_g \cdot c_g}$$

where:

- $\mathbf{n_g}$ is the number of symbols used in the grammar excluding any symbols which are required only for reasons of readability (eg some instances of meta-symbols)
- $\mathbf{c_g}$ is the 'cost' (in bits) of each symbol calculated in the same way as $\mathbf{c_r}$.

The descriptive power ($\mathbf{p_g}$) of a grammar may be defined, with reference to a given body of raw data, as

$$\mathbf{p_g = s_r - s_e}$$

The efficiency ($\mathbf{e_g}$) of a grammar, with reference to a given body of raw data is:

$$\mathbf{e_g = p_g / s_g}$$

12. FIGURE CAPTIONS

Figure 1. Data and grammars to illustrate concepts of simplicity and power.

Figure 2. An abstract space for grammars and computing systems.

RAW DATA

A,B,C,D,P,Q,R,A,B,C,D,A,B,C,D,P,Q,R,A,B,C,D,P,Q,R,P,Q,R,A,B,C,D

PRIMITIVE GRAMMAR 1

1 --> A,B,C,D,P,Q,R,A,B,C,D,A,B,C,D,P,Q,R,A,B,C,D,P,Q,R,P,Q,R,A,B,C,D

Encoding of raw data: 1

PRIMITIVE GRAMMAR 2

1 --> A
2 --> B
3 --> C
4 --> D
5 --> P
6 --> Q
7 --> R

Encoding of raw data: 1,2,3,4,5,6,7,1,2,3,4,1,2,3,4,5,6,7,1,2,3,4,5,6,7,5,6,7,1,2,3,4

WELL STRUCTURED GRAMMAR 1

1 --> 2,3,2,2,3,2,3,3,2
2 --> A,B,C,D
3 --> P,Q,R

Encoding of raw data: 1

WELL STRUCTURED GRAMMAR 2

1 --> A,B,C,D
2 --> P,Q,R

Encoding of raw data: 1,2,1,1,2,1,2,2,1

BADLY STRUCTURED GRAMMAR 1

1 --> A,B,3,2,C,D,A,B,3,2,3,Q,R,P,2,C,D
2 --> Q,R,A,B
3 --> C,D,P

Encoding of raw data: 1

BADLY STRUCTURED GRAMMAR 2

1 --> Q,R,A,B
2 --> C,D,P

Encoding of raw data: A,B,2,1,C,D,A,B,2,1,2,Q,R,P,1,C,D

Figure 1

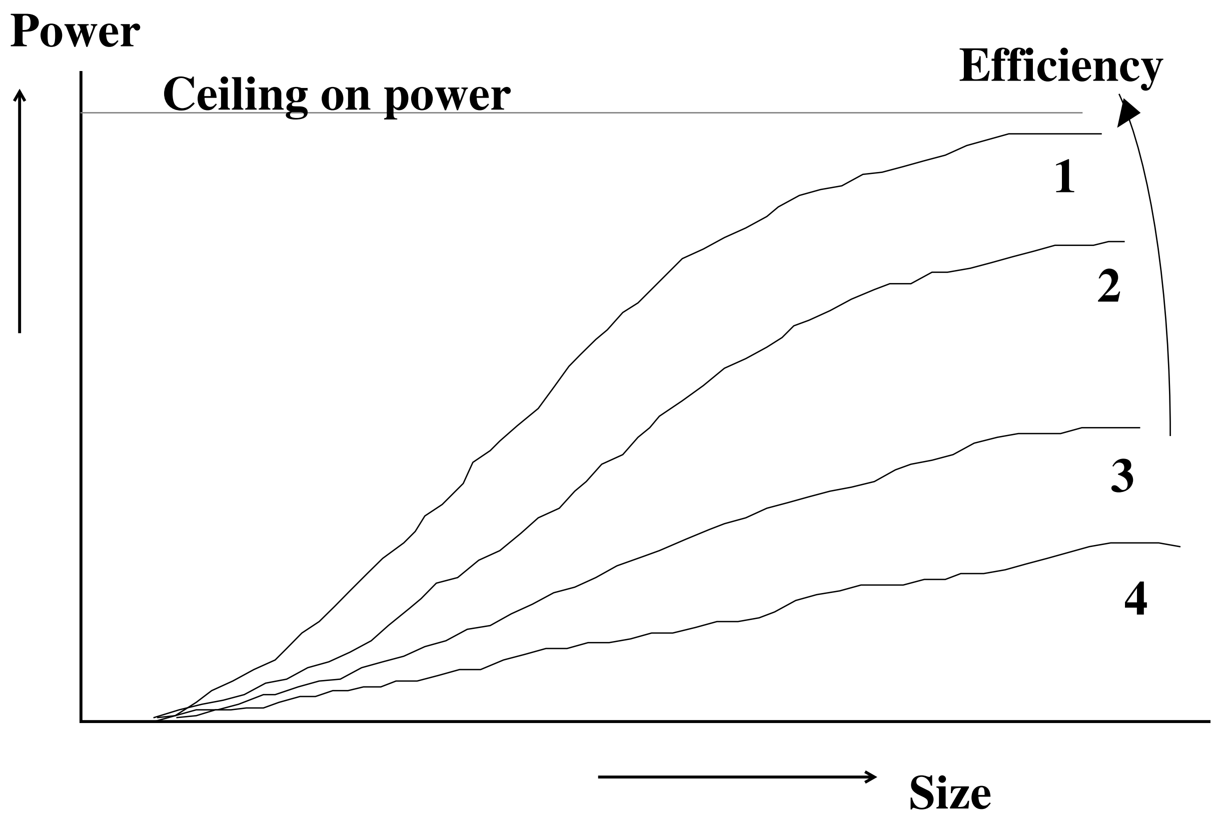

Figure 2