LOGIC PROGRAMMING: CONTEXT, CHARACTER AND DEVELOPMENT

Thomas H. Conlon B.Sc. (Hons)

A thesis submitted to the University of Glasgow for the degree of Master of Science

Department of Computing Science
University of Glasgow

January 1986

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Acknowledgements

The ideas presented in this thesis are due to the logic programming community worldwide. Their vigour, creativity and imagination has made this modest synthesis possible, and the debt is acknowledged in full.

Special thanks are due to those members of the logic programming community with whom I have had the pleasure of personal contact. Included here are Jonathon Briggs, Steve Gregory, and Keith Clark. I am grateful to my supervisor, Dr Rob Sutherland, for his support and for his encouragment of my enthusiasm. Finally, acknowledgment is due to the Boards of Governors of Jordanhill and Moray House Colleges of Education, in whose employment I have been and on whom I have depended for their goodwill towards my logic programming activities.

ABSTRACT

Logic programming has been attracting increasing interest in recent years. Its first realisation in the form of PROLOG demonstrated concretely that Kowalski's view of computation as controlled deduction could be implemented with tolerable efficiency, even on existing computer architectures. Since that time logic programming research has intensified. The majority of computing professionals have remained unaware of the developments, however, and for some the announcement that PROLOG had been selected as the core language for the Japanese 'Fifth Generation' project came as a total surprise.

This thesis aims to describe the context, character and development of logic programming. It explains why a radical departure from existing software practices needs to be seriously discussed; it identifies the characteristic features of logic programming, and the practical realisation of these features in current logic programming systems; and it outlines the programming methodology which is proposed for logic programming. The problems and limitations of existing logic programming systems are described and some proposals for development are discussed.

The thesis is in three parts. Part One traces the development of programming since the early days of computing. It shows how the problems of software complexity which were addressed by the 'structured programming' school have not been overcome: the software crisis remains severe and seems to require fundamental changes in software practice for its solution. Part Two describes the foundations of logic programming in the procedural interpretation of Horn clauses. Fundamental to logic programming is shown to be the separation of the logic of an algorithm from its control. At present, however, both the logic and the control aspects of logic programming present problems; the first in terms of the extent of the language which is used, and the second in terms of the control strategy which should be applied in order to produce solutions. These problems are described and various proposals, including some which have been incorporated into implemented systems, are described. Part Three discusses the software development methodology which is proposed for logic programming. Some of the experience of practical applications is related. Logic programming is considered in the aspects of its potential for parallel execution and in its relationship to functional programming, and some possible criticisms of the problem-solving potential of logic are described. The conclusion is that although logic programming inevitably has some problems which are yet to be solved, it seems to offer answers to several issues which are at the heart of the software crisis. The potential contribution of logic programming towards the development of software should be substantial.

CONTENTS

1 THE CONTEXT OF LOGIC PROGRAMMING

Logic programming is a comparatively recent arrival on the computing scene. As a latecomer, it must compete with programming styles which are better established and which have at the very least, the advantages of familiarity. It follows that the success of logic programming will be influenced by the extent to which it offers solutions to problems which have proved to be difficult or insoluble within existing approaches. Software engineers are not going to abandon their existing methods and tools without good reason: they will consider doing so only if they are convinced that these methods and tools are inadequate and that alternatives exist which not only remedy the inadequacies but which are also demonstrably superior by any fair set of criteria.
This section, then, explores the context of logic programming. It begins by recalling the original von Neumann concept of programming. Machine-language programming is shown to have been quickly displaced by the development of language translators, and early unstructured methods are shown to have led to the discovery of a 'software crisis'. The main response of the computing community in the nineteen-sixties and seventies was the development of the 'structured programming' movement with an associated family of programming languages and development methodologies. However, the so-called 'high-level' languages have been largely faithful to the machine-oriented view of programming. Although the structured programming software development methodology did make an important contribution to the advancement of software science, particularly through its emphasis on the application of abstraction and modularity, it remains true that the efficient development of correct, reliable, maintainable software is beset with problems. At least some of these problems, such as that of constructing programs which can exploit parallel computer architectures, appear to require a solution which departs radically from the von Neumann approach to programming.

1.1 Programming and the von Neumann machine

The originator of modern electronic computing is generally recognised to be John von Neumann, the mathematician whose design is still today the basis of computer architecture. It is appropriate therefore to begin with a brief examination of the von Neumann design.
The Princeton papers [2, 3] in which von Neumann set out his ideas specify a computer having two principal parts, which

we shall identify as the Central Processing Unit (CPU) and the store. Conceptually, the CPU is the 'mill' which operates on the raw material of data located in the store; the store is also the destination-place of the processed data. The store is partitioned into a number of different locations, each location being capable of holding one word of data - a word being some fixed number of binary bits, depending on the machine - and each being associated with a unique numeric address. The CPU and the store can be thought of as connected by a tube, through which the CPU can read the word in any location and write a word in any location. The CPU can perform basic operations on words of data, such as the addition of two words, as well as performing operations such as testing whether a word is zero. In a later description of the design [4], von Neumann described the role of programming :-

> '... any computing machine that is to solve a complex mathematical problem must be "programmed" for this task. This means that the complex operation of solving that problem must be replaced by a combination of the basic operations of the machine.'

Programs are described as 'order-systems', which are to be located in the store along with the data:-

> 'Of course, the order-system - this means the problem to be solved, the intention of the user - is communicated to the machine by "loading" it into the memory. This is usually done from a previously prepared tape or some other medium.'

An 'order' is described as

> '... physically, the same thing as a number ... an order must indicate which basic operation is to be performed, from which memory registers the inputs of that operation are to come, and to which memory register its output is to go.'

The orders may include those which transfer control to other orders in the program -

> '"Branching" is most conveniently handled by a "conditional transfer" order, which is one that specifies that the successors address is X or Y,

> depending on whether a certain numerical condition has arisen or not - e.g. whether a number at a given address Z is negative or not'.

As to the question of precisely which basic operations the CPU should be able to perform, von Neumann suggested that this should be influenced by the type of program which was to be executed -

> 'For a given class of problems one set of basic operations may be more efficient, i.e. allow the use of simpler, less extensive, combinations, than another such set.'

1.2 The development of translators

Although it was written in the earliest years of computing, the above quotations from von Neumann are still quite recognisable as a description of programming as it is usually practised today. Indeed, a personal computer enthusiast working in the machine code of his machine's microprocessor might well be struck by the apparent absence of any advances in the years which have elapsed since. In fact, however, the writing of 'order-systems' strictly in terms of the 'basic operations' of the machine - that is, machine language programming - was rapidly overtaken by the development of translators. The first of these were assemblers which offered convenient mnemonics for the machine language instructions: each mnemonic statement of the assembly language program was compiled by the assembler into a single machine language instruction. Soon more powerful translators were developed, partly to help with special types of programming, and partly to enhance the convenience of the assemblers. The earliest so-called high level languages evolved naturally through this process of gradually extending translational power. An example of these languages was FORTRAN, which enabled the programmer to refer to variable names instead of register addresses, and which could compile a mathematical expression such as

```
0.5 * B * C * SIN(A)
```

into corresponding machine language instructions. So advantageous were these translators over raw machine language programming that they were initially named 'Automatic Programming Systems', and the 'Communications' of May 1959 reported that almost 100 systems of this type

(for IBM, UNIVAC, FERRANTI and other computers) were held in the ACM library. With the exception of FORTRAN, these translators - with names such as SPEEDCODING, SYMBOLIC ASSEMBLY, FAST, MYSTIC, BASIC AUTOCODER, ARITHMATIC, DUMBO and SUMMER SESSION - have long since been forgotten.

In general, the early translators were each designed to operate with only one make and type of computer. As computing became more widespread and the types of computer became more diverse, this brought severe problems. A 1958 report from the Ad-Hoc Committee on Universal Languages [5] opened with an expression of concern about

> ' ... the considerable length of time required to develop an effective method of communication with the machine. Morever, it seems that the ability to communicate easily is no sooner acquired than the language changes, and the problem is renewed, usually at a higher level of complexity.'

The Ad-Hoc committee identified the rapid obsolescence of machines, the growing complexity of machine languages, and the fact that compilers for acceptable languages only became available for individual machines as the machines were on the point of being replaced, as being the main causes of problems. In remedy, the committee proposed a Universal Computer Oriented Language (UNCOL): a generator should be written for each existing translator to produce UNCOL code, and a translator should be written for each computer to translate UNCOL code into the corresponding machine language. In the event, the UNCOL system never materialised. Instead, the pressures for standardisation led to the development of the high-level languages ALGOL 60 [6] and COBOL [7], which together with FORTRAN were to dominate the programming of the nineteen-sixties.

1.3 Software Crisis and response

The trend towards standardisation did not, however, prevent the emergence of major problems which the software pioneers were soon to encounter. For although von Neumann had provided a blueprint for a general purpose computer, there was no blueprint explaining how to construct the programs which would enable it to solve real problems. At first, when the problems tackled were straightforward - routine mathematical caculations, for example - the ad-hoc programming methods seemed to work, and no doubt there was even some thrill in the business of 'fixing' the machines to do whatever was required. But when ad-hoc approaches are simply scaled up in an attempt to tackle ever more complex problems, the result is disaster sooner or later. This is exactly what the new computing industry discovered in the nineteen-sixties. The phrase (it was first recorded at the Conference on Software Engineering held at Garmish in 1968) which was coined to describe the state of affairs was 'software crisis'. As Wulf [8] described it in 1977:-

> 'By now it is almost a cliche to say that there is a "software crisis". Nearly everone recognises that software costs more than hardware, and that the imbalance is projected to increase. Nearly everyone recognises that software is seldom produced on schedule. And worse, that the typical software product, costing more and delivered later than originally planned, seldom meets its performance goals; it's bigger, slower, and vastly more error prone than was originally anticipated. The aggregated cost of a failure to meet performance goals, measured in additional resources, time, and reconstruction of data lost due to an error, may vastly outweigh the original development cost.'

Indeed, for a time the very notion that large programs could be developed without trauma virtually lost credibility. As E.E. David [9] wrote in 1971:-

> 'Production of software for large systems has become a scare item for management.'

The case of operating systems provided one manifestation of the crisis. As the hardware advanced, the manufacturers struggled to provide operating systems which would make the increased power usable. The growing complexity quickly exposed the inadequacy of the software techniques which were available. McKeag [10] contrasts the operating system for Cambridge University's TITAN computer of the nineteen-fifties, which comprised 40,000 machine instructions, with the GEORGE 3 operating system for the nineteen-seventies ICL 1900 series, which with 400,000 machine instructions was ten times as large. The written documentation for this software became correspondingly overwhelming. The User Specification Manual for GEORGE by its twenty-third amendment was (when measured in the Imperial units of the time) two-and-a-half inches thick, and its Implementation Manual included an inch-and-a-quarter of what McKeag describes as 'incomprehensible flowcharts'. Brooks [11] reports that during the construction of the operating system OS/360, IBM workers maintained a documenting workbook: when it grew to five feet thick it was increasing at the rate of two inches per day, and only a switch to microfiche prevented further uncontrollable explosion. As for the quality of the software when it finally emerged, Hoare [12] - who had himself led in the mid-sixties a disastrous system software project for Elliot computers which collapsed with the loss of thirty man-years of programming effort - commented in April 1976 that:-

> 'Among manufacturer's software one can find what must be the worst engineered products of the computer age. No wonder it was given away free - and a very expensive gift it was, to the recipient!'

In effect, the ad-hoc languages and programming methods which had sufficed for the small problems in the early days were exposed as grossly inadequate to meet the new demands. In the face of growing complexity, something much better would be required.

1.4 The 'structured programming' school

Of the many voices which were raised in the sixties and seventies in the debate over what to do about the 'software crisis', the most influential were those who can now be identified as comprising the 'structured programming' school. Prominent among them were the names of Dijkstra, Dahl and Hoare [13], Mills [14], and Wirth [15]. The structured programming school viewed the construction of sizeable computer programs as an engineering activity and they looked to engineering traditions as a source of guidance. The control of complexity was recognised as the main problem of software construction: only through the development and application of the right tools and methods could the problem be solved.
In the event, the methods and tools which were developed by the structured programming advocates are those which still today dominate the practice of software engineering. A brief account of them is worthwile here. Software products were identified as having a 'life cycle' which, in a typical delineation comprises the five stages -

1. Specification
2. Design
3. Implementation
4. Testing
5. Operation and maintenance

The major goals relating to software construction were identified: software products should be validateable, verifiable, be reliable, secure, efficient, flexible, maintainable and economic. It was recognised that some of these considerations would at times be in mutual opposition, so that trade-offs would be required. The high costs of the fifth stage of the software life cycle were recognised (Sommerville [16] suggests that these costs typically exceed the other costs combined by a factor of four), and the implication was drawn that, as far as possible, decisions in the earlier stages of development should be based on the need to minimise these high later costs.
Although the structured programming school collectively commented on factors relating to all stages of the software life cycle, their main contribution has been to the development of methods and tools concerned with program design and development. On these subjects a vast library of literature has accrued, and it is apparent from this that 'structured programming' does not neccessarily mean quite the same thing to all of its many exponents. Early writings focussed on particular programming practices which are seen to epitomise the worst of the ad-hoc techniques: an example

of this is the use of the GOTO statement, condemned most notably by Dijkstra [17], with other writers quoting the so-called Structure Theorem of Jacopini [18] which proved that any flowchartable program could be equivalently re-written in GOTO-less fashion by using only the constructs of sequencing, decision-making (IF-THEN-ELSE) and repetition (WHILE-DO-).
Notwithstanding the diversity, two clear principles run through the structured programming literature. These are the principles of abstraction (mainly applied to program design) and of modularity (mainly applied to program implementation). Abstraction is concerned with the selection of essential aspects of a problem and the deliberate subordination of inessential aspects: it has long been recognised as a crucial problem-solving principle, particularly in mathematics. Modularity is a long-established concept of engineering, where it is recognised that it is advantageous in constructing a large system to partition the system into a collection of sub-systems or 'modules' which, though interdependent, can nevertheless be constructed independently. By applying the principle of abstraction to program design, the structured programming school arrived at the programming methodology of stepwise refinement. Wirth summarised this process as follows:-

> 'In each step a given task is broken up into a number of subtasks. Each refinement in the description of a task may be accompanied by a refinement of the description of the data which constitute the means of communication between the subtasks. Refinement of the description of program and data should proceed in parallel.' [15]

The stepwise refinement method would partition the program into a collection of sub-programs. The pursuit of modularity led to the introduction of programming principles which would reduce the coupling between sub-programs; important examples were the principles of information hiding, which means that sub-programs have access only to that information which they actually need, and of localisation, which requires that programs should textually contain their own sub-programs.
In pursuit of tools to support their methods, the structured programming school turned their attention to programming languages. Wirth observed (in 1971) that

> 'It is remarkable that it would be difficult to find a language that would not meet these important requirements

> better than the one language still used most widely in teaching programming: Fortran.'

Building on the positive features of Algol 60, Wirth developed Pascal [19]. Other languages, such as Algol 68 [20] and PL/1 [21], were also influenced to various extents by the requirement to support structured programming. Although these languages have significant differences (as indicated, for instance, by the fact that Wirth's Pascal emerged out of dissension from the Algol 68 development, and by the fact that PL/1 aroused much criticism from Dijkstra [31] and others), they have features in common such as function and procedure sub-program facilities, extensive data typing provision, parameter-passing mechanisms and scope rules, and appropriate repetition and decision constructs; all of which assist in the implementation of modular programs which have been developed by stepwise refinement using the principle of abstraction.

1.5 Unsolved problems of software construction

The contribution to software engineering of the structured programming school has been positive and extensive. Testimony to this fact is abundant today within the journals and records of the practising software engineering profession, in which the concepts of structured programming are predominant. Even inside the commercial data-processing sector, where inertial forces are traditionally strong, the methods of structured programming - albeit clothed in a suitably palatable form (see for example, Jackson [21]) - are widely accepted. Yet whilst it is mainly agreed that the structured approaches are vastly better than the earlier ad-hoc methods, dissatisfaction over software construction methods and tools is still widespread. This is well expressed by Darlington, writing in August 1985:-

> 'Most professional (and amateur) programmers would like to claim that what they do is scientific, but compared with the standards attained in other, more mature engineering disciplines such as aeronautical or civil engineering, programming has a long way to go. If one were asked to build a bridge, I doubt that it would be acceptable to construct an initial version, try it out, and, when it falls down, correct the mistakes made in the design, and then repeat the process until the bridge stays up. This

> is, however, the paradigm that most practising programmers follow as they debug their programs towards a working state.' [197]

Software is still often unverified, insecure, inflexible, inefficient, difficult to maintain and costly to produce. In these respects, the 'software crisis' has not been resolved by the contributions of the structured programming school. The following sections describe some of the problems more fully.

1.5.1 The referential opacity of programs

As has been described, Von Neumann characterised a computer program as an 'order sequence' which solves problems by successively re-calculating and re-assigning values to the locations of the store. Whilst this concept has provided an operational basis for the first few generations of digital computers, it has been less satisfactory in providing human beings with a model with which they can reason about programs. This is well expressed in the observation of Glaser, Hankin and Till [23] that :-

> '... the notion of a global state that may change arbitrarily at each step of the computation has proved to be both intuitively and mathematically intractable.'

Backus has suggested that the statements of imperative programs, and in particular the assignment statements, create an 'unorderly world' which is 'conceptually unhelpful' [1]. He gives as an example the fragment of Algol:-

```
c:= 0
FOR i:= 1 STEP 1 UNTIL n DO
  c:= c + a[i] * b[i]
```

Backus writes that:-

> 'Its statements operate on an "invisible state" according to complex rules ... it is dynamic and repetitive. One must mentally execute it in order to understand it.'

He concludes:-

> 'Von Neumann languages do not have useful properties for reasoning about programs.'

The term 'referential opacity' is now widely used to describe the property of programs whereby the same expression may have different evaluations at different points in the program's execution history. In general, programs written in conventional languages which allow the free use of the assignment statement are referentially opaque. Darlington [197] gives an example of a Pascal program in which the consecutive statements:-

```
writeln(g(2) + f(1));
writeln(f(1) + g(2));
```

respectively produce the output 10 and 8. He points out:

> 'Thus, commutativity, one of the simplest manipulation laws ... does not apply to Pascal programs.'

The assignment statement of course corresponds to the update operation on the store of the von Neumann computer. As such it is fundamental to the traditional, imperative view of programming. The identification of assignment therefore as a major source of program opaqueness presents a profound challenge, but if the problem of finding ways to reason effectively about programs is to be solved then it is a challenge which appears to be unavoidable.

1.5.2 Predominance of informal development methods

In his recent review of current software engineering practices, Sommerville [16] records the following tools as being among those commonly applied at the development stages of the software life cycle.

1. Specification stage: free English prose, structured English, high-level formal requirements languages, prototype development tools.

2. Design stage: Data flow diagrams, structure charts, HIPO charts, high-level design description languages, structured walkthroughs.

3. Implementation stage: Programming languages, tools of the programming environment.

4. Testing and debugging stage: Code inspections, top-down and bottom-up testing, test data generators, execution flow summarisers, file comparators, symbolic dump programs, program trace packages, static program analysers.

In general, a sofware project will involve some combination of these tools and methods, selected according to the prevailing preferences of the developers. For example, at the design stage Sommerville indicates a preference for a progression which involves data flow diagrams, structure charts and pseudo-formal design languages.
There is no doubt that all of the tools and methods have been useful in practical experience, and that they represent an attempt by programmers to advance beyond the wholly ad-hoc efforts of the early years of computing. However, it is clear that the lack of continuity of tools and methods between and within the stages of software development impedes efficient software construction. Each transition represents a fracture point which is a source of difficulty and potentially of error. As Wasserman [225] observes in a recent report,

> 'The state of the art of software tools leaves much to be desired ... there are few settings in which the tools actually work effectively together and in support of a software development methodology.'

It can be noted that many of the methods and tools still have a strong ad-hoc flavour: this is especially true in the case of graphical tools such as structure charts and data flow diagrams which have no real formal basis. Their use makes it rather difficult to reason with certainty about the correctness of the development of software. Sommerville for example warns that the task of deriving the 'most appropriate' structure chart from a data flow diagram represents a 'major problem' for the software engineer.
One consequence of the liberal use of informal methods is that the vast majority of software which is being developed today is not formally verified, but is only 'tested' in a manner which, as is well understood, is capable of showing the presence of errors but is quite incapable of showing their absence. As computers are used to tackle ever more complex problems, in which they are entrusted with responsibilities (such as control of air traffic, safety monitoring of nuclear power plants, and acting as expert systems in medical and other domains), so the requirement

that software be correct and reliable becomes more vital. Neither has the proliferation of informal methods and tools been successful in lowering the cost of software; or not, at any rate, by comparison with the changing cost of hardware. As Turner has pointed out recently, over the past two decades there has been a dramatic reduction in hardware costs and over the next decade VLSI developments are expected to reduce hardware costs 'practically to zero', whereas

> 'There has been no corresponding reduction in software costs. In real terms (man-hours or whatever) it costs about the same to produce a given piece of software today as it did fifteen years ago.' [226]

Many researchers have come to the view that the efficient production of verifiable software will depend on the development of formal methods and tools. Thus, Darlington suggests (somewhat polemically) that:-

> 'Our goal should be the precision of mathematics. Noone feels the need to debug a mathematical theorem or relies on laws that are probably correct apart from a few residual bugs. Programs are superficially similar to mathematical notations, so why can't we share their degree of certainty?' [197]

However, he goes on to suggest that a necessary condition for this to happen is abandonment of conventional languages. The referential opacity of programs written in these languages which was referred to above renders them intractable to formal methods:

> 'Referential opacity means that a system's behavior may be time-dependent; i.e. the meaning of a fragment may depend on the history of what happened prior to the evaluation of that fragment. No simple, meaning-preserving, deductive rules can be developed for that system.'

1.5.3 Dissatisfaction with programming languages

Since programming languages are implicated in the general condemnation of software development which is expressed above, it is hardly surprising that complaints are frequently heard about them. Perhaps, indeed, the wonder is that dissatisfaction is not more widespread. Today's commonly used programming languages are criticised for many reasons, but two criticisms which seem to arise particularly often are that programming languages are not sufficiently problem-oriented and that they are becoming too complex. Another is that in spite of their differences, languages such as FORTRAN, PL/1, Pascal and Ada have fundamental similarities which might suggest that the problems will not be solved merely by further refinements and extensions.
The criticism of programming languages as non problem-oriented usually refers to the large gap which lies between the specification of software and the implementation stage at which the programming languages start to become useful. A specification provides a description of a requirement, usually in terms of the relationship which should exist between the expected input and the output. Implementation on the other hand cannot proceed until an algorithm is identified which corresponds to the relationship. Programming languages enable algorithms to be expressed in an executable form, but in general they offer no help in their discovery. In practice, the gulf between the descriptive specification of what is to be computed and the algorithmic specification of how it is to be done is wide. A more problem-oriented programming language would start to become helpful closer to the specification; the programming language should enable the programmer to formulate some kind of computer-intellible problem specification at an earlier stage than is now possible with conventional languages.
An evident trend in the development of programming languages is their growing complexity. Hoare [29] notes that :-

> 'Programmers are always surrounded by complexity; we cannot avoid it. Our applications are complex because we are ambitious to use our computers in ever more sophisticated ways. Programming is complex because of the large number of conflicting objectives for each of our programming projects. If our basic tool, the language in which we design and code our programs, is also complicated, the language itself becomes part of the problem rather than part of its solution.'

Language designers have tended to equate language power with the provision of large numbers of language features. Hoare argues that this is a fallacy and that the engineering maxim that 'the price of reliability is the pursuit of the utmost simplicity' holds good for language design. However, his advice to this effect, when supplied to the working parties responsible for the design of first Algol 68 [20], then PL/1 [21] and most recently Ada [30], was ignored on each occasion :-

> 'Gadgets and glitter prevail over fundamental concerns of safety and economy.'

Dijkstra [31] points to the harm which language over-complexity (as instanced by PL/1) does to the thinking skills of programmers :-

> 'Using PL/1 must be like flying a plane with 7,000 buttons, switches and handles to manipulate in the cockpit. I absolutely fail to see how we can keep our growing programs firmly within our intellectual grip when by its sheer baroqueness the programming language - our basic tool, mind you! - already escapes our intellectual control.'

Backus [1] comments that:

> 'For twenty years programming languages have been steadily progressing towards their present condition of obesity.'

and he observes that :-

> 'Since large increases in size bring only small increases in power, smaller, more elegant languages such as Pascal continue to be popular. But there exists a desperate need for a powerful methodology to help us think about programs and no conventional language even begins to meet that need.'

Nor even would every beginning student of computing science agree that Pascal is quite so 'small'. Indeed, writing seven years after he developed the language, Wirth [32] himself noted :-

> 'My primary conclusion is that Pascal is a language which already approaches the [limits of] complexity, beyond which lies the land of diminishing returns.'

The growing complexity of programming languages is both a response to the software crisis and a contributory factor to it. Large languages attempt to provide the maximum of support for structured programming, but at the same time their complex syntax and semantic rules extend the range of possible errors and make correctness arguments more difficult. Furthermore, they make unwieldy tools for thought. At the same time, there is agreement that a simple, elegant language is not the same thing as a simple-minded, naive one, as in (for example) BASIC or FORTRAN.
The idea that conventional languages are fundamentally alike is well expressed by Backus. He writes:-

> Conventional languages are based on the programming style of the von Neumann computer. Thus variables = storage cells; assignment statements = fetching, storing and arithmetic; control statements = jump and test instructions.' [1]

The development of FORTRAN first introduced the abstractions which are identified in this comment. Essentially the same abstractions as those of FORTRAN have been applied to the design of almost all languages since that time, and it can be observed that the observation applies with total validity to Ada, which of course is one of the most recent arrivals. Although there is no doubt that Ada is a vastly more complex and sophisticated product than FORTRAN, it can be expected that it too will have inherited any basic weaknesses which exist in the von Neumann design.

1.5.4 Lack of scope for exploiting parallelism

As has been noted earlier, the von Neumann computer design is built around the assumption that only a single processor will be available. As Backus [1] points out, this design incorporates a 'bottleneck' which impedes computation :-

> 'The task of a program is to change the contents of the store in some major way; when one considers that this task must be accomplished entirely by pumping single words back and forth throught the von Neumann bottleneck, the reason for its name becomes clear.'

It now seems likely that advances in hardware design will produce new, multi-processor computers which offer an escape

from the von Neumann bottleneck. Recent consideration has been given to architectures based on the data-flow model of computation (see for example Treleaven, et al [33]) for which an experimental machine is presently under development at Manchester University [34].
The question arises as to how programs will be constructed for parallel machines, assuming they can be built. Programming systems which incorporate an allowance for parallelism have been developed on the basis of conventional languages (for example, Dijkstra's PARBEGIN/PAREND [35], Hoare's parallel execution commands and CSP [36], the Concurrent Pascal of Brinch-Hansen [37] and the Path Pascal of Campbell and Kolstad [38]) but these have typically been systems based on sequential execution with augmented facilities for programmer-specified parallelism. It now appears however that this approach will fail to exploit the extent of the parallelism which VLSI will make available. This is the view emphatically expressed recently by Turner:-

> 'One approach that can be dismissed more or less straight away in this context is the idea that we should take some conventional sequential language, such as FORTRAN or PASCAL, and add some new primitives for launching processes and controlling communication between processes (perhaps along the lines of the tasking facilities of ADA). Such an approach may work well where the number of processes to be controlled is small, but when we are talking about hundreds, thousands or even tens of thousands of processes being controlled in parallel, this cannot possibly be under the conscious control of the programmer. Parallelism on this scale can only arise naturally, from some basic 'asynchronousness' of the language being used and must not depend on any deliberate action on the part of the programmer.' [226]

Fundamentally, most conventional languages are sequential in nature and this again can be traced to the assignment statement. The update operation on the store of the von Neumann machine is time-dependent. This is the reason for Turner's decription of conventional languages as 'completely unsuitable' for representing parallel processes. It is not certain whether a highly parallel computer will possess a store (in the von Neumann sense of global memory) but there is a strong consensus among researchers that if so, then unfettered assignment to its locations will not be permitted (see for example Chamberlin [39]).

1.6 Conclusion: Forty Years of Programming

When von Neumann's designs were first published, the 'difficult' aspects of computing were universally believed to be related to the hardware. After forty years of programming, there is now an equally widespread recognition of the difficulties which are presented by the requirements of software development. The earliest ad-hoc programming efforts in machine languages were quickly dropped in favour of high-level languages and the structured programming school did bring some discipline to the chaos that was software construction. But overwhelmingly the 'high-level' languages have closely reflected the underlying machine architecture. Although structured programming made an important contribution in applying the principles of abstraction and modularity to the problems of software, it has operated within a machine-oriented view of programming which is fundamentally the same as von Neumann's. Thus, Darlington writes recently that

> 'The invention of the first high-level languages, such as FORTRAN, represented a significant advance over the use of machine code and improved programmer productivity tenfold. It is a pity that not many other quantum leaps have been made on the software side. Modern high-level languages do not differ radically from FORTRAN. Structured programming, the white hope of the sixties and seventies, has demonstrably failed to provide the final solution'. [197]

This is the context in which logic programming is emerging as an alternative. Software construction has become beset with severe problems which keep programmer productivity low, which threaten software reliability, and which may altogether frustrate our ability to exploit advances in hardware. As computers begin to be applied to tasks in which the consequences of failure are increasingly serious, so the need for a radically different and much superior approach to software development becomes more urgent.

2 LOGIC PROGRAMMING: FOUNDATIONS, ISSUES AND SYSTEMS

2.1 Foundations and Issues

This section presents the foundations of logic programming in the Horn clause subset of first-order logic and the controlled deduction from Horn clauses through the top-down application of the resolution rule of inference. However, the fact that logic programming is relatively new (and developing vigorously) means that foundations fairly quickly give way to issues: the two main issues identified here being the strategy for controlling deduction and the selection of language extensions. First, a short informal overview of logic programming is provided which may help in forming an early perspective of the subject.

2.1.1 An Overview

A leading architect of logic programming is without doubt Robert Kowalski. It is appropriate then to consider a short summary of his own of the subject:-

> "Logic programming is based upon (but not necessarily restricted to) the interpretation of rules of the form
>
> A if B and C and ....
>
> as procedures
>
> to do A, do B and C and ....
>
> This interpretation is equivalent to 'backwards reasoning' and is a special case of the resolution rule of inference." [40]

The rule

A if B and C and ...

which is quoted here by Kowalski is representative of the Horn clause form of predicate logic. Hence, a logic program basically comprises a set of Horn clauses which are intended to be descriptive of the problem to be solved. Each clause can be understood in its own right as a statement expressing a relationship between objects occuring in the

problem. This is the logical or declarative semantics of logic programming. However, a top-down theorem-proving system can view each clause as a procedure for solving problems: a problem which matches the head of a clause can be decomposed into a set of sub-problems given by the clause body. This is the procedural semantics by which a theorem-prover can in effect become an executor or interpreter of logic programs. These twin interpretations of Horn clauses - the logical and the procedural - yield the dual semantics which are a central feature of logic programming.
Kowalski's rider that logic programming is not restricted to the 'backwards reasoning' procedural interpretation of Horn Clauses is important. At least two directions for extension are clearly apparent. First, Horn clauses form only a small subset of first-order predicate logic, and there is obvious scope for extending the language used whilst still remaining within the accepted confines of logic. Second, 'backwards reasoning' offers one form of procedural interpretation, but other procedural interpretations of logic can be envisaged, such as (for example) those which might be suggested by 'forwards reasoning' and 'middle out reasoning'.
Perhaps the best metaphor for logic programming is that of

Computation = Controlled Deduction

proposed by Hayes [56]. A second metaphor, due to Kowalski [57], is the pseudo-equation

Algorithm = Logic + Control

which suggests the distinction - between what the knowledge is which is required to solve a problem and how the knowledge is to be applied in order to reach the solution - which is fundamental to logic programming and which distinguishes logic programming from conventional imperative programming systems.
Both of the above are frequently quoted in the logic programming literature.

2.1.2 First-Order Logic

First-order logic, which is sometimes referred to as the predicate calculus, is the underlying language of logic programming and an informal outline of its syntax and semantics is appropriate at this point. For a more complete treatment than is presented here, the books by Robinson [48] and Hodges [42] are convenient references.
The formulae of first-order logic are constructed from the following symbols:-

The quantifiers,
∀ ('for all') and ∃ ('there exists').
The propositional connectives,
& ('and'), v ('or'), ¬ ('not'),
-> ('implies'), <- ('is implied by'),
<-> ('is equivalent to').
A set C of constant symbols.
A set V of variable symbols.
A set P of predicate symbols.
A set F of function symbols.

A term is a variable, a constant, or a functional expression of the form

f(t1, t2, ..., tn)

where f is a function symbol and the ti are terms. An atom (or atomic formula) is an expression of the form

p(t1, t2, ..., tn)

where p is a predicate symbol and the ti are terms. A literal is either an atom (A) or a negated atom (¬ A). A formula is either an atom or an expression of the form

X & Y, X v Y, X -> Y, X <- Y, X <-> Y, ¬ X, ∀ yX, ∃yX

where X and Y are formulae and y is a variable. A sentence is a formula which contains no free (unquantified) variables.
The usual semantics of first-order logic are the model-theoretic semantics as presented for example by Tarski [52], in which the meaning of a set of sentences rests on the notions of a universe of discourse and an interpretation. Intuitively, the universe of discourse of a set of sentences is the set of all individuals described by the sentences. An interpretation of a set of sentences can be regarded as a set of assignments of one of the two truth values true and false to each atom obtainable by combining

an n-place predicate symbol with a set of n arguments taken from the universe of discourse. Such an interpretation, together with the axioms of first-order logic, permits the assignment of a truth value to each sentence in the set. A set of sentences which possess an interpretation within which each sentence is assigned the value true is known as a consistent set of sentences; the corresponding interpretation is known as a model for the set.
The syntax of logic has been prone to some variations in different presentations of the language. These syntactical variations have carried over into logic programming also, as will become evident in this thesis. Usually however this does not present a major source of difficulty, and in what follows no fuss will be made of it except where the risk of confusion is sufficient to require that attention be drawn to the forms which are being quoted.

2.1.3 Full clause form and Horn clause form

It can be shown that every set of sentences of first-order logic can be converted to an equivalent (in terms of satisfiability) set of sentences of the form

A1 v A2 v ... v An v (¬B1 v ¬B2 v ... v ¬Bm)

(n, m >= 0), where all variables are taken to be universally quantified and in which each Ai and Bj is an atomic formula. (The book by Nilsson [43] includes a conversion algorithm). Sentences in this special form are known as clauses. A logically equivalent sentence to the clause above is

A1 v A2 v ... v An <- B1 & B2 & ... & Bm

and this variant of clause form, which is sometimes known as Kowalski form, is the one which will be used in what follows.
Clauses (in Kowalski form) which have atoms on both sides of the arrow are called implications. Special cases are clauses of the form

A1 v A2 v ... v An <-

(ie, where m = 0), interpreted as assertions, and

<- B1 & B2 & ... & Bm

(ie, where n = 0), interpreted as denials, and the empty clause

<-

(ie, where n = m = 0), interpreted as false.

Clause form provides a normalised form for first-order logic. Its use reduces the potential redundancy of having equivalent formulae expressible in different ways. This is likely to be of significant value in any system for the automatic processing of logic.
An important subset of clause form is the set of clauses which have at most one conclusion. This subset comprises clauses which are either implications of the form

A <- B1 & B2 & ... & Bn

(n > 0), in which the consequent is often termed the head and the antecedent is termed the body, or else are unconditional assertions of the form

A <-

or else are denials of the form

<- B1 & B2 & ... & Bm

(m >= 0, with the empty clause denoting falsehood). These are known as Horn clauses, named after the logician Alfred Horn who first investigated them [44]. It can be shown that any problem which can be expressed in logic can be re-expressed by means of Horn clauses (although there is often a loss both of economy and of naturalness of expression: see, for example, Kowalski [45, pp16, 193-206]). In addition to possessing this completeness property, Horn clauses have other attributes which make them attractive from a computational perspective. These include their extremely simple syntax; the fact that resolution (q.v.) has a particularly straightforward interpretation for Horn clauses; and their resemblence to conventional procedures (having a head and a body) in an Algol-type language.

2. 1. 4 Resolution

The problem of showing that a set of clauses is satisfiable, or otherwise, by searching arbitrary universes for a model for the set appears rather daunting. Fortunately Herbrand showed that the only universe which need be considered is that comprising the set of variable-free terms which can be formed from the constant and function symbols which appear within the clauses (the so-called Herbrand Universe). A proof is contained in Bundy [49]. In effect, this makes it possible to investigate the consistency of clauses 'syntactically', by constructing a proof consisting of inference steps. Surprisingly perhaps, it turns out that only one inference rule, Robinson's resolution rule [41], is ever required for each step.

The resolution rule is an extremely powerful inference rule for first-order logic. It can be regarded as a generalisation of the rule known as modus ponens in propositional logic:-

> From the truth of P
> and the truth of P -> Q
> assert the truth of Q

In the following we present first the full clausal form of the resolution rule and then its restriction to Horn clauses. Many sources are available which provide much fuller accounts of resolution, including the books by Robinson [48], Kowalski [45], and Bundy [49]. Later we will turn to consider resolution-based proof procedures.

2.1.4.1 The full clausal form of Resolution

Given two clauses

```
C1:  P1 v P2 v ... v Pi  <-  p1 & p2 & ... & pj
C2:  Q1 v Q2 v ... v Qk  <-  q1 & q2 & ... & ql
```

where we may assume in general that the two clauses have differently named variables, the resolution rule states that if some most general substitution S for variables can make some Ps identical to some qt (1 <= s <= i, 1 <= t <= l), written as

```
[Ps]S = [qt]S
```

then the clause

```
C3:  [P1 v P2 v ... v Ps-1
        v Q1 v ... v Qk
        v Ps+1 v ... v Pi
           <- q1 & ... & qt-1
                 & p1 & ... & pj
                 & qt+1 & ... & ql]S
```

can be derived. The clause C3 is known as the resolved clause (or resolvent) and the clauses C1 and C2 are known as the parent clauses.

It is an important feature of resolution that given two clauses, any atom in the conditions of either clause is a potential candidate for being matched with any atom in the conclusions of the other. In general, where two atoms can be matched there will be more than one substitution of terms for variables available, but one of these substitutions is more general than all the others (it yields

an atom of which the other substitutions only give instances). For example, the two atoms

Pred(x1, 4, x2), Pred(Con, y1, y2)

(in which 4 and Con are constants and the other arguments are variables) have the following matching substitution:-

{x1 = Con, y1 = 4, x2 = 3, y2 = 3}

but the most general substitution is

{x1 = Con, y1 = 4, x2 = y2}

since this substitution makes the least specific assignment of terms to variables. The process of finding this most general substitution, or unifier, is called unification. It is generally accepted that the successes of resolution in automatic deduction systems are substantially due to its use of unification. Robinson [62] points out that earlier deduction systems, such as those of Gilmore [60] and Wang [61], attempted exhaustive substitution of terms for variables and were effectively swamped by the numbers of possibilities. By delaying the substitution of terms for variables for as long as possible unification effectively prevents many worthless specific replacements from being tried. An account of the advantages of unification relative to the Gilmore procedure is contained in Bundy [49].
Robinson (op. cit.) attributes the discovery of the significance of unification to Herbrand; however it was apparently overlooked until independent re-discovery in 1960 by Prawitz [63] and its incorporation by Robinson into the resolution principle.

2.1.4.2 Resolution applied to Horn clauses

Resolution has a particularly simple form when applied to Horn clauses. Given two Horn clauses

```
C1:  P <- p1 & p2 & ... & pj
C2:  Q <- q1 & q2 & ... & qk
```

in which some most general substitution S matches P with (say) qt (1<=t<=k), the resolvent is the clause

```
C3:  [Q <- q1 & ... & qt-1
         & p1 & ... & pj
         & qt+1 & ... & qk]S
```

A special case of Horn clause resolution is that which Kowalski [45] calls top-down resolution. From parent Horn clauses having the form

```
C1:  P <- p1 & p2 & ... & pj
C2:    <- q1 & q2 & ... & qk
```

where some most general substitution S can match P with qt (1<=t<=k), the resolvent is

```
C3:   [ <- q1 & q2 & ...& qt-1
          & p1 & ... & pj
          & qt+1 & ... & qk]S
```

Top-down resolution resolves a denial with an implication to obtain a new denial. It can be regarded as a generalisation of the classical modus tollens rule of propositional logic. Furthermore, top-down resolution can form the basis of refutation (or proof by contradiction) procedures, in which the required conclusion is first denied and a contradiction is then established. Resolution of the initial denial with an implication leads to further denials, with the derivation of the empty denial denoting contradiction. Kowalski describes this as 'backwards reasoning' and he identifies it with the analytic process of decomposing problems into sub-problems.
Another important special case of Horn clause resolution is that which Kowalski calls bottom-up resolution. From parent Horn clauses having the form

```
C1:  P <-
C2:  Q <- q1 & q2 & ... & qk
```

where some most general substitution S can match P with qt (1<=t<=k), the resolvent is

```
C3:   [ Q <- q1 & q2 & ...& qt-1 & qt+1 & ... & qk]S
```

Bottom-up resolution resolves an assertion with an implication to obtain a new implication. As a special case, where qt is the only condition in the body of the second parent clause, an assertion is resolved with an implication to generate a new assertion. It can be regarded as a generalisation of the classical modus ponens inference rule of propositional logic. Furthermore, bottom-up resolution can form the basis of direct proof procedures, in which the given assumptions are used to infer new conclusions with the aim of eventually deriving the required result. Kowalski describes this as 'forwards reasoning' and he identifies it with the synthetic process of combining old information into new.

2.1.4.3 Properties of Resolution

Two important properties of the resolution rule of inference are its soundness and its completeness. The soundness property ensures that the rule is correct, in the sense that any clause derived from a self-consistent set of clauses through some sequence of resolution inference steps will necessarily be a logical consequence of the given clauses. The completeness property (sometimes called the refutation completeness) guarantees that from any given set of inconsistent clauses, the empty clause (denoting contradiction) will be derivable by some finite sequence of resolution steps. Proofs of both properties are given by Bundy [49]; the completeness proof essentially depends upon a theorem which is due to Herbrand [53].
It is a significant theoretical limitation that resolution, along with all other inference rules, is affected by the Incompleteness Theorem of Godel [54]. Godel's result implies that no purely mechanical procedure can be guaranteed to correctly determine the validity of an arbitrary sentence of logic. It follows that a proof system which exploits the refutation completeness of resolution cannot be assured to terminate in cases where no refutation exists (because the clauses are satisfiable). A good informal account of the incompleteness problem is to be found in Sheperdson [55].

2.1.5 Proof Procedures as Logic Interpreters

A proof procedure P for clausal logic is a systematic method of applying a set of rules of inference to a set L of clauses in an attempt to reach a required conclusion. A derivation of a clause Cn from L using P is a sequence of clauses

C1, C2, C3, ... Cn

such that C1 belongs to L, and each clause in the sequence is derived from its predecessor by applying inference according to P.
Kowalski [45] notes that all existing proof procedures for clausal logic (and so by subsumption, for Horn Clause logic) are refutation procedures. These procedures seek to show that the denial of the required conclusion is inconsistent with the given clauses (the input set) by forming their union and deriving from it the empty clause. Assuming that the input clauses are consistent in themselves, this is

equivalent to showing that the required conclusion is a logical consequence of the input clauses. At first this may look like convoluted thinking, but in fact justification lies in the refutation completeness of the resolution inference rule as described earlier. This completeness ensures that, starting with

a set L of input Horn clauses, plus
a denial G, expressing the problem to be solved,

there does exist (at least) one derivation of the empty clause using the resolution rule whenever G is inconsistent with L. Derivations which end in the empty clause are known as successful derivations. If the goal clause contained variables, then 'answers' to the problem may be extracted by tracing the substitutions which have been made during a successful derivation. This is known as answer extraction.
In the context of logic programming, proof procedures are wholly or partly automated and are known as logic interpreters or program executors. The pairing of a set L of input Horn clauses with a denial G is known as a logic program, and G is known as the goal clause of the program. That is,

Logic Program = {Input Set of Horn Clauses L}
+ Goal Clause G

The submission of the logic program to the logic interpreter is progam execution, and each derivation of a sequence of clauses which results is called a computation. A computation is successful or unsuccessful depending on whether or not the empty clause is derived; it is terminating or non-terminating depending on whether or not the computation is finite.

The problem of finding effective program executors is a central problem of logic programming which is often referred to as the control problem. In theory, an investigation of the entire search space comprising all possible derivations using resolution from the input clauses and the goal clause will always find a successful computation if one exists, but in practice this search space can be very large or even infinite. Fortunately, it is also typically highly redundant, containing many duplicated computations. Many researchers, including Kowalski [45], Kowalski and Kuehner [56], Robinson [41], [47], Loveland [57], and Siekmmann and Stephen [58], have investigated the problem of how to specify resolution proof procedures which are sound (or correct) in the sense that successful computations actually are logical refutations, complete (in the sense that the empty clause is computed whenever a refutation exists), and efficient in the sense that redundant searching is eliminated as far as possible.
It should be stressed at this point that a logic program is independent of the logic interpreter which is selected to

execute it. Logic programs express the domain-specific information content of problems, whereas the interpreter contributes control whereby the solutions are deduced from the logic. In illustration, it can be observed that given a fixed logic program, it is possible to substitute the application of any one correct and complete proof procedure for any other; the only possible effect will be to alter the efficiency by which solutions are found. This separation of logic from control contrasts with traditional imperative computing, in which programs (or algorithms) mix them together. Kowalski [45] expresses this symbolically as

Algorithm = Logic + Control

Many of the advantages claimed for logic programming are rooted in its separation of logic from control, and these will be discussed later. However, it is worth noting here Kowalski's observation that

> "The control component can be expressed by the programmer in a separate control language; or it can be determined by the program executor itself. ... A completely satisfactory, autonomous control strategy ... has not yet been designed". [45]

Hence, in parallel with the work to develop better automatic proof procedures, researchers have sought to provide control facilities for the use of programmers. More will be said about this later.
To date, it seems that little use has been made of procedures based on bottom-up inference. (An exception is the hyper-resolution system of Sickel [64]). The main problem with such systems is that they are difficult to motivate correctly. Kowalski [45] observes of bottom-up procedures that

> "... they generally lead to combinatorially explosive behaviour, generating assertions which follow from the general description of the problem-domain, in addition to assertions which follow from the assumptions of the particular problem in hand."

However, a Horn clause theorem-prover based on bottom-up resolution combined with top-down resolution has been described by Kuehner [57]. In the field of automatic theorem proving, an example of a (non-resolution) system which exploits inference in both directions has been constructed

by Bledsoe [58]. Kowalski has described a procedure, the connection graph proof procedure, which permits both bottom-up and top-down resolution to be mixed [45]. It is clear, however, that most of the effort of logic programming researchers to date has been concentrated on proof procedures which are limited to the top-down application of resolution. Hence, top-down resolution will be the focus of our attention in the next section.

2.1.6 Top-down Resolution Procedures

Given a logic program L, comprising a set of Horn clauses together with a denial G, a goal clause representing the problem to be solved, a top-down resolution procedure is a systematic method of applying top-down resolution inference in an attempt to derive the empty clause. Recalling that this inference step always resolves a denial with an implication to generate a new denial, it will be clear that a top-down derivation is a chain of denials

G, G', G'', ...

where each succeeding denial is the result of resolving its predecessor with an implication (or assertion) of L. (It is significant that it is not possible for denials in L to participate in such a derivation; this is part of another important problem of logic programming, the so-called negation problem, which will be discussed later.) As before, derivations are referred to as computations in the procedural terminology. A successful computation is one which generates the empty clause.
Kowalski [45] regards the top-down procedural interpretation of Horn clauses as providing their fundamental problem-solving interpretation. In a top-down computation

G, G', G'', ...

the body of each denial is (in general) a conjunction of atoms, say

<- g1 & g2 & ... & gn

and is viewed as a goal statement, representing a conjunction of goals or problems to be solved. If a goal statement includes variables x1, x2, ... , xk then the problem which is specified by it is logically expressed as

Find x1, x2, ... , xk
which solve the problems g1 and g2 and ... and gn.

The implications of L are regarded as procedures which can

potentially solve those goals which can be matched (unified) with the procedure heads. A procedure which matches a goal transforms it into the collection of sub-goals given by the atoms of the procedure body with the matching substitutions applied. Thus a goal can be interpreted as a procedure call and a procedure responding to the call does so by generating the set of sub-calls given by the atoms of the procedure body. In general, substitutions (sometimes called instantiations or bindings) will be made both for the variables of the procedure (regarded as procedure input) and for those of the goal (regarded as procedure output). Assertions are regarded as procedures which are capable of solving problems directly, without requiring that further sub-problems be solved.
Two distinct types of decision-making are associated with top-down resolution proof procedures. To illustrate this, consider an arbitrary point at which a top-down computation has arrived at a denial (say)

```
<- g1 & g2 & ... & gm
```

This denial is one parent clause in the next resolution step. To proceed to the next step in the computation, it is necessary to make two kinds of selection:-

(1) A goal gk (1<=k<=m) must be selected from among the m goals of the denial to be the atom which will be an attempted match for the head of one of the input clauses.

(2) In general, there will be several clauses among L which have heads which are unifiable with gk. One of these clauses must be selected to be the other parent in the resolution.

The strategy by which a given top-down proof procedure makes decisions of type (1) is known as its computation strategy, whilst that by which type (2) decisions are made is the search strategy.
The soundness of a top-down resolution procedure is assured, irrespective of its computation and search strategy. But this fact, which follows from the soundness property of resolution as described earlier, only ensures that a logic interpreter based on top-down resolution will not generate answers which are actually incorrect. Clearly it is also highly desirable that an interpreter shall positively discover the complete set of answers , and shall do so as efficiently as possible. In these respects the computation and search strategies are of crucial importance, albeit in different ways, and much effort has been expended into researching various alternatives.
The scope for experiment with the computation and search

strategies of top-down procedures arises as a consequence of the inherent non-determinism of logic programs. As has been pointed out earlier, a logic program expresses the information content of problems but it does not dictate how the information may be used. In particular, logic does not determine the order in which the conditions of implications should be explored, and neither does it determine the order in which alternative clauses for the same relation should be investigated. Indeed, the investigations of both types could in principle proceed in parallel fashion. These forms of non-determinism are known as the and form of non-determinism and the or form of non-determinism respectively. The scope for varying the computation and search strategies is the corresponding proof procedure (control) counterpart of these (logical) non-determinisms. The prospects for actually realising the two potential forms of parallelism, known respectively as and-parallelism and or-parallelism, in practice will be considered later.
Fixing upon a particular pair of computation and search strategies is equivalent to fixing the control component of the

Algorithm = Logic + Control

relationship. A logic program paired with a fixed computation and search strategy then becomes a deterministic algorithm. (A slightly differing perspective on this is suggested by Clark, McKeeman and Sickel [59], who suggest that a logic program with control unspecified can be regarded as a non-deterministic algorithm which is the family of all the deterministic algorithms that can be obtained by adding specific computation and search strategy control). As the pseudo-equation suggests, the same algorithm could result from different combinations of logic and control, and in particular the efficiency of a given execution might be modified either by keeping the logic fixed and altering the control or vice-versa. In this section however it is the control of logic programs which will be explored.

2.1.6.1 Top-down search trees

Given a logic program (L, G), where L is a set of Horn clauses and G is a goal clause, together with a control strategy C comprising a computation strategy and a search strategy, the top-down search tree for the triplet (L, G, C) is the unordered tree with G at the root and where each branch from the root represents a top-down computation from G. Each node is labelled with a denial, and has sons which are the denials obtainable by resolving the denial at the father node with one of the implications (or assertions) of

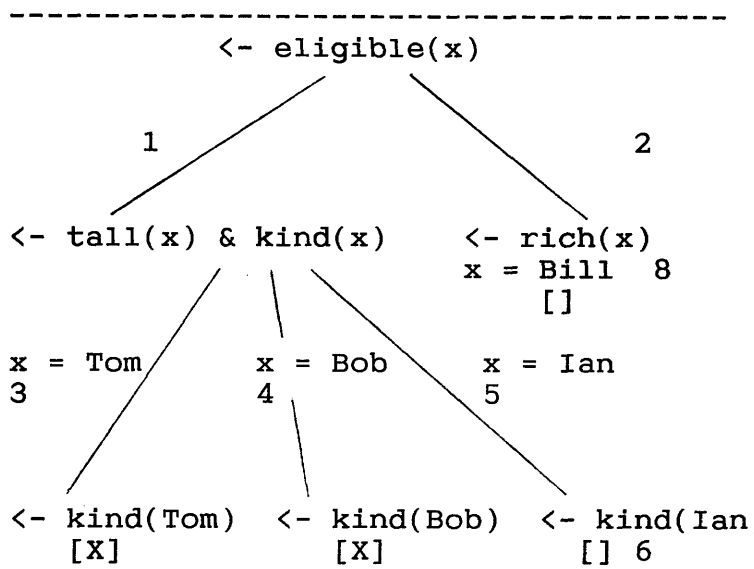
L, selecting the atom for resolution from the denial according to the computation strategy component of C. Where a branch ends with the empty clause the tip is marked '[]', indicating a successful computation; where a branch ends with a denial which cannot be resolved further with a clause of L, the tip is marked '[X]', indicating an unsuccesful computation. The arcs are labelled with substitutions and are indexed to show the clause of L which has been involved in the resolution.

The size and shape of the search tree depends wholly on the logic program and on the computation rule component of the control strategy C, as the example following will illustrate. It is helpful to view the task of a logic interpreter based on C as being a search (although of course it is really a construction) of the tree, guided by the search rule component of C, in order to find branches which end in '[]'.

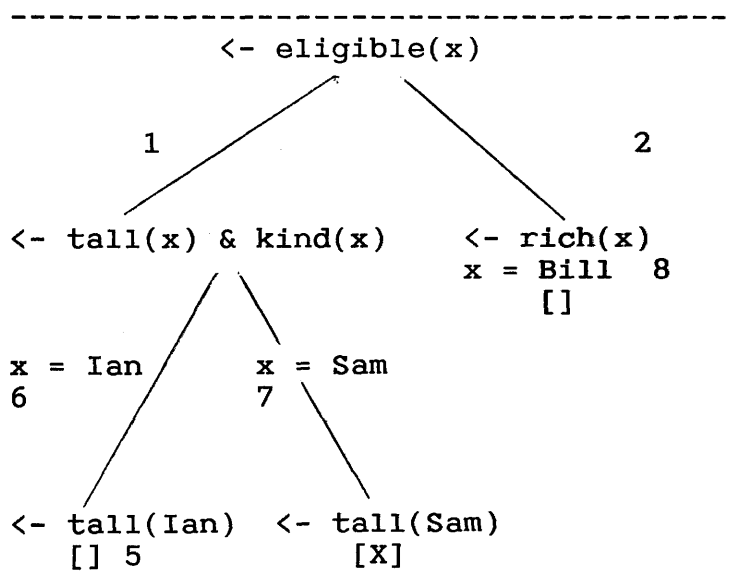
To illustrate, consider this logic program:-

```
---------------------------------------
Input clauses L

1   eligible(x) <- tall(x) & kind(x)
2   eligible(x) <- rich(x)
3   tall(Tom)   <-
4   tall(Bob)   <-
5   tall(Ian)   <-
6   kind(Ian)   <-
7   kind(Sam)   <-
8   rich(Bill)  <-

Goal clause G

    <- eligible(x)
---------------------------------------
```

Here is the top-down search tree obtained by a computation rule which selects goals in a last-in, first-out left-to-right order:-

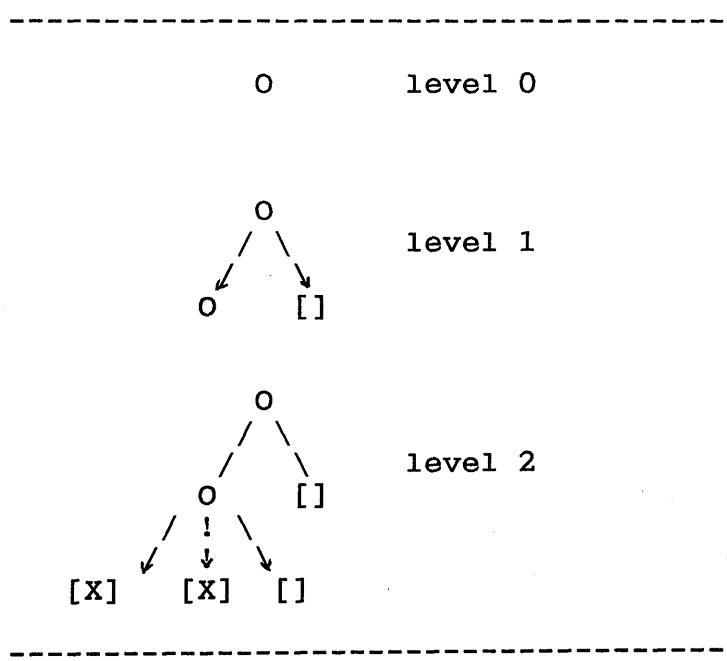


Another search tree for the same logic program, but this time with deduction controlled by a last-in, first-out right-to-left computation rule, is:-



Both search trees contain successful computations from which can be extracted the complete set of solutions x = Ian and x = Bill, but the trees are different. The tree corresponding to the last-in, first-out left-to-right computation rule is here the larger of the two, producing as it does an additional unsuccessful computation. On the assumption that the search rule in both cases specifies an exhaustive search of the trees, a logic interpreter based on the first control strategy will reasonably be judged to execute this program less efficiently. As Kowalski [45] points out, it is desirable in general to minimise the size of the tree to be searched: this suggests either keeping the same control, and re-formulating the logic with the aim of reducing the size of the tree; or else fixing on the same logic program, and improving the control strategy of the logic interpreter. The latter is an important long-term goal for logic programming which is further discussed below.

In contrast to the computation rule, the search rule does not influence the size and shape of the tree. It only determines the manner in which the tree is searched (or rather, more correctly, constructed). The two main kinds of search are breadth-first and depth-first searches. A breadth-first search investigates the branches of the tree evenly starting downwards from the root; all the nodes at level n are explored before investigating any node at level (n+1). Thus the first tree depicted above will be searched as follows:

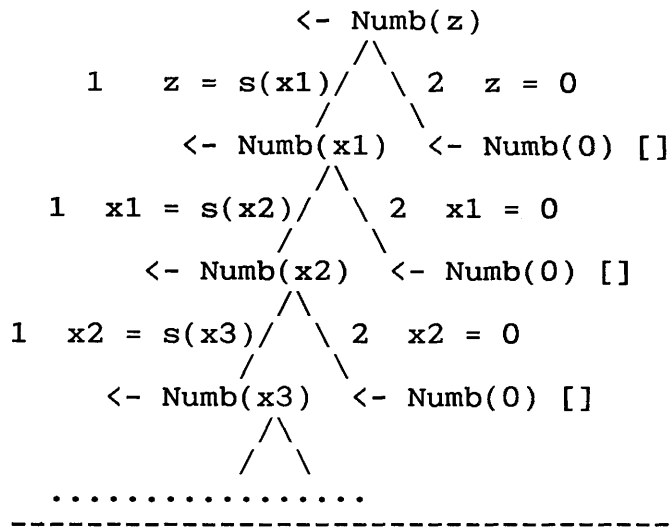


The development at each level in a breadth-first search could be a genuinely parallel one. If it is only quasi-parallel, however, then the search rule must specify the order in which to explore the nodes at a given level. A depth-first search on the other hand develops the branches one at a time; when a tip is reached, it backtracks to the closest ancestor node which has unexplored sons and selects, according to some selection strategy, one of the corresponding branches to continue the search. An obvious selection strategy (although not necessarily the best in every case) is to select matching input clauses in the order in which they are written. Coupled with a depth-first type search, this gives a depth-first, top-to-bottom search rule. Given a fixed and finite search tree which is to be exhaustively searched, the only possible difference between logic interpreters operating different search strategies will be the differing orders in which solutions may be found. Sometimes however it may not be necessary, or desirable, to search the entire tree, since it may include branches which are infinite or which are of no interest. As before, this is a problem which can be tackled either by changing the logic or changing the control. The logic can be changed so as to specify only those computations which are of interest; alternatively the control can be modified so that the search rule prevents exploration of the unwanted branches. The question of how the latter can be accomplished is discussed later.

An important theoretical result, attributable to Hill [76] among others, is that all top-down resolution inference procedures are complete providing they exhaustively investigate the entire top-down search space (that is, eventually every node must be explored). Thus, with the proviso of exhaustive search, completeness is independent of the computation strategy. Kowalski [45] has pointed out that procedures which employ depth-first search may lose completeness if the execution 'falls down' an infinite branch before all the solutions have been found. Thus depth-first search strategies are 'unfair' in that they are liable to devote an infinite amount of processor time to developing one computation whilst ignoring others, and in these instances the search is not exhaustive. He gives a simple example, in which the input clauses comprise a definition of the natural numbers and the goal is to find a natural number:-

```
-------------------------------------
Input clauses L

1   Numb(s(x)) <- Numb(x)
2   Numb(0)    <-

Goal clause G

    <- Numb(z)
-------------------------------------
```

(s here denotes the usual successor function over the natural numbers). The top-down search tree (which in this example does not depend on the choice of a computation rule, since conjunctions of goals never appear) begins like this:-



The tree contains an infinity of finite branches giving successful computations from which the respective substitutions

z = 0, z = 1, z = 2, ...

can be extracted. However, a logic interpreter which follows a depth-first top-to-bottom search rule will not find any of them. The reason is that execution will descend fruitlessly down the one infinite branch. One remedy here would be to change the control strategy, say to breadth-first or to depth-first bottom-to-top; another would be to re-order the clauses of the logic program and to keep the control strategy as it stands.

2.1.6.2 The Computation Strategy

A computation strategy is of the last-in first-out kind if the atom selected from the current denial for matching with one of the input clauses is always one of the atoms most recently introduced into the denial. For example, if the current denial is

```
<- A1 & ... & An
```

then the selected atom will be some atom Ai which was introduced in the last resolution step. If Ai unifies with the head B of some procedure

```
B <- B1 & ... & Bm
```

through some substitution S, then the new goal statement becomes

```
[ <- A1 & ... & Ai-1 & B1 & ... & Bm & Ai+1 & ... & An ]S
```

and the next selected atom will be one of the B1, .. ,Bm.
Most top-down resolution systems have employed a last-in first-out computation rule. These systems include ordered linear resolution [65], SL-resolution [56], inter-connectivity graph resolution [66], analytic resolution [67], and SLD-resolution. The last of these is the basis of the PROLOG family of logic programming systems [69] which utilises a last-in first-out computation rule with a strict left-to-right ordering for the selection of atoms (so that A1 and B1 respectively would be selected in the example above). Unsurprisingly then, some writers (such as Hogger [70]) now refer to the last-in first-out left-to-right computation rule as the standard computation rule in the logic programming context.
The advantages of the standard computation rule for logic programming are apparent. It is simple for programmers to understand, and arguably therefore makes debugging less difficult than a more complex strategy. It has proved fairly straightforward and efficient to implement (see later). Furthermore, with the standard rule the execution ordering of the calls within a procedure body is similar to that of conventional programming languages; for example, the procedure

```
procedure(...) if
  firstsubprocedure(...) &
  secondsubprocedure(...) &
  thirdsubprocedure(...)
```

when called will invoke the three subprocedures in the expected sequence (although a depth-first search rule will generally complicate this with backtracking behaviour). More

consideration of the algorithmic behaviour of logic programs will be given in a later section.
The major weakness of the standard computation rule stems from its uniformity. In always selecting the atom A1 from the current goal statement

<- A1 & ... & An

it fails to take account of the opportunities which may be present in the given specific problem to facilitate the computation. A study of such opportunities has been undertaken by Kowalski [45], who identifies the following factors.

1. Independent sub-goals could be distributed to independent problem-solvers without any danger of interfering with one another. Thus, if two or more of A1, .. , An have no variables in common then an excellent opportunity for parallelism (or quasi-parallelism) is present. Even if the sub-goals are dependent, parallel problem-solvers can still be employed; for example, a solution to the conjunction could be found by finding the most general common instance of the substitutions which satisfy each individual sub-goal (Conjunctive parallelism will be considered further in a later section).

2. Where sub-goals are dependent (that is, have variables in common), Kowalski suggests the general principle of selecting the sub-goal to which the fewest procedures apply: the aim is to reduce the overall size of the search tree by minimising the number of branches which descend from each node. The principle can be viewed either as a 'principle of procrastination', which delays the selection of explosive sub-goals for as long as possible, or else as a 'principle of eager consideration', which favours sub-goals which can be solved in few ways. In particular, a sub-goal which can be solved in at most one way should be eagerly considered; it needs to be evaluated eventually anyway, but should it turn out to fail (be insoluble) then early discovery of the fact will enable the whole goal statement to be failed without the need for further computation.
It is recognised that the number of procedures which apply to a sub-goal only gives a 'local' measure of explosiveness. Look-ahead techniques, such as the mini-max strategy described in Nilsson [71] and elsewhere, could provide more accurate measures.

3. A related principle to the above, and one which Kowalski suggests should be easier to apply, is to select a sub-goal which involves the least 'finding' and the most 'showing' of relationships: a severe 'finding' sub-goal is one which has many uninstantiated variables, wheras a wholly 'showing' sub-goal has none. This principle should favour sub-goals which contain input (instantiated variables). It is

generally highly inefficient to execute sub-goals which contain no input; an example is the program:-

```
Sort(x y) <-  Ord(y) & Perm(x y)
<- Sort((2 4 1 5 3) y)
```

where Sort(x y) holds when y is a sorted version of the list x, Ord(y) holds when y is an ordered list and Perm(x y) holds when y is a permutation of the list x. Here, the goal asks for a sorted version of the list (2 4 1 5 3). If the sub-goal Ord(y) containing no input is selected first, it will generate an arbitrary ordered list which Perm((2 4 1 5 3) y) will test as a possible (if highly improbable) permutation of the input list. A computation strategy which schedules sub-goals first to minimise 'finding' should select the atoms in the reverse order, resulting in more satisfactory behaviour.

4. Lemma generation can be utilised. When a sub-goal is solved, its solution is recorded. The benefit is gained when the same sub-goal arises more than once during the program execution. If the occurences are on different branches of the search tree then only a single computation step is required to solve it after the first solution. If the occurences are on the same branch, and the second occurence has been re-introduced by means of a procedure, then an infinite branch is indicated: a 'loop' has been detected and action can be taken accordingly. 'Negative' lemmas, which record that a goal has been found insoluble, can also be employed.

5. Some logic programs, which cannot be executed with acceptable efficiency with any simple sequential computation rule, respond well to co-routining. Kowalski quotes the example of the admissible pairs problem, in which the goal is to construct a pair of lists such that for all i, the i-th element in the second list is twice the i-th element of the first list, and the (i+1)-th element in the first list is thrice the i-th element in the second list. The first list is to begin with one. Thus, an admissible pair of lists is:-

```
(1  6  36)  and
(2 12  72)
```

The top clause in Kowalski's program is:-

```
Adm(x y) <- Double(x y) & Triple(x y)
```

Under a computation rule which executes Double to the finish before initiating Triple, or vice-versa, the program is intolerably non-deterministic: it repeatedly generates an arbitrary pair of lists which lie in the Double (Triple) relation and then checks whether this pair also lie in the

Triple (Double) relation. But when the two calls are allowed to behave as co-operating sequential processes, with control switching from one to another as soon as sufficient input is available, the program begins to generate admissible pairs in a highly efficient manner.

It is clear from the above that there is considerable scope for constructing a logic interpreter which implements a computation strategy considerably more sophisticated than the standard computation rule. However, Kowalski notes that the problem of describing an efficient algorithm for scheduling procedure calls has still to be solved, and that the principles of procrastination and eager consideration mentioned above '... work efficiently in a large number of cases. But they are inadequate when all procedure calls are non-deterministic'. [45] The quest for an improved autonomous computation strategy must be an important goal of future research.

In the absence of an effective automatic computation strategy, much attention has been concentrated on the provision of mechanisms which enable the programmer to specify the scheduling of procedure calls directly. Gallaire and Lasserre [72] have identified various such mechanisms including those of pragmatic control, control annotations and metarules. Since these mechanisms have been aimed at the control of top-down resolution in general - that is, at the search strategy component as well as at the computation strategy component of control - they will be discussed together in a later section. However, a brief consideration here will be appropriate.

Gallaire and Lasserre say that pragmatic control 'consists of writing a program tailored to the fixed strategy of the interpreter'. By illustration, a PROLOG programmer who knows that his logic interpreter incorporates the standard last-in first-out left-to-right computation rule can arrange the ordering of atoms within the body of each procedure so as to obtain the desired algorithmic effect. Most PROLOGs, including Warren's DEC10-PROLOG [73], also offer special predicates (the so-called metalevel primitives) which can assist with this effort. One example is the var primitive, where var(x) succeeds if x is an unbound variable. Another is the assertz primitive, which takes a clause as its argument and which succeeds by appending the clause to the logic program, therebye offering a facility which can be used for lemma generation as discussed above.

Control annotations are annotations attached directly to the program text to give control information to the interpreter. A large set of such annotations are provided by the IC-PROLOG system developed at Imperial College [74]. One example is the '//' annotation which can substitute for '&' in the body of a procedure. IC-PROLOG's default computation rule is the standard one, but the use of '//' overides the standard rule in favour of a timesharing parallel evaluation. This and other control annotations of IC-PROLOG

are further described by Clark and McCabe [75].
The use of metarules provides a quite different form of programmer specified control. Metarules are special rules, syntactically separated from the logic program, which give control information to the interpreter. An example of a metarule proposal is that of Gallaire and Lasserre [72] whose metarules take the form of Horn clauses. Again, the default computation rule is the standard one, but it may be overridden by a metarule specifying special conditions for scheduling procedure calls. An example is

READY(P(x, y, z)) - INST(x)

which says that the atom P(x, y, z) should be selected for evaluation as soon as the variable x becomes instantiated.

2.1.6.3 The Search Strategy

As stated earlier, the size and shape of the top-down search tree is determined by the logic program and by the computation rule component of the control strategy. The search strategy determines the way in which the tree is searched (or more precisely, constructed). Hill [76] has shown that every top-down resolution inference system is complete, regardless of the computation rule, provided that the search tree is exhaustively explored. Hence the completeness of a logic programming system depends on its search rule, which must be guaranteed to eventually select each one of the alternative procedures which match the atom chosen for elimination from the current denial.
Of the two main types of search strategy, namely the depth-first and the breadth-first search strategies, it has already been pointed out that the depth-first strategy risks losing completeness where computation may 'fall down' an infinite branch of the search tree before successful computations have been discovered. This is a major weakness of depth-first systems. A breadth-first strategy could also lose completeness in cases where a node of the search tree has an infinite number of sons (corresponding to a denial resolving with an infinite number of input clauses - say, arithmetic assertions); it is not clear how serious a limitation this might be in practice. The PROLOG family of logic programming systems [69] incorporate a depth-first search rule with the reselection of clauses on backtracking determined on a top-to-bottom basis (that is, alternative clauses are tried in the order in which they are written); this is often referred to as the standard search rule. An example of a logic programming system using a breadth-first search strategy is the LOGLISP system [70].
The standard search rule, like the standard computation rule, is relatively efficient and cheap to implement (see

later). However, in addition to the risk mentioned above of losing completeness, this strategy has at least two other weaknesses. First, as experience with earlier programming systems which incorporated backtracking has shown, a backtracking behaviour can be difficult to predict and debug. Dowson [77], in a retrospective account of the now defunct Artificial Intelligence backtracking language Micro-Planner [78], writes that

> '... backtracking makes programs almost impossible to debug even with (as was the case with Micro-Planner) the availability of powerful tracing, breakpointing and single-stepping mechanisms'.

However, Dowson concedes that backtracking was a strength as well as a weakness of Micro-Planner, and that the language also suffered from other weaknesses. A second problem with a backtracking search strategy is identified by Kowalski [45]:-

> 'Although backtracking is effective in many cases it can be distressingly unintelligent in others.'

The basic problem is that (naive) backtracking systems learn nothing from their failures. Returning after failure to the most recent ancestor node with unexplored sons and exhaustively searching the remaining sub-space is redundant if the actual cause of the failure lies with a unification higher in the search tree. Redundant behaviour of this type was also a feature of Micro-Planner, of which Dowson (op. cit.) writes:-

> 'Micro-Planner provides no convenient mechanisms for passing back, after a failure, information on the cause of the failure which can be used to guide subsequent exploration of the problem space'.

Several proposals aimed at improving the 'intelligence' of the depth-first strategy in the context of logic programming have been advanced, including those of Bruynooghe and Pereira [79], Cox [80], and Pereira and Porto [81]. These proposals mainly follow Kowalski's suggestion that when an unsolvable sub-goal is generated, the program executor should analyse the substitutions which caused the failure and backtracking should select an ancestor node which can actually undo them (which will not always be the nearest ancestor node) [45]. This should imply no loss of completeness,

since a backtracking strategy operating in this way in effect only declines to search parts of the search tree which have already been shown not to contain successful computations. A small price would be the increased complexity in the behaviour of an interpreter which incorporated such a scheme; but more serious may be the cost of implementation. The only known quantified costs for a scheme of this kind are those reported by Bruynooghe and Pereira (op. cit.) who adapted an existing PROLOG interpreter (written in the language 'C') to support a simplified form of their intelligent backtracking proposal. Their results include execution times which are up to 99.7% faster for the modified than for the standard interpreter, and they conclude that 'implementation of intelligent backtracking at a low level is worthwhile'. Unfortunately, their results also include examples where the modified interpreter was slower than the standard version by up to 119% (although it appears significant that the gains were largely among the examples of 'pure' logic progams whereas the losses mainly correspond to pragmatically constructed programs written with the interpreter's control strategy in mind). It is interesting that both Cox (op. cit.) and Pereira and Porto (op. cit.), in proposing their own intelligent backtracking schemes suggest that it is reasonable nonetheless that the programmer should be prepared to give explicit advice about backtracking to the interpreter in some cases.

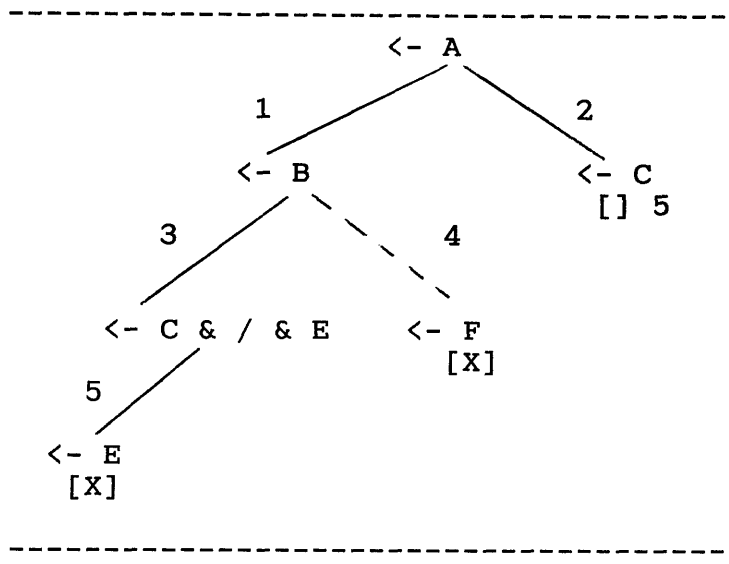
The need to give programmers some control over the extent of backtracking within depth-first programming systems has been recognised before. Kowalski [45] suggests that the inefficiencies of backtracking in the PLANNER family led to the development of CONNIVER [82], in which the programmer has more control over the search strategy. Davies [83], reporting the development from PLANNER of POPLER [84], states that POPLER integrated 'greater and more sensitive control' with the PLANNER backtracking approach. The string-processing language SNOBOL4 [85], which employs a depth-first search algorithm for pattern-matching, provides facilities for the programmer to specify the extent of the backtracking. However, in the case of logic programming systems at least there are differing ideas about the best mechanisms for providing this control. The most frequently discussed mechanisms can be distinguished as pragmatic control, control primitives, control annotations and metarules. These mechanisms, which are relevant to the control of search generally and not just to depth-first control, will be discussed in a later section devoted to a general discussion of the control of logic programs. However, a brief explanation here is appropriate.

Pragmatic search control consists of tailoring the logic program to the known search strategy of the logic interpreter. In the case of an interpreter which operates the standard strategy, such as a PROLOG interpreter, the

depth-first top-to-bottom rule will always investigate the textually higher clauses for a relation before the textually lower ones. Programmers can therefore select a judicious ordering of their clauses to ensure that successful branches of the search tree are discovered (constructed) first; if there is also a facility to terminate the search at the user's choice after each successful computation has been discovered (as there is with most PROLOG systems), then it can be argued that this may be all that is required.

Control primitives appear syntactically as atoms within clauses. The best known example is the 'slash' or 'cut' primitive which is a common feature of most PROLOG systems and which is used to control the extent of backtracking. The cut may appear in the body of a procedure, as with any other atom. Hence a procedure call may introduce a cut into a goal statement, corresponding to some node in the search tree. When the point is subsequently reached where the cut is selected for evaluation, it 'succeeds' but with the side effect of 'pruning' from the search tree all unexplored sub-trees between that point and the father of the node which introduced the cut. An example of a (contrived) logic program and its search tree, constructed with the standard computation rule, is shown below with a dashed line to indicate the search path produced by the standard search rule. The effect of the cut, shown in the program as the symbol '/', is to prevent the interpreter from exploring the branch which ends in the goal <- F (as shown by the dotted path). In this example, nothing is lost because the excised branch contains no solutions: plainly however, the cut also has the potential to prune successful computations from the tree. Hence, the use of the cut introduces another possible risk of losing completeness. The benefit which is hoped for is the greater efficiency which comes from eliminating redundant search, but to be effective the programmer must apply it skillfully and with sound knowledge of the interpreter's control strategy.

```
---------------------------------------
Input Clauses

1  A <- B
2  A <- C
3  B <- C & / & E
4  B <- F
5  C <-

Goal Clause

   <- A
---------------------------------------
```



Aside from the cut, a variety of other primitives have been provided within PROLOG systems to assist with the control of search. The microprocessor-based micro-PROLOG system [94] includes a 'single solution' primitive (syntactically, the symbol '!') which operates similarly to the cut but which is somewhat different in the manner of its pruning. The primitive FAIL, which can be used to (artificially and unsuccessfully) terminate a branch of the search tree, is available in most PROLOG systems.
Two forms of search control annotation which have been implemented in the IC-PROLOG system [74] are head annotations and guards. Head annotations are exemplified in Clark, McCabe and Gregory [75] by the following pair of procedures:-

```
[x has-descendant y^ <- x parent-of z & z parent-of y,
 x has-descendant y? <- x parent-of y & x parent-of z]
```

The bracketing expresses to the interpreter that the procedures are control (and not logical) alternatives. If the goal to be solved is of the form x has-descendant y with y an unbound variable, then the '^' annotation on the head of the first procedure will select that procedure for resolution. If, on the other hand, y is bound to a non-variable term in the call, then the '?' implies that the second procedure will be selected. The intended effect is to produce the differing algorithmic behaviours which efficiently solve the respective problems of finding descendents and finding ancestors. Guards follow from the idea of Dijkstra [86]. By annotating the first call of a procedure by ':', as illustrated by

```
B <- G: A1 & ... & Am
```

the call G becomes a guard which is evaluated when the procedure B matches the current goal (before the matching substitution has been applied to the A1, ... , Am). The aim is to more efficiently select the procedure which should respond to the given goal.
Metarules are statements which refer to program clauses and components of clauses whilst being syntactically separate from the program text. They can be used to augment the logic interpreter's default search strategy. An illustration is given by the PLANNER family [78], which enabled the programmer to supply 'recommendation lists' showing the order in which procedures should be investigated in response to a given procedure call. This mechanism can be regarded as a special kind of evaluation function, as investigated by Nilsson [43] and others for choosing between the nodes of a search space in path-finding and similar problem types. More recently a proposal for the use of metarules to direct the search control of logic programs has been advanced by Gallaire and Lasserre [72]. Their proposals incorporate a powerful form of PLANNER-style recommendation list, in which conditional orderings may be imposed upon the selection of procedures which are candidates for resolution with a given form of goal. This is illustrated by a metarule of the form:-

+OPORDER(P(x, y), n1.n2....NIL) - C1 - C2 - ... - Ck

which directs that goals which match P(x, y) are to be resolved with clauses numbered n1, n2, ... in that order, if the conditions C1, C2, ... are satisfied. The ordering can also be based on the content of clauses (so that when two or more clauses potentially respond to a call, preference can be given to a clause the body of which contains some specified condition). Gallaire and Lasserre's metarule scheme is extremely flexible: as but one illustration, they demonstrate how it could be used to impose a breadth-first search strategy on a depth-first PROLOG logic interpreter.

2.1.7 Extensions beyond Horn Clause Form

Several studies have investigated the representative capability of Horn clause logic and the computability theory of Horn clause logic with resolution inference. Among the most significant are those of Hill [76], Tarnlund [227], Andreki and Nemeti [228], and Sebelik and Stepanek [229]. These studies show that Horn clause logic is a universal computing formalism: it is equivalent in computational power to all the other universal formalisms which have been identified by the theory of computability. Indeed, the investigation by Tarnlund for example has shown that logic programs which are restricted to binary Horn clauses (those

which contain at most one condition) alone form a basis for computability.
Notwithstanding the theoretical adequacy of the Horn clause form of logic, it has frequently been argued that logic programming systems should go beyond it, expanding to some degree towards full clausal form and outwards towards the standard form of first and higher order logic. Kowalski [45] suggests that the representation of problems in standard form is often more economical and more natural than the clausal form representation, and that an extension of clause form is desirable for the specification of programs. Other extensions have been proposed on the grounds of possible benefits either in language power or in execution efficiency. However, it is clear that each such extension potentially brings with it complications of syntax and semantics, and in particular the effect (if any) of any extension on the soundness and completeness of standard SLD-resolution must be understood by programmers.
Some of the more important extensions are discussed below.

2.1.7.1 Negation

The negation operator '¬' of standard first order logic is not available in the clausal form. A headless clause such as

<- likes(x Tom)

('there does not exist an individual x who likes Tom') expresses a negation, but such a clause can only be useful in a top-down resolution inference system if it happens to be the goal clause: each resolution step resolves the goal clause with a headed clause (assertion or procedure), so that 'negative facts' cannot be deployed. For practical programming purposes however, probably the most restricting aspect of the so-called 'negation problem' for logic programming is the fact that clauses containing negative literals in their bodies, as illustrated by

P(x) <- ¬ Q(x)

which are allowed in standard form, cannot be directly represented in the Horn clause form of logic where all literals in clause bodies must be atomic formulae.
In theory, it is possible to express problems without explicit negation. Hogger [87] for example gives three alternative formulations, each one free of explicit negation, of a logic program corresponding to a problem which is specified using negation. His formulations respectively use special predicates, pragmatic control primitives, and computed arguments to substitute for the negation. However, it is seldom argued that such devices are

preferable to some extension beyond Horn Clause logic which enables negation to be explicitly deployed. Such an extension should provide a significant improvement in practical expressive power and should enable logic programs to be written which arise more naturally from problem specifications which include negation.
In fact most logic programming systems either do provide a negation meta-predicate, often named NOT, or else they make it possible for programmers to define one for themeselves.The intended effect of NOT (which is a meta-predicate because it takes an atom as an argument) is to implement negation-by-failure in which a call NOT(P) succeeds if a call P fails and vice-versa. Much investigation has been conducted into the soundness and completeness of negation-by-failure and its relationship to classical and other forms of negation.
A study of the negation problem has been conducted by Lloyd [130]. He presents a proof (due to Clark [88]) that negation-by-failure is sound (under SLD-resolution) on the main condition that programs are taken as representing their own completions. The completion of a program is formed by adding clauses corresponding to the 'only if' halves of the programmer's 'if' halves of definitions, together with certain axioms of equality: conceptually it amounts to the programmer's acceptance of the closed world assumption that the only information required to solve problems is that contained within the database. A second requirement for soundness is that in any call of the form NOT(P) which contains unbound variables, the success of P should not depend on binding any of those variables.
Kowalski [45] points out that negation-by-failure is easy to implement, efficient to use and has much of the expressive power of negation within the standard form of logic. Although its soundness depends on the closed world assumption, it is a fact that this assumption is already widely accepted in database work. It can however be argued that the 'global' application of the assumption over all program predicates is too indiscriminate and that the negation-by-failure property should be confined to individual predicates designated by the programmer. Such a refinement would be fairly straightforward to implement in most PROLOG systems, which often define the NOT by the meta-clauses:-

```
NOT(P) <- P & / & FAIL
NOT(P) <-
```

where the metavariable P represents an atom which could however be specialised for individual predicates as required.
It should be noted however that the above definition for NOT fails to apply the check on the binding of variables of P, and thus risks losing soundness. Lloyd (op. cit.) suggests that a fair computation rule should seek to delay the

selection of negative literals until they can be evaluated without breaking the bindings requirement. The MU-PROLOG system [131] has a fair computation rule, but the standard PROLOG computation rule as described earlier is not fair in Lloyd's sense. Some unfair PROLOG systems preserve soundness by generating an error condition if the bindings requirement for negative literals is breached, but others (for example, micro-PROLOG [94]) do not. Systems in this latter category attempt to reduce execution costs at the risk of soundness.

Within the closed world assumption, negation-by-failure is known not to be complete. Where a call NOT(P) succeeds, no solutions for the variables of the call can be produced because of the implied failure of P. It is this incompleteness of negation-by-failure which has led to much criticism of the implementation of negation within PROLOG systems. For example, Turner [89] has pointed out that NOT NOT P, which in standard logic is identical to P, is different from P in the PROLOG interpretation because the success of a goal NOT NOT P cannot generate bindings for variables (on account of the failure of the intermediate goal NOT P). Consequently, negative conditions 'become almost impossible to understand, especially when backtracking must be considered'. He proposes a scheme in which, in effect, variables range over a set of values which is declared explicitly to the program executor, so that solutions to negated conditions can be computed directly. A similar criticism of PROLOG is made by Wise [90], who describes the differing binding treatments (of negated as against un-negated atoms) as introducing 'a fundamental asymmetry'. Wise suggests that a partial solution to the problem is to add to the program a set of 'virtual items' which can be interpreted as negative facts.
An interesting question concerns the extra conditions which must be satisfied to ensure the completeness of negation-by-failure under the closed world assumption. This problem has recently been addressed by Lloyd (op. cit.), who proposes a proof credited to Jaffar, Lassez and Lloyd [91] of completeness in the rather special case where negation-by-failure is restricted to variable-free goal atoms (variables in negated goal atoms are not allowed, and literals in clause bodies must be positive). As yet no more general completeness result for negation-as-failure is known, and this appears to be a priority task for future research.

There are grounds for hope that a version of negation can be found for logic programming which corresponds to a larger fragment of classical negation than negation-by-failure whilst being tolerably efficient to implement. A recent proposal which appears promising is that of negation-as-inconsistency by Gabbay and Sergot [132]. They propose a scheme in which negative facts and rules, from which negative information could be explicitly computed, would be added to the database. It is argued that negation-as-inconsistency is always logically sound, and that relative to negation-by-failure it provides greater expressiveness (through the ability to represent negative information explicitly) and greater completeness (negated calls which succeed through negation-as-inconsistency can return solutions). Another significant claim for negation-as-inconsistency is that, like classical logic, it is monotonic: all previous results continue to hold when the set of clauses is enlarged (this is not a property of negation-by-failure, since new clauses may cause a previously failed call to succeed). The researchers, who admit that the implementation of this form of negation will be less efficient than that of negation-by-failure, propose to incorporate 'a fragment' of negation-as-inconsistency within their N-PROLOG logic programming system.

2.1.7.2 Lists of Solutions

Early users of Horn Clause programming systems discovered a difficulty in combining elements of information which originated in different parts of the search tree. As an illustration, Warren [93] notes that from a database of assertions such as

```
drinks(david, beer)
drinks(david, milk)
drinks(jane, water)
.....................
```

finding the answer to the question 'How many people drink milk?' seems to require that the solutions to the goal drinks(x, milk) should be somehow accumulated, in order that their enumeration could then give the required result. Early PROLOG systems required the programmer to employ ad hoc devices, usually making use of the assert primitive for adding a clause to the database, to accomplish the accumulation. These devices were typically tedious to program, difficult to understand and inefficient.
In addressing the problem, Warren (op. cit.) argues in favour of the introduction of set expressions to denote the set of all (provable) solutions to some goal. Consequently, more recent versions of his DEC10-PROLOG [73] incorporate a

new-built in procedure setof. The goal

```
setof(X, P, S)
```

(to be read declaratively as 'the set of instances of X such that P is provable is S') attempts to construct a list S containing all solutions for the variable X which make the goal P succeed. The goal statement

```
<- setof(X, drinks(X, milk), S) & size(S, N)
```

(assuming the existence of a program size for computing the length of a list) illustrates how the milk-drinking problem can be solved with the new construct. Because the setof predicate takes an atomic formula (P) as an argument, it qualifies as a meta-predicate (Warren calls it a 'higher-order' extension).
One measure of the appeal of setof is the appearance of similar set-constructor primitives in other PROLOG systems. For example:-

```
DEC10-PROLOG   setof(X, P, S)
IC-PROLOG      S = {X: P}
micro-PROLOG   (ISALL S X P)
```

It is evident that setof makes the closed world assumption for P. Under this assumption there seems no reason why the construct should not be sound. Its completeness, however, is a different matter and there are significant, but different, operational restrictions which affect each version. The IC-PROLOG [75] version cannot be called with S bound to a non-variable term (so that it cannot be used to check whether a list satisfies some condition) whereas the others can; although at least in the case of micro-PROLOG [94], this is generally unsafe since ISALL computes only one list S, the ordering of which is not always easy to predict. Only the DEC10 implementation represents sets as lists without duplicated elements (the others are liable to return lists with multiple copies of terms which were computed as solutions more than once in the evaluation). Only the DEC10-PROLOG primitive is 'backtrackable', generating alternative lists if the list expression contains variables: however Warren (op. cit.) explains that implementation of this capability required that the meaning of setof be restricted so as to exclude empty sets.

2.1.7.3 Other logic extensions

A number of other extensions to Horn Clause logic have been proposed for logic programming systems and several have been implemented. Three of these are outlined below.

2.1.7.3.1 Disjunctions

The Horn clauses

```
P <- Q
P <- R
```

can be expressed as a single clause in the standard form of logic, by forming the disjunction of conditions:-

```
P <- Q v R
```

A disjunction operator is provided in several PROLOG systems under various syntactical forms. The DEC10-PROLOG [73] user writes:-

```
P <- Q ; R
```

whereas the micro-PROLOG [94] programmer expresses the same clause as:-

```
((P) (OR (Q) (R)) )
```

Disjunction is easy to implement and it is efficient to use. It should entail no loss of either soundness or completeness. Used with restraint it can provide a welcome economy of expression, but there is a danger (especially where disjunctions and conjunctions are mixed within the same clause) of reducing readability. Clocksin and Mellish [95] recognise this in their tutorial guide to DEC10-PROLOG where they advise: 'You should always consider whether it may be worthwhile avoiding a ; by defining extra clauses'.

2.1.7.3.2 Conditional Alternatives

The clauses in the standard form of logic

```
P <-   Q & S
P <- ¬ Q & T
```

can be translated straightforwardly into the language of Horn clauses augmented with negation-as-failure. However, this introduces a source of inefficiency in that the same goal Q is sometimes evaluated twice. The use of conditional alternatives can avoid this inefficiency. Again, the syntax is prone to variation:-

```
micro-PROLOG    ((P) (IF Q (S) (T))
DEC10-PROLOG    P :- (Q -> S ; T)
IC-PROLOG       P if Q then S else T
```

It will be noted that the use of conditional alternatives obviates the need for negation-by-failure. The construct is

easy to implement (the micro-PROLOG definition of IF requires two simple clauses which make use of the 'cut'), is clearly sound and appears to preserve completeness. However, Clark, McCabe and Steel have pointed out that the efficiency advantage may be marginal in practice, because the single clause which uses the conditional alternative must be headed by an atom which is the 'most general' version of the heads of the two clauses which it replaces: hence less use can be made of unification to distinguish terms within the single clause [94]. Also, some reservation about the effect of such a construct upon readability appears as valid here as with the disjunction operator described above.

2.1.7.3.3 Implication conditions

In standard first-order logic, the definition of the subset relation can be given as:-

x subset-of y <- ∀z [z member-of x -> z member-of y]

Kowalski [45] shows how this can be expressed in the form of Horn clauses augmented with negation-by-failure:-

x subset-of y <- not ∃z [z member-of x & not z member-of y]

In effect, the standard form clause employs an implication as a condition and the re-written clause attempts to capture the meaning by two applications of negation-by-failure (sometimes referred to as equivalent double negation). Clark and McCabe have provided a primitive, the FORALL primitive, for expressing implication conditions directly within their micro-PROLOG system [96]. A clause such as

```
((subset-of x y)
   (FORALL ((member-of z x)) ((member-of z y))))
```

illustrates its use to define the subset relation. The construct is implemented directly via equivalent double negation; for interest, its actual defining clause is:-

```
((FORALL X Y) (NOT ? ((? X)(NOT ? Y))))
```

(where '?' is the micro-PROLOG meta-predicate which succeeds if its arguments succeed as goals). The micro-PROLOG reference manual [94] describes FORALL as 'a very high level concept, and can often replace explicit recursions in a program'. It is clearly more transparent to express implication conditions with FORALL than by the use of the equivalent double negation. On the other hand, this implementation of implication conditions inherits the weaknesses of negation-by-failure as discussed earlier.

Programmers who use it are implicitly accepting the closed world assumption for the predicates involved. The micro-PROLOG implementation of NOT is not sound (it omits the check on variable bindings) and, of course, the use of negation-by-failure risks a loss of completeness.

2.1.7.3.4 Metalanguage

In the presentation of clausal logic which was outlined earlier, variables implicitly range over a universe of discourse (the so-called Herbrand universe) comprising all variable-free terms which can be constructed from a language's constant and function symbols. The arguments of predicates are either variables, constants or functional expressions. The sets of predicate, constant, function and variable symbols are all mutually disjoint. Intuitively, a set of clause formulae expresses relationships between the objects which are named within the clauses.
However, a willingness to go beyond pure first-order logic enables the restrictions mentioned here to be relaxed. For example, variables can be allowed to represent predicates, atoms and clauses, and predicates can be permitted to take predicates, atoms and clauses as arguments. The sentences of this form of language, sometimes called the metalanguage to distinguish it from the object language described above, can express relationships between objects which are the formulae of the object language.
The metalanguage concept has been explored since the earliest days of logic programming. Significantly, the original Marseille PROLOG interpreter was developed specifically to support work in natural language processing, where the ability to name a whole sentence in one language (French) by means of a single variable in another (PROLOG) was found to be invaluable (an account of this is given by Colmerauer [97]). Kluzniak has noted that the core of the Marseille interpreter was itself a PROLOG program, and that subsequent PROLOG interpreters generally followed this lead: it became common to use PROLOG as a means to implement PROLOG (via bootstrapping) and the fact that clauses, atoms and functional expressions shared the same essential syntactical structure meant that they could all be manipulated by unification, so that the natural way to process PROLOG programs was through other PROLOG programs [98].
However, exploitation of the metalanguage potential of logic programming appears to have been somewhat haphazard, at least until recently. A variety of facilities are to be found in different PROLOG systems which are evidently intended to encourage some form of metalogical programming. For example, shown below are some primitives available under DEC10-PROLOG, together with brief explanations:-

clause(X, Y) succeeds when (the bindings of) X and Y can be matched with the head and body respectively of a database clause.

assertz(X) succeeds by appending X to the database. retract(X) removes the clause.

functor(T, F, N) holds when T is a functional expression having function symbol F and number of arguments N.

arg(N, T, A) holds when A is the N-th argument within the structured term T.

call(X) holds when X succeeds as a goal.

Most of these primitives have special restrictions, such as that X should be sufficiently instantiated to make the predicate symbol known in a call to clause(X, Y).
The question of how, and when, the metalanguage should be used is still relatively open. It is clear that metalanguage facilities such as those of DEC10-PROLOG can make programs very opaque; a striking example being the use of call(X) where it can often be difficult even to determine the identity of the procedure which is being called. Warren [93] has argued that allowing variables to represent function and predicate symbols adds nothing to the power of PROLOG, and notes that 'if predicate variables are used in more than small doses, the program becomes excessively abstract and therefore hard to understand'. Several authors have pointed out that the use of assertz (for example) creates a side-effect which makes programs difficult to follow. It also contradicts the simple conceptual basis of logic programming, which is deduction from a central fixed theory. However, some of the benefits of metalanguage applications to date have been substantial. A recent illustration is given by the micro-PROLOG system, which has relied heavily on the use of its metalanguage capability to implement a variety of software tools including PROLOG interpreters, editors and various debugging utilities [94]. Although the core of micro-PROLOG is written in assembly language, many of the primitives which implement the extensions to Horn Clause logic (such as the negation-by-failure, disjunction, and alternative condition extensions which are discussed above) are defined by PROLOG clauses which make use of metavariables.
Clark and McCabe have suggested analogies from other programming languages to the three main forms of metavariable use which are supported by the micro-PROLOG system [96]. A metavariable which represents a predicate is likened to a function or procedure being passed as a

parameter in Pascal, except that the Pascal parameter is not a 'first class' data object which can be processed arbitrarily (say, stored in a data structure) as can the metavariable representing a predicate in micro-PROLOG. Metavariables representing atoms are linked to the call-by-name mechanism of Algol 60, and metavariables representing argument lists are compared to the mechanism giving access to the arguments of a call as a list of items within 'C' or BCPL. Clark, McCabe and Steel [94] report that:-

> 'The meta-variable is very important to the usability of micro-PROLOG: it enables many of the second-order programs found in LISP (say) to be expressed succinctly in micro-PROLOG.'

It appears that what is needed is some means of exploiting the power of the metalanguage which at the same time preserves the clarity of logic programs and which also protects their clean semantics. A proposal which may well lead to progress in this direction has been presented by Bowen and Kowalski [92]. They argue that the object language of a logic programming system should be extended to include that part of a metalanguage which deals with the provability relation of the object language. This can be accomplished by the definition of a metalanguage predicate Demo, where Demo(prog, goals) can be derived in the metalanguage whenever the goals named by goals can be derived from the clauses named by prog in the object language. Having defined Demo, any direct execution in the object language can then be simulated by a Demo computation in the metalanguage. In a sense, the definition of Demo generalises the metarule schemes, such as that of Gallaire and Lasserre [72] outlined earlier, which are directed towards the control of logic programs. Bowen and Kowalski suggest that the language produced by amalgamating the object language with this form of metalanguage will have greater expressive and problem-solving power whilst the standard semantics of logic will be preserved. They show how their scheme could provide a device for the local definition of predicates within logic programs, and how it could be used to implement a 'lists of solutions' procedure (as discussed in section 2.1.7.2). Elsewhere, Kowalski has successfully applied this amalgamation of object language and metalanguage to the problem of database management in logic [66]. The object language is used to describe and query databases, whilst the metalanguage is used to construct and manipulate them. A metalanguage formulation comprising a set of four rules is given for assimilating new information into the database. The example appears to be a powerful illustration of the value of a systematic exploitation of metalanguage.

2.1.7.4 Subsidiary Features

In addition to providing a language for expressing relationships as logical formulae, together with some means of arranging deduction from those formulae, all practical logic programming systems have certain subsidiary features. This section considers two of the most important categories of those features, namely those which are concerned with arithmetic primitives and input/output provision, and it identifies some of the issues which seem to arise from experiences to date.

2.1.7.4.1 Arithmetic Primitives

Logic assigns no special meanings to terms such as product and to symbols such as '+'. However, there are good reasons for pre-defining such identifiers within logic programming systems consistently with their usual arithmetic roles: one reason is to provide a convenience for the programmer, and another is to attempt an efficient implementation by building the definitions at a suitably low level into the system.
The majority of PROLOG systems, including DEC10-PROLOG, the UNIX PDP-11 PROLOG system, and the DEC LSI-RT11 PROLOG system, have pre-defined the predicate is for the purpose of arithmetic evaluations. A goal of the form:-

```
X is Y
```

is legal if X is uninstantiated and Y is instantiated to an expression which evaluates to an integer, and the goal will succeed by binding X to the integer. (It is also legal for X to be instantiated to an integer, in which case the goal succeeds if the integer is identical to the result for Y). The expression may include the operators '+', '-', '*', '/' and 'mod', which have their usual interpretations. The other main arithmetic facility is the provision of six predicates used for comparing integers, as follows:-

```
X  =  Y   (equality)
X \=  Y   (inequality)
X  >  Y   (greater than)
X  <  Y   (less than)
X >=  Y   (greater than or equal)
X =<  Y   (less than or equal)
```

where the usual restriction is that both X and Y must be instantiated to integers (DEC10-PROLOG permits them to be expressions which evaluate to integers).
An significant criticism of these facilities is that they do not fully support a logical view of the relations which are

represented. For example, the is primitive can almost be characterised as implementing a conventional function, which here delivers to the variable on its left the integer result of evaluating the expression on its right. The characterisation would be a precise one were it not for the possibility that the left argument may be already instantiated, which may be regarded as a very small concession to the invertibility of programs that is such a central feature of logic programming. Even this concession is missing in the predicates, however, and a goal such as

```
X > 0
```

with X uninstantiated will return an error message on these systems. This is not because no integers exist which are larger than zero, of course; it is simply a consequence of '>' having been implemented as a boolean function which takes two call-by-value parameters.

The justification for such an approach to arithmetic provision within logic programming systems rests on the grounds of efficiency. It is reasonable to expect that a purely deterministic, functional implementation of arithmetic will by and large be less costly to implement and will run more efficiently than a non-functional implementation. However, it can also be argued that the functional approach is too narrow to fit well into the relational world of logic programming. The restrictions on the input/output patterns force the programmer's attention away from the specification of relationships and onto the instantiation states of variables. The positioning of arithmetic conditions relative to other conditions within the bodies of clauses becomes critically important. In other words, considerations of control are necessarily brought to the fore. Furthermore, experience has shown that such restrictions on the use of primitives tend to have repercussions which spread beyond the clauses in which they appear and outwards into the program beyond.

Some implementers have tried to construct systems which provide support for arithmetic in a manner which is more in keeping with the spirit of logic programming. Exeter PROLOG [99] for example does not insist that the right hand argument of is evaluates fully to an integer: given a goal such as:-

```
7 is Y + 3
```

with Y uninstantiated, it is capable of discovering the solution Y = 4. Micro-PROLOG [94] takes the relational approach further, abandoning the is in favour of individual predicates such as SUM and TIMES where, for example,

```
SUM(5 x 20)
```

(with x uninstantiated) succeeds with the binding x = 15.

Thus, SUM can be used for subtraction as well as having its expected use in addition. Clark and McCabe, writing about the arithmetic primitives in their tutorial guide to micro-PROLOG [96], explain that

> 'Although each arithmetic relation is implemented by a machine code program, so as to make use of the hardware operations of the machine, we can think of each relation as being defined by an implicit data base of facts.'

The idea of implementing arithmetic predicates as simulations of the relations which they compute appears to offer a promising alternative to the functional approach. In general, of course, an infinite number of tuples will lie in each relation: in the case of SUM, an infinitity of triplets (X Y Z) solve the equation X + Y = Z and so satisfy the relationship SUM(X Y Z). A call to SUM should succeed in as many ways as its arguments can be instantiated so as to lie in the addition relation. Thus a call of SUM(X Y 10) with X and Y unbound should generate pairs of numbers which add up to ten; a call X LESS 0 should generate negative numbers; and so on.
In fact, micro-PROLOG does not go as far as this. It insists that the arguments of calls to arithmetic primitives are sufficiently instantiated to ensure that a deterministic computation is all that is required to solve the call (for example, any SUM call must have at least two of its three arguments bound). The justification which is offered for this restriction is based on considerations of efficiency. IC-PROLOG [74], however, does have genuine non-deterministic arithmetic, albeit restricted to the natural numbers. It implements three primitives TIMES, PLUS and LESS which have no instantiation restrictions on their use, so that for example TIMES(x, y, 36) with x and y unbound will return pairs of factors of 36. Clark, McCabe and Gregory have shown how this enables elegant arithmetic programs to be written which are just the obvious definitions of the relations which they compute [100]. For instance

```
x divides z <- TIMES(x, y, z)
```

defines the divides relation and can be used for testing (x and z instantiated), finding divisors (only z instantiated) and finding multipliers (only x instantiated). Similarly

```
has-divisor(z) <- s(s(x)) divides z & ¬s(s(x)) = z
```

defines the has-divisor relation (where ¬ represents negation by failure and s is an IC-PROLOG primitive defining the successor relation for natural arithmetic, so that s(s(x)) denotes any integer greater than one). The IC-PROLOG query

{z: has-divisor(z)}

will now generate the sequence 4, 6, 8, 9, ... of properly divisible natural numbers, terminating when interrupted or when overflow occurs in the host computer.

It is clearly highly desirable that the algorithms which implement non-deterministic arithmetic can recognise opportunities for termination. As a simple example, the IC-PROLOG program for TIMES recognises that the search for solutions to the goal TIMES(x y 12) can be limited to pairs of natural numbers up to twelve. Unfortunately the execution of the conjunction

LESS(1, x) & LESS(x, 9)

as a goal will not terminate upon generating the solutions 2, 3, .. , 8. Under the standard backtracking control strategy (which is the default strategy of IC-PROLOG) the first call to LESS will continue to generate numbers greater than one even though none of them can satisfy the second condition.
It is clear that a non-deterministic implementation of arithmetic gives the programmer scope for introducing an arbitrary level of inefficiency. Nevertheless it enables the specification of relations to proceed from purely logical considerations and it guarantees that these specifications will be executable, however inefficiently, from the first stages of development. This is consistent with the view of logic programming software development which will be discussed in Part Three.
Non-deterministic arithmetic needs further investigation. The question of whether it can be extended over the whole of integer arithmetic, and of whether there exists a worthwhile extension to real arithmetic, should be explored. The relationship between non-deterministic arithmetic and the control of logic programs is also of interest.
Although the relational notation has made possible non-deterministic arithmetic, it is recognised that functional notation can be more natural and more compact. Kowalski [101] quotes as an example the (non-Horn) clause defining the relation of orderedness for a sequence of terms:-

x is ordered <u>if</u> <u>for all</u> i (xi <= xi+1)

A functional notation is used here to name the items of the sequence (xi, x1+1) and the integer successor of i (i+1). Exchanging the functional for a relational notation gives something like:-

```
x is ordered if for all i, j, u, v
      (u <= v if Plus(i, 1, j) and
      u is the i-th item of x  and
      v is the j-th item of x).
```

The relationship between functional and relational notation has been described elsewhere by Kowalski [40]. He suggests a method whereby functional equations, provided they are first-order (that is, functions may not take functions as arguments), can be transformed into Horn clauses. In consequence, logic programming can exploit the convenience of functional notation whilst retaining the semantics of relations.
In practice however most PROLOG systems have given little or no support to the functional notation beyond the *is* predicate for simple arithmetic. One of the exceptions is micro-PROLOG (when augmented with the required library procedures) in which a clause such as

```
related(x y) if
                  linked#((double x) y1) &
                  y = (3 + (cube y1))
```

is permitted (this example is rather contrived). Here, the terms (double x) and (cube y1)are functional expressions which will be evaluated by applying the programmer-defined functions double and cube to the respective arguments. (Unfortunately, micro-PROLOG does not have the functor syntax for data constructors common to most other PROLOG systems, so that functional expressions must be written in list form). Functions of n arguments are defined first as n+1 -place relations, for example

```
double(x y) if TIMES(x 2 y)
```

A subsequent command function double then adds a control clause (in effect, a metarule) to the database which tells the interpreter to recognise double in expressions as a function. The result returned by the function for a given input is obtained from the relation's second argument when the relation is evaluated with its first argument replaced by the input. The '#' which follows linked is a control annotation warning the interpreter that some of the arguments which follow this predicate are expressions which need to be evaluated. The predefined equality predicate is similar to the is of traditional PROLOG, except that both arguments may be expressions, and (as the example shows) these may include programmer defined functions in addition to the usual arithmetic operations.
Although the micro-PROLOG implementation of functions can be criticised (on the grounds of its syntactic inelegance, for example) it does show that logic programming systems can support the functional notation as a practical proposition. The main benefit should be the convenience of this notation,

although there should also be efficiency advantages to be exploited. However, if the selection of a functional notation implies an implicit control decision on the part of the programmer, then there is a risk of losing completeness even although the predicate involved might later be implemented as a totally invertible relation; this seems to be undesirable. More experience of the use of functional notation within practical logic programming systems should cast more light on this aspect.
Some researchers have criticised the relatively meagre mathematics facilities of existing logic programming systems. Clearly the limitation of traditional PROLOG to integer numeric data and the basic operations of arithmetic reflects its origins in artificial intelligence. Fogelholm [99] has suggested that in order to support a broader range of applications, PROLOG systems should provide floating-point arithmetic and 'a reasonable set of transcendental functions'. Some more recent systems, such as the hybrid POPLOG system which is embedded within POP-11 [102], have managed to provide some such facilities by inheriting the primitives of the host environment. Again, micro-PROLOG is surprisingly good in this respect notwithstanding the limitations of an underlying microprocessor architecture: it provides full floating point arithmetic, up to the limits of the hardware. Micro-PROLOG does not, however, offer transcendental functions.

2.1.7.4.2 Input/Output provision

Logic programs must communicate with the outside world. They produce output which must be directed onto some appropriate output device, and frequently they require input which must be gathered from some input device. Obviously then, logic programming systems are required to make suitable provision for organising input and output.
The standard input and output mechanisms are the query and the answer extraction facilities respectively. These facilities are natural and unobtrusive and they are built into logic programming systems in a variety of forms. For example, a typical query/answer interaction with the DEC-10 PROLOG system might be:-

```
?- likes(X, mary).
X=john;
X=paul;
no
```

(where the user's input is shown underlined). The corresponding interaction with the micro-PROLOG SIMPLE interpreter of is:

```
which(X: likes(X mary))
john
paul
No (more) answers
```

This 'default' input/output provision can be made more flexible, for example by providing query mechanisms which arrange for formatted or graphical output, or which enable the selection of different output devices, and so on. Facilities of this kind are commonly provided by relational database systems, for example. Existing logic programming systems also offer some extensions of this type, and this seems to be a desirable area for further developments. The query/answer mechanism is the form of input/output provision which fits most naturally into the logic programming view of computation as controlled deduction, and it obviates the need for programmers to introduce explicit input and output-handling procedures into their programs.
However, the need to write sub-programs specifically to manage input and output cannot always be avoided. Interactive programs must generate run-time interactions with users; many programs require access to data held on secondary storage devices; and output requirements are sometimes so specialised that 'custom' programming is the only realistic choice. Hence, logic programming systems have provided specific input/output primitives which are independent of the basic query/answer mechanism. The read(X) and write(X) primitives of DEC10-PROLOG, which read into X a term from the current input stream, and which write the term X into the current output stream respectively, are typical examples. To select an arbitrary file as the input or output stream, the primitives see and tell are provided (the keyboard and screen are defaults). Calls to these primitives may be incorporated into program clauses as with any other atomic formulae.
The input/output primitives of PROLOG systems have caused much discussion. The following observation by Sergot [103] seems fairly representative:-

> '... the writing of interactive PROLOG programs has remained a problem. Input-output facilities are notoriously non-logical. Read and write commands are used only for the side-effects they cause, so interactive PROLOG programs cannot be understood without knowing how they will behave.'

It is interesting to compare this with an observation by Kowalski that interactive logic programs can exhibit declarative input/output [40]. He quotes (from a medical expert system program) the clause:-

y is unsuitable for x if y aggravates z
and x has condition z

Kowalski suggests that the definition of x has condition z can be provided dynamically by the user. He argues that:-

> 'This makes input-output declarative in the sense that it can be understood entirely in logical terms: the output is a logical consequence of the information initially contained in the system together with any information provided by the user ... The input can be given in any order, provided it does not affect the logical implication of the output.'

Whilst accepting the importance of the logical reading of the clause, many programmers would probably argue that the procedural interpretation is also highly significant here. The need to constrain the type of interaction generated by the program forces attention onto the order in which the two calls in the body of the clause will be executed. At a lower level of definition (and especially at the level at which read and write calls, or their equivalents, must be inserted directly), the logical interpretation often disappears and only the procedural reading is meaningful.
An obvious partial solution to the input/output problem is to apply a disciplined style of programming which isolates low-level input/output calls from the rest of the program. Some researchers, including Sergot (op. cit.) and Ennals, Briggs and Brough [104] have suggested that logic programming systems should assist by providing higher-level facilities for input and output. As one example, Sergot describes a query-the-user system for logic programming in which the format of the user's queries to the system mimics that of system's queries to the user. The facilities he describes, which are implemented in micro-PROLOG using metalogical programming, are available as part of a package of tools designed to support expert systems development. The package is known as APES [105]. Another example of high-level input/output 'packaging' is provided by the SIMPLE front end to micro-PROLOG which includes, in addition to the primitives R and P which correspond to the read and write of DEC10-PROLOG, a primitive is-told which enables a complete input/output interaction to be programmed by a single call. An illustration, based on an example given by Clark and McCabe [96], is the clause:-

Smith sells-electrical x if
Smith sells x and
(x electrical) is-told

A call of (say) (light-bulbs electrical) is-told will cause

the prompt light-bulbs electrical ? to be displayed. The call will succeed or fail depending on whether the user responds yes or no respectively. The is-told primitive can also cope with arguments containing uninstantiated variables, and micro-PROLOG's backtracking can be conditioned by the user's response. Clauses which use this primitive generally have a good logical reading, as illustrated by the example shown here.

Notwithstanding the worth of higher-level primitives such as is-told, it seems likely that there will always be occasions when requirements are sufficiently unusual to necessitate that programmers create their own input/output procedures. This is particularly true in realistic applications where input must incorporate error-trapping and, indeed, where the man-machine interface may require sophisticated screen displays incorporating, possibly, icons and windowing. If it is agreed to approach applications of this type purely within a logic programming framework (and as Part Three will show, some researchers would doubt the wisdom of this) then a highly modular implementation of the required procedures seems to be called for. The provision of suitable module facilities within logic programming systems will be discussed later.

To date, it seems that relatively little attention has been given to questions relating to the generation of graphical output by logic programming. A collection of (machine dependent) graphics primitives is to be found in the various implementations of micro-PROLOG (such as those for the Sinclair Spectrum [106] and the BBC microcomputer [107]): the M.Sc. thesis of Julian [108] discusses micro-PROLOG graphics at length. An interesting short contribution to the use of graphics within logic programming is offered by Kowalski [109], who uses the relationship x names y to relate a 'picture-plan' x, which is a sequence of actions (coded in a style similar to the 'turtle geometry' of the programming language LOGO [110]) denoting a picture, to the name of the picture y. Kowalski then shows how a predicate draws can be defined such that a call to x draws y will construct a screen image of the picture. He argues that the specification of pictures as picture-plans by means of Horn clauses which can be interpreted as procedures to actually construct the pictures has advantages both in proving properties of programs and in reasoning about the pictures themselves.

2.2 Logic Programming Systems

This section reviews some existing logic programming systems. Since PROLOG systems of various kinds have so far dominated the practical aspects of logic programming, they have a predominant place here. First, however, consideration is given to some basic aspects of implementation.

2.2.1 Implementation aspects

The great majority of logic programming systems which have been implemented to date have been PROLOG systems. Until quite recently very little had been published on the technical aspects of implementation, but the situation has now changed. The book edited by Campbell [111] contains a large collection of papers on PROLOG implementation and represents a significant source of information. Hogger [87] provides a thorough introduction to implementation, and the collection of papers edited by Clark and Tarnlund [112] includes three papers on the subject. It may be helpful here to briefly sketch the mechanism of logic program interpretation which is most commonly documented. Note, however, that the description given here is a somewhat idealised abstraction of many variations.
The essential task of a Horn clause logic interpreter is to construct and store the search tree which corresponds to the user's program and the goal. However, it would be wasteful to store a concrete representation of the goal statement obtained at each node, since this would involve making a fresh copy of the body of a procedure each time the procedure is used. Instead, only one copy of each clause need be stored in an area of memory known as the heap. The construction of the search tree is usually represented by a stack of frames, known as the execution stack. Each frame (or activation frame as it is sometimes known) is a record of a single unification, and it contains a system of pointers to parts of the heap and to other parts of the stack which together represent the current state of control. In addition, each frame contains pointers which enable the substituting value for each variable involved in the unification to be retrieved. The device whereby many calls to a procedure can all 'share' the same representation of the procedure is known as structure sharing and is due to Boyer and Moore [113]. As it happens, the same principle of structure sharing can be applied to the representation of the values of variables: a structured value (functional expression or list) can be represented by a 'skeleton' which yields its structure together with an 'environment' which yields the values which are to be applied. In this scheme, the value of a structured variable can be represented within an activation frame by a pair of pointers (known as a

molecule) to the corresponding skeleton and environment. The actual skeletons and environments can be stored on the heap. Implementations which use this system for representing data are known as structure-sharing implementations. However, non-structure sharing implementations have also been developed: these implementations compute the value of a structured term directly (with pointers to other values where necessary) and store this value on the heap. Discussions of the respective merits of structure-sharing versus non-structure sharing implementations are to be found in Mellish [114], Bruynooghe [115], and Kahn and Carlsson [116], but the outcome is far from clear-cut and both machine architecture and intended applications are proposed as factors which should influence future choices.
At least two interpreting algorithms for PROLOG programs using the stack-frame representation for unifications have appeared. Both Hogger [87] and van Emden [117] have published interpreting algorithms in an Algol-type language: these appear to be very similar and quite straightforward. A crucial component of any interpreter is the unification mechanism itself, and this has been the object of much investigation. Robinson [118] has noted that

> 'The unification algorithm started its existence as one of the most inefficient algorithms ever proposed. It was exponential in space and time. Now we have progressed so far that, rather than being exponential, it is now nearly linear'.

Recently a new unification algorithm has been presented by Martelli and Montanari [123], who claim that their version compares favourably with the more established algorithms of Huet and of Paterson and Wegman. The researchers claim that the new algorithm can achieve an exponential saving of computing time in a resolution-based theorem prover. It is also claimed to perform well in detecting those illegal cases in which variables may become bound to structures which include occurences of themselves (for example, x to f(x)): many existing PROLOG interpreters omit this so-called occur-check entirely. The justification for the omission of the occur check is usually given in terms of efficiency (it is only very rarely required), but the saving is known to be potentially unsafe and it can actually destroy the soundness of SLD-resolution. Lloyd [119] for example presents several examples in which a PROLOG system without the occur check will give wrong answers. The problem has been studied by Plaisted [120] who suggests that a preprocessor could identify the program clauses which might cause problems so that appropriate action can be taken (for example, full unification could be invoked for these clauses).
Methods of improving the efficiency of PROLOG interpreters have also been studied. Bruynooghe [115] for example has

described the opportunities presented by deterministic procedure calls, and by tail recursion, to conserve memory. Elsewhere he has written about PROLOG garbage collection [121]. Substantial gains in execution speed are offered by compilation, which at present is only available in Warren's DEC10-PROLOG system; a major source of increased efficiency stems from the machine-code unification routines which the compiler generates specifically to match the formal parameters of each program clause (this gain can be improved further if the programmer is willing to provide so-called mode declarations corresponding to a procedure's intended pattern of use). Warren [122] has compared the execution performance of his compiler with the Stanford DEC10-LISP compiler, which is recognised to produce quite fast code: he reports that for simple list-processing examples, PROLOG is slower than LISP by a factor of about 0.6, but that in other examples PROLOG can be faster than LISP by a factor of two or more. Warren also points out that his compiler is optimised for space rather than for speed and he has hopes of a further speed improvement by a factor of two.
PROLOG systems have been developed using several implementation languages, including ALGOL, FORTRAN, PASCAL, 'C', PROLOG and various assembly languages. However, it has become common to implement at least the logic extensions to Horn clause logic in PROLOG itself.

2.2.2 Features of Existing Systems

This section outlines the main features of some existing logic programming systems. It is hoped that the systems selected will suggest the variety of facilities which have been documented to date. A short overview is provided for each system, following which attention is given to the language syntax, the logic interpreter's control strategy, the language's logic extensions to Horn clause form, and the language's subsidiary features. Finally, a note is included which briefly documents the main implementation aspects of the system.
It should be noted that this section attempts to express the general 'flavour' of the systems covered. An effort has been made to avoid excessive detail. References have been given to sources of further information.

2.2.2.1 Micro-PROLOG

Overview

Micro-PROLOG [96, 94, 106, 107] is a microprocessor-based implementation of PROLOG running under the CP/M and MS-DOS operating systems, as well as a number of individual machine operating systems. It has been developed by Logic Programming Associates, which is led by the two well-known logic programming researchers, Keith L. Clark and Frank G. McCabe. Notwithstanding its microprocessor architecture, the micro-PROLOG system has incorporated some innovative features. The core of micro-PROLOG itself is small with a LISP-like language syntax. However, it provides full support for metalogical programming and (unusually for PROLOG systems) there are facilities for constructing modules. Among the modules which are provided with the system are editors, debuggers, translators, and various other tools, all of which are actually PROLOG programs in which metavariables feature prominently. Notable among them are the SIMPLE translator, which provides a convenient syntax and a pleasant set of facilities for PROLOG programming, and another translator which implements a large subset of the facilities of the DEC10-PROLOG system. Either of these translators can act as 'front ends' which overlay the basic micro-PROLOG system.

Syntax

The Horn clause

```
Grandparent(x, y) <- Parent(x, z) & Parent(z, y)
```

could be expressed in core micro-PROLOG as

```
((Grandparent x y)(Parent x z)(Parent z y))
```

A micro-PROLOG clause is a list of atoms. Each atom is a list of terms, where the head term denotes the predicate. Terms may be numbers (integers or floating point), constants (sequences of characters beginning with a letter, such as Grandparent), variables (which are limited to x, y, z, X, Y, Z, x1, y1, z1, X1, Y1, Z1, ...), and lists of terms. The list notation (x¦X) is available which denotes a list having x as its first term and X as the list of all remaining terms. Lists in fact are used to the exclusion of the functor-prefixed structures which are provided by most other PROLOG systems. A query is expressed by '?' followed by a list of atoms which are interpreted as conjoined goals, for example:-

```
?((Grandparent x y)(male y))
```

The system response to such a query indicates only success or failure: if further information, such as answer bindings, is required then it must be explicitly programmed into the query. The SIMPLE translator offers a 'sugared-up' syntax in which the clause above would appear as

```
Grandparent(x y) if
                    Parent(x z) and
                    Parent(z y)
```

The SIMPLE equivalent of the query above would be:-

```
is(Grandparent(x y) and male(y))
```

In addition to the standard prefix atomic form, SIMPLE provides an alternative postfix notation for unary atoms and an alternative infix notation for binary atoms. Micro-PROLOG does not provide an operator precedence grammar (as described below) as do many other PROLOG systems.

Control

The micro-PROLOG system interprets logic programs using the standard control strategy as described earlier. That is, the computation rule is the selection of goals on a last-in first-out left-to-right basis. The search strategy is depth-first (backtracking) with clauses investigated textually top-to-bottom.
Programmers may limit the extent of the backtracking by means of the cut (/) and the single-solution annotation (!). Both of these were discussed in an earlier section.

Logic Extensions

Micro-PROLOG incorporates primitives corresponding to negation by failure (NOT), lists of solutions (ISALL), disjunctions (OR), conditional alternatives (IF), and implication conditions (FORALL). These extensions were all discussed in section 2.1.7.
Metalogical programming is supported. A variable may be used to name a predicate symbol, an atom in the body of a clause, or the tail of the body of a clause. Various primitives are provided which take atoms or clauses as arguments, such as the logical operators mentioned above and there are various database-modifying predicates such as ADDCL and DELCL.

Subsidiary Features

Several important subsidiary aspects of micro-PROLOG, including its arithmetic primitives, the facility for

defining functions (which in most versions is actually accessed through a utility module), and certain input/output facilities have already been mentioned. Among other features of the system, the module construction facilities and the external database provision are especially interesting.
A micro-PROLOG module is a named collection of clauses. A module may communicate with other modules, and with the un-modularised or 'workspace' clauses, via import and export lists. The form of a module is:-

```
<module name>
<export list>
<import list>
.....................
{collection of clauses}
.....................
CLMOD.
```

A module's import list contains the constants which arise outwith the module but which must be recognised inside it: these constants include predicates defined outside the module, and also 'data constants' (such as the constant arguments of relations) which must be recognised by the module's clauses. A module's export list contains all the predicates defined inside the module which are to be made available outside (that is, they will be available to the workspace and also to other modules which mention them in their import lists). Constants which appear within a module, but which are not included in the import/export lists, are 'local' to the module and these constants can be used elsewhere without fear of a clash of names. Micro-PROLOG maintains an independent dictionary for the (import, export and local) names of each module, and it also maintains a dictionary for the workspace clauses: by this means, the clauses defining module relations are 'invisible' to the user from the point of view of his workspace clauses.
The external database provision of micro-PROLOG makes it possible to extend the apparent workspace available for clauses beyond the limits of the microcomputer's direct access memory. It does this by storing part of the database on a disk file, the clauses of which however can nevertheless be utilised with the same generality as those which occupy the main memory. From the programmer's viewpoint, the extension (which involves the use of a system utility module called EXREL) is almost completely transparent. Of course, the programmer's decision to make use of the external database provision represents a trade-off of execution speed against main memory space.

Implementation

Most of the micro-PROLOG system is implemented in assembly language, although parts (including most of the logic

extensions) are defined by PROLOG clauses. It is a non structure-sharing system. The interpreter, which omits the occur check on unification, has the capability to exploit the space saving opportunities which are offered by deterministic procedure calls and by tail recursion. In addition to providing 'extensibility from above' through the module construction facility, micro-PROLOG provides 'extensibility from below' in the form of a machine-level interface which permits new predicates to be defined through assembly-language programs.

2.2.2.2 IC-PROLOG

Overview

The IC-PROLOG system [74, 75, 100, 124] was developed at Imperial College and is established on IBM-370 and CDC-6000 mainframe computers. The main motivation behind its construction appears to have been experimental with special emphasis on PROLOG control provision. It has also been used for the purpose of teaching a variety of programming concepts, including those of lazy evaluation and communicating processes.

Syntax

The basic syntax of IC-PROLOG is very close to the standard syntax of Horn clause logic. For example,

```
grandparent(x, y) <- parent(x, z) & parent(z, y)
```

is a valid procedure in IC-PROLOG and

```
parent(John, Mary)
```

is a valid assertion. Variables are identifiers which begin with lower-case letters. Functional terms are permitted. The system provides a form of operator precedence grammar which enables functors and predicates to be declared by the programmer as infix, prefix or postfix, with a specified associativity (either RIGHT, LEFT, NOT or ASSOCIATIVE) and a relative precedence (from 0 to 100): this provides some syntactical flexibility from the prefix normal form. As an illustration, IC-PROLOG pre-defines the Cons functor symbol which can be used for lists as in LISP so that the list (A B C) can be written as

```
Cons(A, Cons(B, Cons(C, NIL)))
```

Alternatively, with the declaration of the symbol '.' as a list functor using the IC-PROLOG system directive

```
$oper('.', 0, INFIX, RIGHT)
```

the dot subsequently becomes a right associative infix functor with precedence zero, and the same list can now be written as

```
A.B.C.NIL
```

IC-PROLOG has three forms of query. The most general is

```
{t: L1 & .. & Lk}
```

where t is a term and the Li are literals: this requests the set of all values of t which make the goals Li succeed.

Control

The default control strategy of IC-PROLOG is the standard strategy. However, the system does not provide a 'cut' and most of the other commonly found metalogical features (such as isvar and assert) are absent. In compensation a rich set of control annotations are available, some of which (such as the bracketing of procedures which are search control alternatives) have already been described. The three examples below, which are taken from a fuller account written by Clark et al [100], illustrate the facilities for non-sequential evaluation of procedure calls. Each example refers to the classic problem of checking that two given tree structures have the same leaf profile.

1) Unsynchronised Parallel Evaluation

The '//' symbol in the procedure

```
sameleaves(x, y) <- w profile-of x // w profile-of y
```

has the declarative meaning 'and', but with the control effect of executing the two calls as pseudo-parallel processes. The processor timeshares between them, giving each process at least enough time for a single resolution step. It is not possible for more than one process to bind the same variable at the same time. If one process adds a leaf label to w which is not then matched by the other process, the parallel evaluation ends.

2) Parallelism with Directed Communication

The '^' variable annotation in the procedure

```
sameleaves(x, y) <- w profile-of x // w^ profile-of y
```

specifies that the second call is the so-called producer of the variable w. Only this call is allowed to generate a binding for w; the other call, which is suspended if it tries to bind w, acts like a so-called consumer in that it is confined to checking bindings which are passed by the consumer.

3) Data Triggered Coroutining

By reverting back to the '&' connective, the procedure

```
sameleaves(x, y) <- w profile-of x & w^ profile-of y
```

becomes one in which evaluation alternates between the two calls. The second call acts like a so-called lazy producer for w. As soon as it generates a label for w, the first call is activated to check that the profile of x agrees with this label. If it does not then the evaluation fails.

These control annotations, together with others which are described by Clark et al (op. cit.), can be mixed to specify a variety of different control strategies.

Logic Extensions

IC-PROLOG provides negation-by-failure (¬A), lists of solutions (x = {t: A}), and conditional alternatives (P THEN Q ELSE R).

Subsidiary Features

The invertible arithmetic relations of IC-PROLOG have been discussed previously. Two other interesting features are stream input/output and system directives. Stream input is implemented via the READ(x) primitive which binds its argument to the entire stream of characters typed at the terminal. The stream is processed as a list which is lazily produced. The corresponding WRITE(x) primitive can also be used with its argument generated as a stream. An example of a system directive was given above. All directives appear syntactically as assertions for predicates which commence with the character '$', and they either relate to the way in which the system processes the input set of clauses or else they provide control information to the interpreter. Some examples are:-

```
$EDIT (<Relation name>)
          (Invoke inbuilt editor for relation)
$SAVE (File)
          (Save program in File)
$TRACE
          (Turn on interpreter's trace facility)
$FUNCTION (<Relation>, <List of argument positions>)
          (Tell interpreter that Relation
          is a function of given arguments,
          enabling space optimisations)
$OCCUR
          (Invoke  occur check on unification)
```

Implementation

The IC-PROLOG system was written in Pascal. It uses a unification procedure which omits the 'occur check' by default, although as indicated above the programmer can use a directive to change this.

2.2.2.3 DEC10-PROLOG

Overview

The DEC10-PROLOG system [73, 122, 125] was developed in Edinburgh by Warren, Pereira and Pereira. It is well-documented by the standard of logic programming systems; this is the system on which is based the well-known PROLOG tutorial text by Clocksin and Mellish [95]. This is the best known PROLOG system and it has produced many offshoots. DEC10-PROLOG is particularly respected for the quality of its implementation, which includes an option for program compilation.

Syntax

Basic DEC10-PROLOG syntax is close to that of IC-PROLOG. The clauses

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
parent(john, mary).
```

are examples of a DEC10-PROLOG rule and an assertion respectively. Variables are marked by upper-case initial letters.The system must be explicitly told by the command [user]. to accept such clauses from the terminal: in its 'default state' it expects queries. An example of an interaction involving a query is

```
?- likes(X, Y).

X = john,
Y = alfred;

X = alfred,
Y = john;

X = david,
Y = bertrand;

no
```

Here, the underlined text denotes the user's input. In response to each semi-colon DEC10-PROLOG attempts to discover a further solution to the goal and finally it prints no when this is impossible.
Functional terms are permitted, and as with IC-PROLOG an operator precedence grammar enables the programmer to specify some flexibility of syntax from the prefix normal form for functors. This is accomplished by executing a goal of the form op(Prec, Spec, Name) which specifies that the operator Name is to have the associativity corresponding to Spec and the precedence level Prec. To illustrate, the usual dot functor for lists could be declared by means of the goal op(51, xfy, '.') which declares the dot as a right associative infix operator with precedence level 51. In fact, DEC10-PROLOG by default provides programmers with an alternative notation for lists which is very similar to that of micro-PROLOG: the list [X|Y] denotes a list of which X represents the first member and Y represents the list of all other members.

Control

The control strategy is the standard strategy. The 'cut' (symbol '!') is available to control the extent of backtracking. Another pre-defined control predicate (and one which suggests that DEC10-PROLOG encourages a rather pragmatic view of PROLOG programming) is repeat which is defined as follows:-

```
repeat.
repeat :- repeat.
```

Logic Extensions

Negation-by-failure (not), disjunction (Q ; R) and conditional alternatives (Q -> R; S) are all provided. A list-of-all-solutions primitive has been incorporated into later versions.

DEC10-PROLOG makes available a large assortment of metalogical primitives and several of these (including clause, assertz, functor, arg and call) were described earlier. These primitives provide considerable scope for metalogical programming. However this appears to be restricted in comparison to other systems such as micro-PROLOG in that DEC10-PROLOG does not permit predicates to be treated as 'first class' data objects. (Warren has resisted this extension, arguing that it would add nothing to the power of the language [93]).

Subsidiary Features

The DEC10-PROLOG provision for arithmetic (which is integer-only) and for input/output have already been outlined. A large number of other features are available and these have been documented by Byrd et al [125].

Implementation

The current DEC10-PROLOG system comprises an interpreter and a compiler (as described earlier) which co-resides with the interpreter in main store. Its implementation has been described by Warren [126, 127]. Much of the system has been developed in PROLOG with the aid of the bootstrapping technique. Many optimisations have been incorporated to conserve execution speed and (especially) memory consumption. An example is indexing: in storing clauses DEC10-PROLOG automatically constructs an index to memory locations based on the first arguments, in addition to the predicates, of clauses. This can provide substantial efficiency gains, especially when a large set of assertions for some predicate must be searched. The system (which omits the 'occur check') uses a special two-stack version of structure-sharing which facilitates space-saving.

2.2.2.4 LOGLISP

Overview

LOGLISP [70, 128] is an implementation of logic programming within LISP. It was developed by Robinson and Sibert at Syracuse University with the following main aims:-

1. To provide LISP users with a logic programming facility within the LISP environment with which they were highly familiar.

2. To demonstrate that logic programming need not be synonymous with bactracking execution in the style of PROLOG (and PLANNER).

LOGLISP is based on the procedural interpretation of Horn clauses using LUSH resolution and hence falls clearly within the logic programming computational view. Unlike traditional PROLOG systems however, it uses a breadth-first search strategy. Because of this and its relationship to LISP, it is of considerable interest. Unlike the three PROLOG systems described above however, LOGLISP appears to have no substantial community of users and indeed Robinson has described it as a 'laboratory device' [129]. The developers have described the system as providing 'a rich setting for logic programming' and they are now working on a new language which has the name 'Super LOGLISP'.

Syntax

LOGLISP consists of LISP together with a set of logic programming primitives implemented in LISP which are collectively referred to by the name LOGIC. In LOGIC all procedures, queries and other logic programming constructs are represented as LISP data-objects. For example the procedure which appears in Horn clause logic as

B <- A1 & ... & An

is entered into the LOGLISP system by typing the LISP procedure call

(:- B A1 ... An)

thereby invoking the LOGIC function ':-' which has been programmed to store the procedure (with indexing) in the user's database. (Robinson and Sibert [70] use a somewhat different terminology). The atoms B and Ai are written in LISP syntax, as exemplified by

(Likes x y)

with the convention that variables begin with lowercase letters. An example of a LOGLISP query form is

(ALL (x1 ... xt) C1 & ... & Cn)

which is actually a call on the LOGIC defined procedure ALL, which returns as its value a list of all the tuples (x1 ... xt) which satisfy the goal statement C1 & ... & Cn.

Control

LOGLISP computes solutions using LUSH resolution. The computation rule selects goals on a last-in first-out left-to-right basis, as in traditional PROLOG. The search strategy is breadth-first, with branches of the search tree being developed in quasi-parallel.
Several control mechanisms are provided by the system over the LOGLISP breadth-first search. They are provided in order to truncate execution where (for example) search trees have one or more branches of infinite length. The mechanisms take the form of a set of parameters which the authors of LOGLISP call the 'deduction window'. Each parameter enables a bound to be set on the search tree, such as the maximum branch length, the total number of nodes which may be developed, the maximum number of subgoals which may correspond to each node, the number of times in any one branch that rules are applied, and the number of times in any one branch that assertions are applied. Default values are provided for the deduction window, but programmers can overide these by annotating queries accordingly, as exemplified by

```
(ALL X P1 ... Pn RULES: 5  TREESIZE: 1000)
```

Logic Extensions

LOGLISP does not appear to provide any extensions to Horn clause logic. In compensation, the system's developers stress that in consequence of the embedded implementation the full power of LISP (very nearly) is available to users and LISP procedures can be invoked from within the user's programs. By illustration, negation-by-failure could be defined, if so desired, by

```
(NOT p) <- (NULL (ANY 1 T p))
```

which provides the LISP primitive NOT with the extra meaning of negation-by-failure. Here, ANY is a LOGIC-defined query primitive and NULL is the LISP predicate which tests for an empty list. It is a feature of the LOGLISP deduction cycle that LISP expressions are replaced by their reductions according to their LISP meanings.

Subsidiary Features

LOGLISP users inherit the facilities of LISP, including its arithmetic and input/output capabilities. In particular the answer to a query is a LISP data object and as such it can be arbitrarily processed.

Implementation

LOGLISP is a structure-sharing system, although the implementers describe some specialisation of the basic Boyer-Moore technique which was necessitated by the requirement for rapid access to variable bindings arising from the quasi-parallel execution strategy [70]. The unification algorithm appears to omit the occur check. The system is claimed to achieve an execution speed of around 150 unifications per second, or roughly one-sixth of the speed of DEC10-PROLOG running in its interpretive mode, but the developers believe that they will be able to improve this by a factor of at least ten. Unfortunately, no documented statistics on the memory consumption of LOGLISP are known.

3 LOGIC PROGRAMMING FOR SOFTWARE DEVELOPMENT

Part One of this thesis identified some major current problems of software development. It was argued that programs in traditional languages suffered from referential opacity, that the predominance of informal development tools and methods has impeded the efficient development of verifiable software, that programming languages have been insufficiently problem-oriented and are becoming over-complex, and that imperative programming styles lack scope for exploiting parallel computer architectures. This section considers what logic programming might have to offer to the solution of these problems. Where logic programming in its present state appears inadequate, an attempt is made to suggest what if anything can be done to close the gap.

3.1 Logic as a uniform software formalism

The multiplicity and informality of the commonly used tools and methods for software development were identified earlier as serious problems. The utilisation of different languages and notations at each of the specification, design and implementation stages of development was recognised to introduce sources of error: instead of being a coherent single process, software production depends on a sequence of stages each having its own methods and tools. In consequence fracture points are created. Uniform formal methods, which should assist with the development and verification of software, are seldom applied.
An important claim made for logic as a language is its versatility as a formalism. Hogger [87] argues that

> 'Logic can be used to represent data, programs, specifications and the relationships between them; it can be used for both object-level and metalevel descriptions; it can be used to describe the management of software as well as the software itself'.

Lloyd [119] also emphasises the versatility of logic:-

> '...logic thus provides a single formalism for diverse parts of computer science. Logic provides us with a general-purpose, problem-solving language, a concurrent language suitable for operating systems and also a foundation for database systems.'

Kowalski [101] argues in particular that:-

> 'The suitability of logic for expressing both programs and their specifications make it especially useful for program development'.

Thus, logic is proposed as a uniform formalism tool for software development. In what follows, this proposal is considered in detail.

3.1.1 The development process with logic

The development process as outlined by Kowalski (op. cit.) has two essential stages:-

> 1. (Specification stage) The problem to be solved and the information which is needed for the solution are specified. The final specification obtained should be in formal logic, although successive formulations starting from imprecise natural language may be required to achieve this.
> The specification should now be tested using a suitable logic interpreter. An incomplete or incorrect specification can be altered and tried again.
>
> 2. (Efficiency stage) Inefficiencies apparent in the specification are identified and removed, producing an effective program.

In comparison with the complexity of traditional 'structured programming' development methodologies, this process appears rather straightforward. It seems that the only difference between specifications expressed in logic and finished programs rests with efficiency! Elsewhere, Kowalski confirms that this is his conclusion:-

'Logic sufficiently blurs the distinction between program and specification that many logic programs can just as well be regarded as executable specifications. On the one hand, this can give the impression that logic programming lacks a programming methodology; on the other, it may imply that many of the software engineering techniques that have been developed for conventional programming languages are inapplicable and unnecessary for logic programs'.[40]

3.1.2 Logic specifications

One possible criticism concerns the requirement that specifications should be expressed in formal logic. It might be argued that, since specifications originate (at least partly) from users, logic is not sufficiently natural and that it is too difficult to use. To some extent this criticism is answered by the derivation of the logic version of the specification by successive formulations starting from natural language. Although formal methods cannot be used to ensure the correctness of the early (pre-logical) formulations, there is the advantage that as soon as a logic version is written it can be executed and tested against expectations. Should the specification turn out to be incorrect, either in the sense of containing inconsistencies or else in the sense that it does not specify a solution to the problem which needs to be solved, then it can be remedied at that stage. Arguably this is a big improvement over a more conventional software life cycle which fails to test specifications operationally until finished programs have been implemented.
The suggestion that specifications should be expressed in formal languages, including logic, is not new. Sommerville [16] notes several formal notations (such as PSL/PSA, RSL and SADT) which have been developed for the purpose of expressing specifications. In some of these languages statements are machine-processable, but of course they are not programming languages. Logic has been used as a specification language quite commonly, for example by Floyd [24], Manna [133], Hoare [134] and Dijkstra [135]. It is worth noting also that logic has been used as a language for formal description by mathematicians for a very long time.
Logic specifications can only be executable, however, in the presence of logic programming systems which can interpret the form of logic used. As Part Two of this thesis has shown, most current logic interpreters can cope only with a

fairly small subset of logic based on Horn clauses. An example of a Horn clause specification of a significantly complex application is presented by Davis [136], who shows how a graphical display interface can be specified by means of Horn clauses. Her specification is both highly readable and directly 'runnable', enabling it to be easily debugged. More commonly, however, it is argued that Horn clauses are not expressive enough for representing specifications and that something nearer to the syntactic freedom and expressiveness of full first-order logic is required. Thus, specifications are provided in the formal language of logic, but the advantage of being able to execute them directly is lost, at least with current logic programming systems.

This observation may suggest the goal of building logic interpreters which support a larger fragment of logic. Indeed, some progress has been made towards this end, as illustrated by the discussion in Part Two of extensions to Horn clause form. Today the majority of PROLOG systems augment the language of Horn clauses at least to the extent of implementing some form of negation and, generally, sets of solutions. However, implementing even these modest extensions has been problematic (as was shown earlier). The great assets of the procedural interpretation of Horn clauses through SLD-resolution lie in its soundness and (with a fair search rule) its completeness: implementing extensions whilst preserving those assets has been shown to be no easy task. Fortunately, the efficiency with which such an extended logic system would execute specifications written in this enlarged language would not be of too great a concern from a specifications-testing point of view.

As an alternative (or rather, as a complement) to building logic interpreters which can process logic at a high enough level to execute specifications, researchers have investigated means of transforming specifications into a form of language compatible with existing interpreters. From a specification written in full first-order logic, the aim is to use logical implication to derive an executable (say, Horn clause) program. This method, which is sometimes called the synthesis of a logic program from its specification, guarantees the soundness of the derived program. ('Soundness' here is intended to mean the property whereby every relation which holds in the program also holds in the specification: this property is often referred to as partial correctness. Note that soundness is a property of the logic only and is independent of any control which an actual logic interpreter might use). Synthesis of logic programs has been investigated by Clark [137], by Clark and Sickel [138] and by Hogger [139, 140]. Clark (op. cit.) describes the aim of synthesis as being to derive from the specification the set of computationally useful Horn clauses, that is, those clauses which can be used to generate all the instances of the relation which is to be computed. He shows in particular how recursive programs can be derived from non-recursive specifications, and how tail recursive programs can be

derived. If the synthesis is directed such that the definition of a relation in the specification is fully equivalent to the relation's definition in the derived clauses, then in addition to soundness the synthesised program has the property of being complete with respect to that relation. (A program which is both sound and complete with respect to its specification is often termed totally correct).

Generally, the logic specifications presented in the literature describe relatively simple relations. For example, Clark discusses examples of a list membership predicate, a relation describing ordered insertion with binary trees, the factorial relation, and so on. In each example it is straightforward to present logic specifications and the synthesis of Horn clause programs from them is fairly easy. However, it can be perceived that the gap between such examples and the kind of problems typically encountered in production programming is rather large. It seems much less likely that with a large and complex problem, which is possibly poorly perceived initially, a satisfactory full set of specifications in formal logic can be so easily developed. Even if this could be accomplished, however, the synthesis into executable programs would presumably require a further non-trivial effort before any testing could becomae possible. Note that this potentially concedes a major inefficiency, in that weaknesses in the specification may now go undetected until a point long after they have been introduced.

In the interests of detecting specification errors at the earliest possible stage, the steps of program specification (using formal logic) and program synthesis (deriving a version in the executable subset of logic) should proceed together. A methodology which follows this proposal, called top-down logic programming, is described next.

3.1.3 Top-down logic programming

A top-down methodology for the development of logic programs is (recursively) described in the following steps.

1. The required program is first identified with some predicate r, say. A (full first-order) logic specification S of r is obtained in terms of a set of predicates, say r1, r2, ..., rk. However, r1, r2, ..., rk are not themselves formally specified at this stage.

2. A Horn clause (or other executable) program P for r is synthesised from S using the predicates r1, r2, ..., rk. At this point, P is executed with 'dummy' procedures set up to simulate the expected meaning of these predicates, enabling the specification S to be tested and improved if necessary.

3. For each predicate ri,

   a. The predicate ri is given a formal logic specification Si in terms of some set of predicates ri1, ri2, ..., rik, which are not themselves formally specified at this stage.

   b. A Horn clause program for ri is synthesised from Si using the predicates ri1, ri2, ..., rik. This program is substituted in P for the previous 'dummy' procedures, enabling the expanded specification (SuSiu...uSi) to be tested and improved if necessary.

   c. If k>0, then each predicate ri1, ..., rik is developed by top-down logic programming.

The methodology closely resembles the 'stepwise refinement' process of traditional structured programming [15]. A crucial difference here, however, is that only the logic component of the program is being developed. Consideration of the control component (beyond the selection of an autonomous control strategy to enable testing) is deferred to the subsequent, efficiency-improving stage of program development. Furthermore, the use of synthesis to derive each predicate in the program from its specification guarantees the soundness of the whole program, whilst the use of 'dummy' procedures permits the earliest execution and testing of the specifications (effectively, this is top-down development with top-down testing of specifications). A comparable process for top-down programming has been described (although not specifically in the context of logic programming) in a paper by Hoare, who proposes that 'a computer program can be identified with the strongest predicate describing all relevant observations that can be made of a computer executing the program' [141]. This 'strongest predicate' corresponds to the 'top-level' predicate r in the description given above. In the context of logic programming, Hogger [87] has described a somewhat

similar top-down development methodology, goal-directed derivation, to the one given here: it may be observed that the process described above is 'goal-directed' in the sense that the implied use of the program will proceed from a call to the top-level predicate.

3.1.4 Abstraction and modularity with logic

In Part One it was argued that abstraction and modularity were the two major principles which the 'structured programming' school contributed to the attempt to solve the problem of software construction. It is interesting, then, to consider the significance of these two concepts for logic programming.

3.1.4.1 Abstraction

Dijkstra has argued that the fundamental key to effective programming lies with the 'separation of concerns' [155], and the structured programming methodology which he did much to develop encourages this separation in various ways. Programmers are encouraged to formulate a highly abstract version of an algorithm in the first instance, and in a modern imperative language this can be implemented as a top-level procedure. Sub-procedures can be developed separately from one another in hierarchies of abstraction. Data structures can be separated from procedures, textually as well as conceptually, and the methodology recommends that they too should be developed in stages proceeding from an initially abstract representation.
A significant claim which can be made for logic programming is that it accepts the 'separation of concerns' philosophy, but that it takes it much further in advocating that the logic component of an algorithm should be separated from the control component. A logic programmer's first version of a program is merely a specification of the information which is needed to solve the problem: it is more abstract than the imperative programmer's first version because it does not specify how the computer should actually use the information. This argument is made by Kriwaczek, for example, who suggests that, although efficiency considerations can often demand a pragmatic approach,

> '... it still remains true that the final specification, written in the form of a PROLOG program, deals with the subject area of the problem. A program

> in Pascal, BASIC or COBOL deals with the workings of a machine, however abstract it may be. In this sense, PROLOG can be seen as a further step in the evolution of programming languages from low-level machine orientation to higher-level problem and user orientation.' [146]

Kowalski has presented the case for the separation of logic from control by likening it to the arguments (which are already generally accepted) for the separation within conventional programming of procedures from data structures [45]. He points out that the latter separation enables the functions which the data structures perform to be distinguished from the manner in which they perform them. Furthermore, the separation facilitates the improvement of algorithms by replacing inefficient data structures with efficient ones. Similarly, when logic is separate from control it is possible to distinguish what the algorithm does, as determined by the logic component, from the manner in which it is done, as determined by the control component. An algorithm can be improved by replacing an inefficient control strategy with a more efficient one.

The utility of logic as a language for abstracting and expressing relationships between objects has long been known to mathematicians. If computer programs are predicates, as has been argued recently by Hoare [141], then predicate logic would seem to be an obvious means by which to represent them. Furthermore, variables in logic programming appear to offer a powerful tool of abstraction, as Warren [147] notes in the context of using PROLOG for compiler writing. The ability to 'name' arbitrary PROLOG data structures (which may be constants, numbers, lists, or functional terms) with a single logical variable, and to use variables to both construct and dissect data structures with total flexibility, is certainly valuable. Furthermore, data structures can be manipulated using unification without the need for programmers to consider the instantiation states of the variables which the structures contain. These advantages appear to accrue from a realisation of variables which is conceptually close to their intuitive mathematical meaning, as opposed to their implementation as named storage locations in imperative programming languages.

In some minor ways however, modern imperative languages give support for abstraction which is not always equalled by existing logic programming systems. A trivial example is the freedom to use full-length names as identifiers, which is offered by almost all high-level programming languages in current use. The importance of this elementary facility is widely recognised, and so it is disappointing to find that the micro-PROLOG system limits variable names to a single letter possibly followed by a sequence of digits,

a convention which is even more impoverished than some versions of BASIC. Another freedom which is almost taken for granted in imperative languages is that of expressing relationships in functional notation, where appropriate: the absence of this freedom in most current PROLOG systems can hinder the expression of abstract relationships in a suitably natural form.

3.1.4.2 Modularity

Modularity was discovered by the 'structured programming' school to be a crucial concept for effective programming, particularly of large systems. The use of abstraction facilitated the development of programs by a succession of refinements into a (possibly large) number of component procedures. In order that the components functioned correctly within the overall program, without interfering with one another, it was essential that they could be implemented as secure independent modules, each with a well-defined interface to its exterior. The interface served to document the allowable uses of the component and it enabled the program compiler or executor to detect and diagnose illegal uses. Such modularity also benefited the management of the program's conceptual complexity, for individual modules could be used by referring only to their interface descriptions without the need to know how the module worked. The main implementation unit of modularity in modern programming languages has been the procedure, the heading of which provides the interface description. In some languages there is also provision for a 'lower' level of modularity (an example is the textual nesting of procedures in Pascal) whilst others facilitate a 'higher' level (for example, the modules and packages of Modula-2 and Ada respectively).

It can be observed that a degree of modularity is inherent within the language of logic. A Horn clause for example is a modular component insofar as it can be read declaratively, and its procedural interpretation can be grasped, without reference to any other Horn clause. Furthermore, the variables of a Horn clause are entirely 'local' to it and the same variable names can be re-used within different Horn clauses without fear of any clash. The interface between a Horn clause and its exterior is specified by the clause head, and the unification mechanism provides a form of check that the clause will not be used to solve goals which are ill-matched it.

Although the modularity inherent within Horn clause logic is valuable, it is limited in several ways which, whilst they may be unimportant with small programs, are likely to be significant when logic is used for sizeable software projects. A critical weakness would seem to be that predicate symbols are 'global': the meaning of a predicate

symbol is determined by the set of clauses which give its definition, and therefore to ascertain the meaning potentially requires an inspection of the entire program. Consequently, as many PROLOG programmers have found to their cost, to accidentally re-use a predicate symbol within a logic program can be disastrous. Likewise, constant and functor symbols have global scope and clashes can occur when they are passed between clauses by unification.
Unfortunately, it is also true that the mechanism of unification has a number of characteristics which make clause heads alone inadequate as interfaces for truly modular components. First, unification provides only a syntactic and not a semantic check. It does not prevent the standard definition of the list concatenation relation

```
append(( ) X X)
append((x¦X) Y (x¦Z)) if append(X Y Z)
```

from allowing the goal append(( ) 3 3) to succeed, for example, even although this involves a mis-use of the append program which (as its second clause indicates) expects the type of its arguments to be lists. Second, and indeed as the append example shows, the mechanism is not explicit: to determine the allowable uses of a clause, it is not usually sufficient just to inspect its head. And third, unification does not provide information on causes of failure: a goal which has been observed to fail may either have done so because it failed to unify with any clause, or alternatively it may have unified but produced only unsolvable sub-goals. Hence, scrutinised by the criteria of modularity, logic programs appear to have an interface which is insecure and opaque, and their run-time behaviour is unhelpful.
More positively, however, it can be argued that the inherent modularity of logic is useful for small programs but that it needs to be enhanced by some explicit module construction facility for larger projects. It seems reasonable to look for a module facility which would offer logic programmers much the same kind of benefit that modules currently offer to users of Modula-2 or Ada. Unfortunately, modules appear not to have been much discussed within the context of logic programming. Two of the few PROLOG systems that do make any provision for them are the Hungarian M-PROLOG [148] system and micro-PROLOG [94]. The facility for module construction in micro-PROLOG has already been described: it will be recalled that a micro-PROLOG module is a named collection of relation definitions which communicates with other programs via import/export lists. A system of local dictionaries ensures that names which are used in the module but which are not in the import/export lists are local to the module. Although the micro-PROLOG module facility has proved to be of great practical worth (certainly, in the experience of this researcher), at least three significant criticisms of it can be made. First, the interface between a micro-PROLOG module and its exterior does not document the allowable uses

of the module's exported relations. To discover that information, it is necessary to examine the inner functioning of the module and to some extent this defeats the purpose of having modules. Second, unless the programmer has incorporated suitable checks explicitly, modules are not secure against incorrect usages and these will not generally be detected by the system. The third criticism, and perhaps the most serious, is that any construction of this kind, which makes the truth of certain relationships dependent on their textual locations, appears to have no obvious logical foundations. However worthwhile the facility may be in pragmatic terms, it appears to complicate the semantics of logic programs in a way which lacks a logical justification. An implementation of modules which is perhaps less problematic semantically may however follow from the proposals of Bowen and Kowalski for the amalgamation of object language and metalanguage [92]. They touch on the topic of modularity at the end of their paper. Instead of directly executing a goal for a relation which is defined within a constructed module, they propose that a metapredicate would be called with the goal and the collection of defining clauses named as parameters. The logical basis of this is the view of the defining clauses as comprising a specific 'theory' which may be invoked for some arguments but not for others. In this scheme therefore, modules would comprise named groups of clauses each denoting distinct theories. Kowalski and Bowen suggest that their proposal bears much fuller investigation. A recent report by Cuadrado and Cuadrado claims that Bowen has made substantial progress in implementing the idea and says that 'he has painstakingly avoided straying outside the realms of logic' [149].

3.1.4.3 Typing and mode information

The considerations above suggest that a secure implementation of modules in logic programming may require that types be attributed to data and to predicates. This is a question with significant implications and it will be appropriate to consider it here.
The example of PROLOG is useful. PROLOG is an untyped language in the sense that there are no type declarations and each variable in a clause can potentially become bound without restriction to any term. Syntactically a term can be either a variable, a number or a constant, which are the unstructured terms, or a functor term or a list, which are the structured terms. Functor terms are usually interpreted as records of the type given by the functor (an interesting discussion on the relationship between list and record data in PROLOG is offered by Campbell and Hardy [150]). Predicates are 'untyped' also in that the types of their arguments are not declared, although an implicit typing is

usually discernible from an inspection of the defining clauses.
It could be said that in lacking types, PROLOG continues the tradition of languages associated with artificial intelligence applications. LISP, POP-2 and APL are all (generally) interpreted and untyped languages. On the other hand, as Davies [83] points out the tradition is already broken because POPLER 1.5, PLANNER and SAIL are all typed languages. Davies describes how the introduction of typed variables in POPLER 1.5 accompanied the development of an interpreter for the language and observes that the issue cannot be simply decided on the grounds that a requirement of a language is that it be incrementally interpreted. (Recall that PROLOG is usually interpreted but that Warren's DEC10 implementation has demonstrated that compilation is a useful and feasible option). Davies reports that, in some cases, the addition of typing to POPLER improved efficiency and increased program understandability but he adds that 'the issue of whether to type variables is potentially controversial'.
Both Davies (op. cit.) and Hogger [87] point out that programmers are free within untyped languages to construct their own explicit type checks where necessary. An example might be to re-write the first append clause as

```
append(( ) x x) if list(x)
```

where the definition of list might be

```
list(( ))
list((x¦X))
```

A check of this kind will prevent a goal such as append(( ) 3 3) from succeeding, although it will not by itself generate an error message and the subsequent failure could be difficult to diagnose. In any case, the efficiency of the append program will be worsened. Furthermore, a compiler cannot use this sort of type-check to catch errors, which will go untrapped until the program is run.
It is significant that the trend in conventional software engineering has been towards typed languages. Pascal, Modula-2 and Ada are compiled languages which are all strongly typed, and the consensus of opinion among software engineers favours typing because it improves the discipline and expressiveness of programmers. Typing also enhances the readibility of programs and makes reasoning about programs easier. Furthermore, in most cases typing provides program compilers with information which facilitates compilation and improves execution performance. Two points which are stressed most frequently are that typing benefits program security and assists the efficient development of software. By increasing the range of errors which can be trapped at compile-time, users are protected from harmful consequences and fault-finding in the maintainance stage of the software

life-cycle, where it is usually more expensive, is lessened. Essentially, although typing places restrictions on what programmers can do and although there are administrative overheads in the need to make type declarations in programs, the experience of software developers appears to be that typing is well worthwhile. (It has been said that the restrictions are on what mistakes can be legally made, which seems to be a useful sort of restriction).
However, the example of Pascal has shown that it is possible for typing be over-restrictive. A Pascal function which computes the sum of two integers is of no use for finding the sum of two real numbers. Instead of being able to define one function which can operate with different parameter types (a so-called polymorphic function), the Pascal programmer must define one function for each choice of parameter types. (It is not the point that the problem can sometimes be circumvented). Recently, however, it has been shown that the problem of polymorphism can be largely overcome by the introduction of type variables each of which can represent a single, but unspecified, data type. An example of a language which permits the definition of polymorphic functions through the use of type variables is the functional language HOPE [151]. A theoretical interpretation of polymorphism has been published by Milner [152].
Recently a detailed proposal has been advanced by Mycroft and O'Keefe for a type system for PROLOG [153]. They argue that untyped PROLOG is useful for learning PROLOG and for rapid prototype construction, but that the lack of typing is a serious deficiency for building large systems. A PROLOG with typing will enable many errors to be trapped and will facilitate the secure representation of modules. The researchers argue that theorem provers which reason about PROLOG programs could be made more powerful with the aid of type information and that an efficiency gain could be achieved. Finally, type declarations provide documentation which facilitates human understanding of programs. It can be observed that these arguments are very similar to the arguments for typing within conventional languages.
The Mycroft/O'Keefe scheme is polymorphic. Types are defined by

```
Type ::= Tvar ¦ Tcons(Type)
```

where Tvar denotes a type variable and Tcons denotes a type constructor. Thus, assuming that Tvar includes the A, B and C as type variables, and that Tcons includes the nullary constructor int and the unary constructor list, example types are

```
A, int, list(A), list(int), list(list(int))
```

A PROLOG program will be supplied with declarations giving the types of each predicate and of each functor. The type of

the functor list could be declared by

```
type list(A) --> [] ; [A list(A)]
```

and the type of the standard predicate append could be declared by

```
pred append(list(A), list(A), list(A))
```

Variables do not need to be typed explicitly because their types can be determined by the type information supplied for functors and predicates. The researchers show that their scheme has the essential property that in a well-typed program, no predicate is ever applied during execution to arguments of unsuitable types. They have implemented the system in the form of a DEC10-PROLOG program and they observe that a satisfying consequence of the invertibility of logic programs is that, in addition to its expected use of verifying that a program is well-typed, it can also determine the type of a given program.

The proposals of Mycroft and O'Keefe have several attractive features. In particular it can be noted that the only additions to the PROLOG language itself are type declarations, and that, since a well-typed program will behave identically with or without these declarations present, the effect on programming style is slight and there is no loss of program portability. However, their implementation cannot improve the efficiency of a compiled program, since the DEC10-PROLOG compiler is independent of the type-checking program: efficiency considerations strongly suggest that type checking should be built into the compiler. Nor can it cope with metalanguage, although the authors indicate possible extensions both in this direction and towards the provision of abstract data types. (An abstract data type is a type definition along with a collection of routines that may be used to manipulate details of the type. The implementation of this is hidden from the user. Ada and HOPE are examples of languages which provide this kind of facility).

It seems appropriate at this point to consider the provision of mode information within logic programs. The idea of supplying mode information for a given procedure is to specify which of the procedure's parameters will supply input data to the call, and which will as a result of the call become instantiated with output data. (It can be noted that unlike type information, mode information does not attempt to describe the actual composition of the data). The possible relevance of mode information arises from the fact that although the logic which defines a predicate makes no commitment as to the predicate's mode of call, in practice control considerations may severely constrain the modes which are computationally useful. In fact PROLOG programmers frequently write definitions which they know to be intolerably inefficient, or even quite inapplicable, beyond

one specific intended mode of use. A typical underlying cause is the inclusion within the definition of negation by failure, an arithmetic predicate, an input/output primitive, or some other system primitive with procedural restrictions. It seems highly unrealistic to expect that programmers should always supply such definitions for predicates as can support every possible mode of call. However, the situation in which PROLOG programmers can write programs which make undocumented mode assumptions (and where the system is unable to diagnose errors which arise from breaches of those assumptions) is surely highly undesirable.
One proposed solution to the problem requires that mode information be given explicitly for each predicate which specifies the expected forms of use. There are already several logic programming precedents for this proposal. The DEC10-PROLOG compiler recognises mode declarations (although it makes them optional) such as:-

```
mode subset(+, -)
```

which says that every call to the subset procedure will have a non-variable term in the first, and a variable in the second, argument place. The compiler uses this information to optimise code for efficiency. The IC-PROLOG system permits annotations to be written on the head of a clause, as exemplified by

```
subset(x?, y^) if .......
```

which also provides a form of mode information, and this generates a run-time check, although of course since an IC-PROLOG program is interpreted and not compiled there is no efficiency advantage. The PARLOG parallel logic programming language [218] expresses mode information in the form

```
mode subset(x?, y^)
```

which looks like a hybrid of the other two constructs.
The case for enabling both mode and type information to be provided explicitly within logic programs is strong. In the case of type information, the arguments in its favour are essentially the same as those for imperative languages. Mode information is less significant for imperative languages, the deterministic procedures of which support only one mode of use (although a form of security against mode abuse is present in that, for example, a Pascal compiler will detect errors where a variable formal parameter is passed a constant value). However it seems likely that in logic programming systems, explicit mode information can improve program transparency, security and efficiency. It is tempting to wonder whether the two forms of declaration can be combined, perhaps in some scheme

exemplified by:

type subset(+list(A), -list(A))

(following from the type proposal of Mycroft and O'Keefe and the mode declarations of DEC10-PROLOG). More research is needed to investigate this question and others, such as the issue of how to employ type and mode information to provide a secure and logically justifiable module facility for logic programming systems.

3.1.5 Realising efficiency

A logic program which has been synthesised from logic specifications is guaranteed to be sound (because the specification then logically implies the program). If the synthesis has been such as to ensure also that the program implies the specification, then the program will also be complete with respect to the specification. The soundness and completeness of (for example) the SLD-resolution inference procedure then ensures that the program, when executed by a logic interpreter which fairly implements SLD-resolution, will deliver all the answers which are required of it (completeness), without producing any wrong answers (soundness). Hence, having developed sound and correct logic, it should be possible to concentrate wholly on the problem of realising adequate efficiency.
Unfortunately, as Part Two has shown there are several reasons why existing logic interpreters can behave with such 'extreme inefficiency' that no answers are produced at all by its execution, even when the program being executed is proven to be sound and complete with respect to its specification and even when logically computable solutions to the goal do exist. In particular,

1. Many logic interpreters do not implement SLD-resolution fairly. For example, those which (like PROLOG) have an unfair depth-first search strategy lose completeness when execution descends down an infinite branch of the search tree before solutions on other branches have been discovered.

2. Completeness is sometimes lost when logic extensions to the Horn clause form, such as negation, are executed by logic interpreters which implement only a limited version of these extensions.

There is no doubt that these are serious problems, and in fact they have led some programmers to conclude that logic

programming in any true sense is simply not practical at the present time. Some of these continue to use PROLOG whilst abandoning its logical foundations, treating it merely as a convenient high-level, procedural language in which extra-logical features for instance can be used quite pragmatically (see for example, McDermott [142]). However, a more hopeful view is presented by Kowalski [40] who suggests that logic programs which execute inefficiently under PROLOG (including those which exhibit extreme inefficiency) should be transformed in a correctness-preserving way so that efficient behaviour is realised without recourse to the extralogical features of the language. The resulting program will still be logically defensible. Kowalski also recalls the longer-term aim of developing better logic programming languages than PROLOG, and the previous sections have described much that is relevant to this aim.

One question which arises concerns how early in the development of a logic program should the inadequacies of the intended actual logic interpreter be recognised. It seems unwise to wait until the entire development is complete before considering what logic transformations are required for acceptable efficiency with the control strategy which is to be used, since it may transpire that major changes are needed to avoid extreme inefficiency. Instead, it may be better to include efficiency as a factor which influences the development of each part of the program. Thus, the synthesis of each predicate from its specification could be steered towards deriving the Horn clause programs which are computationally useful and which can be executed with acceptable efficiency by the intended interpreter with goals of the type that the programs will be expected to solve.

In addition to seeking efficiency gains by transforming the logic of a program, it is possible to improve efficiency by transforming the control. As Kowalski [45] has pointed out, different algorithmic behaviours can result by applying different control strategies to the same logic, but (providing that the control is always sound and complete) the algorithms are equivalent in the sense that they solve the same problems with the same results. Symbolically:-

```
If   A1 = L + C1   and
     A2 = L + C2
then A1 and A2 are equivalent
```

An impressive demonstration of the practical utility of this relationship has been provided by Clark, McKeeman and Sickel [59]. They define a logic program for numerical integration which is then paired with several different control strategies, so that several equivalent algorithms (in problem-solving capability) are formed with different behaviours. They then show how to transform each pairing of

logic with control into a new pairing, in which the logic is elaborated in order that the control can be simplified to one which is easily available on the IC-PROLOG interpreter. An interesting example of improving efficiency by a logic transformation alone is provided by Hansson and Tarnlund, who show how to transform a logic program which manipulates simple lists into an equivalent program which manipulates difference lists [143].
In the longer term, the general efficiency of logic programming systems will depend on advances in the solution of the control problem, and this is discussed next. Separate consideration will be given later to the issue of parallel control strategies, since many logic programming researchers now see these as being especially important. It is certainly true that some impressive performances have been acheived by PROLOG implementations to date, especially by Warren's DEC10-PROLOG and the more recent Quintus PROLOG system running on DEC VAX machines [145], and PROLOG efficiency has been sufficient to enable its use in some significant applications, as will be described later. In general, however, it is believed that logic programs cannot really expect (with current hardware) to match the efficiency which can be obtained by using an imperative language, since as Part One showed these languages are much more tightly geared to the low-level operations of the von Neumann computing machine.

3.1.6 The Control Problem

At this point, a more general consideration of the control problem for logic programming will be useful. Control can be considered in the two categories of autonomous control and programmer-specified control respectively and these are treated separately below.

3.1.6.1 Autonomous control

As described in Part Two, the main control strategy which is used for executing Horn clauses as programs is SLD-resolution. The PROLOG implementation of SLD-resolution uses a last-in first-out left-to-right computation rule and a depth-first top-to-bottom search rule, but this is only one choice. The previous section considered various ways in which better computation and search strategies might be obtained, and it also identified the opportunities for parallel execution. Discussion of parallelism will be left

until later and the discussion below relates primarily to sequential control strategy.

It will be recalled that the computation strategy specifies the order of calls in a conjunctive goal statement. As has been noted previously, there is substantial scope for implementing an interpreter with a sophisticated strategy for selecting goals which would behave better in the majority of cases than the standard fixed strategy. A drawback would be that its behaviour might be correspondingly more difficult to understand. Furthermore, even with this more efficient hypothetical interpreter there would presumably remain some cases where execution efficiency could be improved by programmer-specified control: but it can be anticipated that the more complex default strategy would hinder this effort. These objections may be pessimistic, however, and practical test-bed implementations of different strategies would be worthwhile. Speculatively, a compromise may be to provide systems which have a compiler switch which can set the computation-rule either to 'clever' or 'standard'.

The second aspect of autonomous control is the search strategy. The choice of the standard depth-first strategy is usually defended on efficiency grounds, but as has been seen above it can suffer disastrously from 'extreme inefficiency' when infinite branches are encountered. It should be possible to reduce some of the milder forms of inefficiency of the depth-first approach with some form of intelligent backtracking, as described previously: this would make execution a little harder to predict, but the price should be small and there is no threat to completeness. More radically, a breadth-frist strategy could be substituted for a depth-first one. Unlike depth-first interpreters, a breadth-first system does not suffer loss of completeness on account of infinite branches and the LOGLISP system described earlier surely indicates that the breadth-first search strategy merits further investigation. However, infinite branches still present breadth-first systems with the problem of detecting when to terminate execution. As Kowalski has pointed out [45], Church's result on the undecidability of logic [46] implies that no logic interpreter can recognise all situations in which a goal is insolvable, so that logic programming will never banish the termination problem without paying some price. More positively however, Church's result can be viewed as saying that there is no 'best' theorem-prover and there is no limit to the extent to which the ability by which a logic interpreter can detect infinite branches may be improved. An example of a recent success using this result is reported by Brough and Walker [144]. They describe two modified depth-first interpreters which, using different checking criteria, will terminate a branch of the search tree on which a loop is detected. They show that both interpreters are better than standard PROLOG, in the sense that they both produce the correct solutions and halt for a larger class of

simple problems than does PROLOG, but that neither is better than the other. They conclude that interpreters with a bottom-up inference component appear promising.
Where logic systems implement extensions to Horn clause logic, it is obviously desirable that these extensions can be properly and efficiently executed within programs. In the case of negation, this problem has yet to be solved for, as has been shown, negation-by-failure whilst efficient operationally incurs the risk of losing completeness. Nevertheless, most programmers would probably agree that a negation operator is worth having even at this price because the gain in expressiveness outweighs the disadvantage of having to be aware of its control limitations. Similar considerations have been applied to other extensions, such as implication conditions, sets of solutions and arithmetic primitives. A problem for the future must be to eliminate the control limitations on these extensions as far as possible, so that they implement with acceptable efficiency a much closer representation of the logical relationships which they compute.

3.1.6.2 Programmer-specified control

Part Two has described various mechanisms which have been developed to date whereby programmers can condition the control of logical inference. To an extent programmers must rely upon these mechanisms to obtain acceptable efficiency from their programs. The difficulty is to determine which mechanisms should be made available in logic programming systems, and how programmers should make use of them.
Whilst the importance of the problem should not be underestimated, neither should it be exaggerated. Lloyd [119] provides a useful reminder that although logic interpreters are resolution theorem proving systems, they generally do not face anything like the computational complexity that a theorem prover in, say, group theory has to contend with. He suggests that many logic programs are almost determinate. However, it seems to be the case that some indisciplined programmers look to control mechanisms to restrain their excessively non-deterministic logic programs. Hogger [87] for example notes that, instead of thinking carefully about how to describe the relation which is really desired, programmers can be tempted into writing a possibly much easier description of a super-relation of it with subsequent use of control (in PROLOG, generally the 'cut') to achieve the desired results. There is a worse possible abuse of control mechanisms, however: it is possible to write unsound logic but with the use of such control as to suppress the production of incorrect answers. In spite of the possibility of abuse, however, it is generally recognised that logic programming systems have to provide

some form of programmer-specified control mechanisms to compensate for the deficiencies of existing autonomous default control strategies.
The four main forms of control are recalled below. Each mechanism is identified with an example, and a discussion of its positive and negative features is provided.

1. Pragmatic control: programmers tailor their program to the known control strategy of the interpreter. A notable example is the textual ordering of conditions and clauses to exploit the PROLOG control strategy. Thus control is made implicit within the logic.
On the positive side, this form of control is easy to implement and programmers seem to find it reasonably natural to use. Furthermore, it does not introduce any additional risk to completeness. Three main criticisms can be made. First, because pragmatic control is implicit the efficiency derived depends on the continued use of the target interpreter. Second, pragmatic control is known to tempt programmers into non-logical approaches. Third, and most serious perhaps, is the criticism that pragmatic control is simply not flexible enough to produce the desired algorithmic behaviour from a fixed logic program.

2. Control primitives: these usually appear syntactically as atoms within clauses, but their sole purpose is to modify control. The best known example is the PROLOG 'cut'.
Control primitives provide a form of control which is explicit within the logic. On the positive side, the 'cut' is easy to implement and certainly it can be used to improve the efficiency of certain programs. There are several significant drawbacks. First, although it is possible to argue that the cut can be used without affecting the logical reading of programs, as a form of syntactic pollution it can hardly be said to help. Second, like pragmatic control the cut is known to tempt programmers into non-logical approaches. Third, the experience of PROLOG has shown that the cut is difficult to use effectively. Fourth, it provides only a very limited type of control. Fifth, programs which make use of such an operator appear to make the assumption of a depth-first sequential logic interpreter. Finally, arguably the most serious criticism is that the cut can cause a loss of completeness (where successful branches of the search tree are pruned away accidentally).

3. Control annotations: these are syntactic markings on the program text to indicate control requirements. They have been introduced into logic programming chiefly by the IC-PROLOG system, as described earlier.
Control annotations provide a form of control which is explicit in the logic. The IC-PROLOG system has demonstrated the feasibility of implementing several types of annotation, and it has shown that they can be used flexibly to produce sophisticated algorithmic behaviour, including coroutining. They appear to support the methodology of logic programming which was outlined earlier, in that programs can be developed and tested as executable specifications and later 'marked up' with annotations to improve efficiency. Furthermore, it can be hoped that the annotations which are implemented can be such that program completeness is never worsened by their application. On the negative side, control annotations can hardly be said to improve the appearance of programs. Furthermore, their effective deployment presumably demands a fair amount of skill. Finally, it is not really clear how to decide which set of annotations should be made available, and it must be doubtful that these annotations will be appropriate for a logic interpreter which uses some form of parallel execution.

4. Metarules: these are special rules which give control information to the logic interpreter. The metarule proposals of Gallaire and Lasserre [72] were outlined in the previous section.
Metarules provide a form of control which is explicit and separate from the program text. They can be seen as a special application of metalogical programming as described earlier. As yet, no existing logic programming system is known which implements this form of control. Nevertheless it appears promising for several reasons. One is that it offers the total separation of a logic program from its control, which is consistent with logic programming as a whole. Another is that the control provided can be extremely general, as illustrated by the Gallaire and Lasserre proposals. Furthermore, since metarules are written in logic the programmer would not be required to learn another formalism. There is the question, however, of whether such a scheme could be implemented efficiently. Furthermore, the same problem arises as with control annotations in identifying which set of metarule primitives should be provided. Finally, it is not certain how easy such a scheme would be for programmers to use in practice, nor indeed whether they would resist the requirement to manage 'two programs' instead of one.

3.1.7 Program verification

The use of logic for verifying programs is not new, and in fact clausal logic was used by Chang and Lee for verifying conventional programs [51]. It is claimed that the verification problem is easier to solve, however, when logic is used to verify programs which themselves are written in logic. This is analagous to the problem of deriving (synthesising) correct programs from specifications, which is simplified when the specifications are composed in logic and the program to be derived is a logic program. Indeed, it has become common for researchers to treat the two problems (of program verification and synthesis) together.

At least two methods of program verification have been described, both of which assume the existence of some formal logic specifications of the main relation which is to be computed. Following Hogger [87], this relation can be termed the principal specified relation (PSR). The method of verification described by Clark, which is known as consequence verification, consists of showing that each of the statements of a logic program can be proved as theorems from the specifications, which are taken as axioms for the PSR. Examples of consequence verification are given in the paper by Clark [137]. A technique which is based on a proof by induction over the length of the computation is illustrated by Kowalski [40]. His method proves that any instance of the PSR which is computed by the program is also an instance according to the specifications.

The form of verification offered by the consequence verification and the induction techniques is the establishment of the property of soundness or, as it is often called, partial correctness. However, verifying that a program has the partial correctness property is often not enough. Informally, it only provides a guarantee against wrong answers. It can be observed that a program which computes a null set of solutions to the PSR is partially correct, even although many solutions may be implied by the specifications. The further property of completeness is usually desirable: a program is complete with respect to its principal specified relation if every specified instance of the PSR is also a computed instance. Intuitively, a complete program is one which computes every correct answer. Verification of completeness with respect to the PSR can be established by reversing the direction of the arguments for partial correctness. It can be done by showing that the specification of the PSR is a logical consequence of the program clauses, or alternatively it can be done by showing that every specified instance of the PSR is also a computed instance.

Hogger [87] describes a program as totally correct if it is

both partially correct and complete. Hence, the total correctness of a program with respect to its principal specified relation can be established by showing that the set of specified instances of the relation is the same as the set of computable instances. Hogger demonstrates a technique, definiens transformation, which proves total correctness by transforming the specifications into program clauses in an equivalence-preserving way.

It should be noted that the total correctness of a logic program is independent of the control strategy exercised by the program executor. The separation of logic from control makes verification of the logic much more straightforward. However, a verification of logic alone is limited to proving the computability of solutions; this may differ from their producibility in an actual execution. The relevant condition, for the set of computed solutions to be the same as the set of produced solutions, is that the proof procedure be sound and complete. Unfortunately, as has been shown earlier, the standard (PROLOG) execution strategy does not meet this requirement because it can happen that infinite computations are entered before finite ones which contain solutions can be investigated. When this does happen, some computable solutions will not be produced. The presence in a PROLOG program of non-Horn clause features such as negation-by-failure and control primitives threatens completeness of producibility in a different way, for as described earlier these features can render computable solutions unproducible even where the search space is totally finite. Hence, although in a theoretical logic programming system - say, one with a fair implementation of SLD-resolution and without any completeness-threatening extensions to the language of Horn clauses - producibility is the same as computability and is a property which can be verified from the logic alone, in a PROLOG system (say) producibility requires a specific analysis of the algorithm which is determined by the application of the control strategy to the logic.

For a pure Horn clause program, the total correctness of a logic algorithm with respect to some principal specified relation can be established by proving that the logic program is totally correct with respect to the PSR and that the algorithm terminates in the sense that the entire execution process halts after some finite time period. Termination indicates that the search space is finite so that any SLD-resolution strategy (even an unfair one such as that of PROLOG) is certain to produce all computable solutions eventually. Methods for proving the termination of logic algorithms have been investigated by Clark and Tarnlund (whose formulation of total correctness differs from that given here) [156] but it is clear that this is an area which requires further work.

It can be observed that the published examples of verification are of very small programs. Even so, the verifications are typically much longer than the programs

themselves. Realistic applications would presumably require proportionally more steps in verification: if the steps are all human-directed, then verifications will surely themselves be at risk due to error. Turner has commented that 'Program proving is just a game until the proofs are computer-based and machine checked' [157] and, although his view is expressed in the context of functional programming, it seems to apply to logic programming with equal force. Some progress has in fact been made in the direction of mechanical verification. Cunningham and Zappacosta-Amboldi for example have developed a suite of modular software tools for manipulating first order logic [158]. They report that by concatenating suitable modules and interacting with them, humans can obtain useful machine assistance in performing verification of programs and related such tasks. Balogh has described an implementation of a human-assisted verification system for PROLOG programs [159].

It is clear that verification can be in principle an arbitrarily hard task of theorem-proving, as the chain of inference connecting program clauses to logic specifications can be arbitrarily long and subtle. However, logic programs are generally much closer to their specifications than are the theorems which a genuine theorem-proving system attempts to derive from its axioms. Although large logic programs may require many steps of verification, each step should be relatively trivial by automatic theorem-proving standards and this gives hope to the practical utility of future verification systems. One known limitation, which follows from the theorem of Church on the undecidability of first-order logic [46], is that no automatic verification system can be produced which is infallible for all logic programs. A more immediate barrier to progress (and in many ways it seems more serious) is that almost all realistic logic programs in current use include features such as negation-by-failure and control primitives which are likely to make the verification of logic algorithms much more difficult.

3.2 Aspects of logic as a computer language

In this section, a number of aspects of logic as a computer language are discussed. The emphasis is on those topics which were identified in Part One as being matters of major concern for traditional software development methodologies. As has been shown, the term 'software crisis' arose from the early attempts to construct sizeable packages of software with inadequate languages, tools and methods. The short history of logic programming has documented a record of implemented projects and a brief survey of these experiences is offered in the first part of this section. The second part relates to the criticism that conventional languages offer little scope for exploiting new parallel computer architectures: an account in this section describes the prospects which seem to be offered for parallel execution by logic programming. In the third part, it is recognised that along with logic programming a significant challenge to imperative computing comes from functional programming languages. It is appropriate then to include here a section which briefly compares logic and functional programming. The final topic which is discussed here is the human perception of logic as a computer language, focussing particularly on the claims that logic is problem-oriented, since the absence of this quality in conventional imperative languages is a major complaint of programmers.
It will be clear that all the issues treated here could easily be (and indeed have been) the subject of major researches in their own right. Hopefully however it will still be worthwhile to try to identify here some of the main points relevant to each theme; in all cases references are given to sources of further information.

3.2.1 The record of applications

It is clear that the number of significant applications of logic programming which have been undertaken is already large and that the list is diverse in the type of projects covered. Furthermore, it is growing rapidly. The overwhelming majority of projects have been with some version of PROLOG. Interesting applications have been described on symbolic integration by Bergman and Kanou [160] and by Belovari and Campbell [161]; on natural language processing by Coelho [162], by Colmerauer [163], by Dahl [164] and by McCord [165]; on expert systems by Clark and McCabe [166], by Hammond and Sergot [167], by Brough and Parfitt [168], by Yazdani [169] and by Hardy [170]; on mechanics problems by Bundy at al [171]; on law representation by Sergot [172] and by Cory et al [173]; on compiler construction by Warren [127, 147];

on chess end-game advice by van Emden [174]; on critical path analysis by Kriwaczek [146, 175]; on game-playing by Clark and van Emden [176]; on database systems by Gallaire and Minker [177]; and on spreadsheets by Kriwaczek [175]. A compilation of small PROLOG applications has been edited by Coelho [186].
Unfortunately, many applications have been described without specific reference to the experience of logic programming or to the advantages and disadvantages of PROLOG relative to its competitors as the implementation language. Since the main aim of many projects is to investigate specific issues rather than to reflect more generally on software development experience, this is of course understandable. However, it is the wider question of software experience with which this section is concerned; therefore, the applications described below are selected from those which do make specific such comment.

3.2.1.1 The Hungarian Experience

A remarkable range of PROLOG applications have been developed in Hungary. An overview of these is given in a paper by Santane-Toth and Szeredi who also supply some very valuable comment [178]. The applications include systems for research management, drug-interaction prediction, air pollution control, chemical information retrieval, architectural planning, COBOL program generation, network modelling and symbolic differentiation. These applications have all been developed since 1975 on various mainframe computers, including an ICL-1905, an IBM-370/145 and a SIEMENS 4004.
The authors report that:

> '... many problems have been solved using PROLOG; problems previously either unsolvable (in traditional programming languages) or solvable only by applying complex algorithms and considerable effort.'

A list of reasons for the success of PROLOG in Hungary is given which, the authors add, 'does not include the well known advantages of PROLOG programming'. The list comprises the following explanations:

> 1. No other artificial intelligence languages were available in Hungary.

2. The PROLOG interpreter (the same one was used for most of the applications) was fast and highly portable.

3. Systems programmers produced useful tools and generally cooperated well with applications programmers.

4. Two 'pilot' applications were successful and these served as a basis for subsequent applications.

5. Most of the people involved in PROLOG programming had little traditional programming background and this allowed them to learn PROLOG easily.

6. The installation of PROLOG on a large SIEMENS machine with an interactive environment aided the development of existing applications and the introduction of new ones.

7. The symbolic manipulation facilities of PROLOG led to the development of new application areas.

On the negative side, several problems with the PROLOG applications are noted:

1. Memory management was problematic. Even on the largest computer, the SIEMENS 7.755 with 3 Mbytes of stack space, there were some occurences of stack overflow. Programmers were forced into 'tricks' to recover space.

2. Several times the demand arose for an interface with algorithmic languages (mostly with FORTRAN). An experimental PROLOG was developed which allowed FORTRAN subroutines to be called from PROLOG, but 'the real solution is still lacking'.

3. The need for handling large databases, which would be held on external disk files, arose.

4. The naive backtracking mechanism caused problems in some programs, and in these cases special search strategies had to be programmed in PROLOG itself.

5. Some applications ran too slowly.

6. There was the lack of a textbook with which to introduce people trained in traditional computer programming to PROLOG.

Partly in response to these problems, the Hungarian workers have developed the MPROLOG system. They describe the essential new features of MPROLOG as being module construction facilities, improved programming aids and a better execution mechanism. A compiler for the language is under construction. Concluding their account, Santane-Toth and Szeredi are optimistic:

> 'In fact in the new MPROLOG system, some of the listed problems are already settled. Hopefully the others will be solved in the near future, also in the framework of PROLOG.'

3.2.1.2 Chess end-game advice

Van Emden has conducted a study of chess knowledge-representation and utilisation by computer [174]. A PROLOG program was developed which can solve chess end-game problems: on being given a description of the board state, the program is capable of finding winning sequences of moves. The application was designed particularly with a view to appraising PROLOG's suitability for this type of project.
The application was completed successfully. Using the Waterloo PROLOG system [179] on an IBM 4331, the program played games in which mating positions were found within nine, twenty-four and twenty-nine moves respectively. The corresponding average CPU times per move were 0.48, 0.80 and 0.72 seconds.
Van Emden identifies several factors which make logic programming attractive for knowledge engineering applications of this type. He notes that Horn clauses can be viewed as production rules, which have emerged (independently of logic programming) as the favoured formalism of knowledge engineering. Furthermore, clauses encompass as a special case the relational data base model: consequently, the usual distinction between program and database disappears, which is 'especially attractive for knowledge engineering'. The researcher compares PROLOG with the 'advice language' approach proposed by Michie [180], in

which knowledge is encoded within tables using the advice language AL/1 and special-purpose interpretive procedures are written in POP-2. The choice of PROLOG avoids the need for two separate formalisms. In addition, PROLOG clauses are more general than AL/1 rules, since AL/1 (sub-) rules may not have conditions attached to them. On the other hand, it is noted that the chess application has what would be considered an uncharacteristically simple advice table in AL/1, and moreover, advice in AL/1 contains in general several advice tables. The completed program in Waterloo PROLOG is described as, on the whole, compact and readable. The distinctive features of PROLOG which proved useful in the application were its rule-based, pattern-matching and automatic backtracking aspects.
Three adverse points are identified. First, the representation of arithmetic expressions by conjunctions of relations hinders program readibility. In remedy, the author points to the development currently of PROLOG systems which offer a functional representation of arithmetic. Second, the Waterloo PROLOG system is criticised as being inefficient in its use of storage. In fact the twenty-nine move game quoted above produced a stack overflow after the twenty-eighth move and the game was completed by restarting with the last position on an empty stack. Third, the researcher laments the absence within Waterloo PROLOG of a sets-of-solutions primitive, since the need to gather a list of solutions occurs for example in the development of forcing trees.
In summary, Van Emden concludes that:

> 'Waterloo PROLOG is probably a good (compared to other implemented alternatives) tool for this type of application, in fact surprisingly good for an early, experimental realisation of logic programming'.

3.2.1.3 Representation of law

A PROLOG implementation of a fragment of British law has been undertaken by a team at Imperial College, London. The particular law chosen was the British Nationality Act (1981). The project was developed using the APES (A PROLOG Expert System Shell) system [181] running on a microcomputer. A system has been produced which is capable of interactively determining the validity of British citizenship in a large number of frequently occuring cases. Descriptions of the work are given by Cory et al [173], by Sergot [172] and by Kowalski [182].
The main concern of the project has been to study the problems of knowledge representation. Sergot (op. cit.) points out that logic programming would appear to be a

natural formalism for a computer treatment of law, since

> '... law treats large sets of complex rules that have long seemed suitable for logical analysis, and once the law is expressed in some appropriate subset of predicate logic, that formulation can function as a program which interprets the law.'

He identifies two general kinds of law which require different treatments in logic. The first kind can be viewed simply as high-level descriptions which more or less precisely define legal relationships, such as the property of citizenship: this kind of law can be relatively easily formalised as logic programs. The second kind of law is the body of 'norms' which state what must or must not be done under certain circumstances: this includes unwritten law such as case law and the 'general legal principles' (such as 'No man may profit by his own wrong-doing') which guide judges. A logic representation of this kind of law poses problems such as establishing what the norms are, what they mean, and how they can be used. Sergot, taking a very simple example of 'norm' type law (actually, hypothetical library regulations), outlines a general approach which attempts to model the norms by logic programs. An important problem for future work is the use of norms to constrain database updates. As with the general problem of logic programming database maintenance, the most promising direction for a solution seems to lie with some amalgamation of object language and metalanguage such as that proposed by Bowen and Kowalski [92].
Many law texts contain references which are bound by time. In the PROLOG representation, a time parameter can be incorporated into the corresponding relations. Other texts include statements which are not readily translatable into Horn clause form: Kowalksi [182] quotes the example

> A person is a citizen if
> .....
> and his mother is a citizen
> or would have been a citizen if she were male.

A PROLOG translation of the 'or would have been..' phrase uses negation-by-failure. In general, there are many legal texts which express 'knowledge about knowledge'. These can be viewed as expressing metalevel information and metalogical programming is suggested for their translation. Kowalski notes that PROLOG has facilities which can accomplish this, although they are not always consistent with classical logic. He restates his belief that the incorporation of correct and powerful metalevel facilities within practical logic programming systems 'would go a long

way toward meeting both the critics of logic programming and the critics of logic'.
Sergot compares PROLOG with LEGOL, a legal computing system which is regarded as the most general legally oriented system available. LEGOL is based on relational algebra. It has found a number of practical applications, in spite of its inadequacy for expressing certain types of legal rules. However Sergot suggests that LEGOL could usefully be viewed instead as a logic programming language tailored for specific applications areas. He considers that the power of PROLOG offers interesting opportunities for more general work in legal computing [183].
The Imperial College project appears to have highlighted some advantages in the explicit representation of legal knowledge which Michie [184] has described as a general by-product of expert systems developments: that through efforts to make knowledge explicit, it becomes better understood and 'refined'. Kowalski has answered an interesting criticism of his work, to the effect that rules which affect human beings need a more flexible interpretation than should be expected from a computer, by saying:

> '... by making rules explicit it is easier to see where they might be inadequate and how they might be improved. The expert systems technology of knowledge ... offers the prospect of extending the use of rules in human organisations. It suggests new and more powerful ways of making rules explicit, of applying them more consistently and of refining and improving them.' [185]

Perhaps the most encouraging outcome of the project has been the realisation of how much can be achieved even with existing PROLOG. Kowalski remarks that

> 'It is remarkable that despite PROLOG's simple problem-solving strategy and except for a few loops removed manually by program-transformation techniques, the rules extracted declaratively from the act behave reasonably efficiently as a logic program.' [182]

3.2.2 Progress in implementing parallelism

It should be possible for logic programs to exploit a parallel processing capability at both a 'fine' and a 'coarse' level of grain. At the 'fine' level for example, the unification algorithm might be reformulated as a parallel algorithm, and this possibility has in fact been investigated by Tarnlund [215]. The greatest benefit however is believed to be obtainable at the 'coarse' level in a parallel interpretation of logic itself, and this is the level which will be discussed here.

As described in Part Two, logic programs potentially support two forms of parallel interpretation. The first form, known as and-parallelism, concerns the possibility of executing concurrently the goals of a conjunction (such as the conditions in the body of an invoked clause). The second form, or-parallelism, refers to the possibility of investigating concurrently the clauses which potentially respond to a given call. These two forms are sometimes referred to as the conjunctive and disjunctive forms of parallelism respectively. Lloyd [130] has referred to the process interpretation of logic as one in which a goal statement

<- G1, ... , Gn

is regarded as a system of concurrent processes, where each step in the computation reduces a process to a system of processes (given by the body of the activated clause) and where shared variables act as communication channels between processes.

Much research effort has recently been aimed at developing logic programming systems which support concurrent programming. As yet however no proposals have been advanced which incorporate both and-parallelism and or-parallelism in full unrestricted forms. For reasons of efficient implementation, existing proposals place various limitations on the allowed forms of parallelism and some introduce new semantic restrictions, such as that only a single solution to a goal will be produced rather than a complete set of solutions. Furthermore, almost all proposals give the programmer a large amount of control over the extent of the parallelism. (This seems to answer the criticisms of Kluzniak and Szpakowicz [213], who have argued that any multiplicity of processors will be defenceless against the anticipated combinatorial explosion of 'unbounded parallelism').

Two of the earliest systems supporting some form of parallelism were IC-PROLOG and LOGLISP, both of which were described in Part Two. IC-PROLOG provided and-parallelism with annotations on variables in clause bodies specifying data-triggered communication between processes. A process which acted as the producer of data for a variable was

indicated by annotating the variable with '^', and consumer processes were indicated with the variable annotation '?'. A consumer process which required data to continue became suspended until the data was made available by a producer. LOGLISP provided or-parallelism implemented by means of a breadth-first search strategy. Neither system offered the form of parallelism provided by the other, and since both systems were implemented using conventional single-processor computers, the 'parallelism' was actually simulated by time-slicing.

Superficially at least, or-parallelism appears to pose fewer implementation problems than the conjunctive form. With one complication, a set of or-parallel computations are essentially independent of each other since they correspond to investigations of different branches of the search tree. The complication is that the computations may begin with a shared unbound variable which may become bound during one of the computations. Special storage mechanisms would be needed to preserve the independence of the computations.

Hogger [216] has shown how the data-flow annotations of IC-PROLOG can be used to describe concurrent algorithms which require communication between processes. He describes the principles whereby problems are reformulated with the introduction of shared variables to act as the vehicles of communication. Since IC-PROLOG does not possess the capability of disjunctive parallelism, a conjunctive formulation of the problem is required. As an example, Hogger quotes the the problem of finding whether a given element E belongs to either of two given sets A and B. An initial (sequential) conjunctive formulation of the problem is with the goal statement

```
<- m(E, A, a1) & m(E,B, a2)
```

where a1 and a2 are 'answer' variables which will eventually become bound to either YES or NO. Assuming a suitable definition for m, a conventional evaluation of this will compute one of the answer pairs (NO, NO), (NO, YES), (YES, YES), (YES, NO), where any answer apart from (NO, NO) indicates that the answer to the problem is positive. Alternatively, a more efficient parallel execution from which the same set of solutions is computable is

```
<- m(E, A, a1) // m(E,B, a2)
```

which uses IC-PROLOG's parallel annotation '//'. Efficiency can be improved still further however by arranging that when one process succeeds, it communicates its success to the other thereby enabling the unfinished process to terminate with its answer variable being bound to (say) DONTKNOW. This can be done by defining a suitable four-place predicate m* and executing the goal statement

```
<- m*(E, A, a1, a2) // m*(E, B, a2, a1)
```

Each process now shares the answer variable of the other and is thus sensitive to its outcome. Answers such as (DONTKNOW, YES) can be computed which indicate that the problem has been answered positively, and furthermore these answers are more efficiently computable than an answer such as (YES, YES).

The absence of disjunctive concurrency and other deficiencies in IC-PROLOG has however provided a challenge for logic programming researchers. Probably the most important response to these deficiencies has been the Relational Language of Clark and Gregory [217]. In the Relational Language, the global backtracking evaluation strategy of IC-PROLOG was abandoned for two main reasons: first, it it was felt to be viable only for single-processor architectures; and second, the cost associated with the failure of a process was considered to be too high. (On the failure of a process, it was necessary to undo every evaluation step that had taken place after the choice point of the failed process.) The Relational Language incorporated conjunctive concurrency in a manner similar to IC-PROLOG, with variable annotations to designate the consumer and producer processes of a parallel computation, but it also introduced a special form of disjunctive concurrency: the program 'candidate' clauses which might respond to a call were tested in parallel and the first clause to pass the test was the one that was used. There was no backtracking on this choice. The test for candidacy included a mode check and a check that a subset of the conditions in the body of a clause (known as the guard sequence) succeeded. The effect was to implement a special version of the or-form of logic programming non-determinism which Clark and Gregory describe as 'committed choice' non-determinism. They liken it to Dijkstra's language of guarded commands [135].

More recently three further language proposals have been published, all of which appear to be derived from the Relational Language: these are Clark and Gregory's PARLOG [218], Shapiro's Concurrent PROLOG [219] and Ueda's Guarded Horn Clauses [220]. It is clear that whilst they have significant differences, these proposals also have a considerable amount in common. Only PARLOG will be described in what follows.

PARLOG divides relations into two types: single-solution relations and all-solution relations. A conjunction of single-solution relation calls can be evaluated in parallel with shared variables acting as communication channels for the passing of partial bindings. Each single-solution relation is defined by a guarded clause program with a mode declaration which specifies constraints on the allowed form of call; each argument of the relation is annotated in the mode declaration with either a '?', to indicate that a call must supply some input for that argument, or '^' to indicate that the corresponding argument in the call must be an

unbound variable (i.e., one which will receive output). If a call does not satisfy an input constraint, it becomes suspended. As in the Relational Language, PARLOG also supports disjunctive concurrency in the form of committed choice non-determinism. This eliminates the need for backtracking and as might be expected, it is reported to greatly simplify the implementation of PARLOG. However, as with the Relational Language, it also means that only one solution can be computed to a conjunction of single-solution relation calls. In recognition of the need to support applications where all solutions are required, PARLOG also incorporates the all-solutions type of relation. A relation of this type is defined by a normal PROLOG program, with no guards and no mode declarations. A conjunction of all-solutions relation calls is evaluated sequentially left-to-right, as in PROLOG, although the defining clauses may be investigated in or-parallel fashion. An interface between the two types of relations is provided in the form of a set constructor.

The first implementation of PARLOG is a simulation which runs on top of a conventional PROLOG system [221]. A fast portable version written in the language 'C' is also under construction. A pilot implementation on the parallel computing machine ALICE [222] has been undertaken and Gregory has described how it should be straightforward to compile PARLOG to the ALICE CTL (Compiler Target Language) [223].

It is clear that PARLOG at least represents a substantial step towards the realisation of highly concurrent logic programming. (It is also beyond dispute that PARLOG marks a radical departure from PROLOG, and as such it should help to underline the distinction - which sometimes seems to be underplayed in the literature - between logic programming on the one hand and its early realisation in the form of PROLOG on the other.) The language's use of mode declarations appears to have rendered the full run-time application of unification unnecessary, so that PARLOG programs should be efficiently compilable. It is uncertain however what effect such features as the committed-choice non-determinism and the separation of relations into two types will have on the usability of PARLOG as a logic programming language. It is noticeable that the developers have avoided the temptation of elaborating the language beyond those extensions which are required for parallelism, even although some extensions such as augmenting the mode declarations to incorporate type-checking information appear to be quite straightforward and attractive. (Gregory has confirmed in a private communication that a single-minded pursuit of the parallel aspects lie behind this decision). Happily, it appears that in general PARLOG programs do have a good declarative reading and it may be hoped that the suggestion of Hogger [216] that the introduction of concurrency does not require departure from the usual way of developing logic programs will prove to be correct.

3.2.3 Relationship to functional programming

In Part One it was argued that all imperative languages are alike in the sense that they are geared to the architecture of the von Neumann computer. Logic on the other hand is a machine-independent formalism. However, it would be false to suggest that logic programming is alone in its departure from the imperative mould. Functional programming represents another alternative, one which is also based on a mathematical formalism. But where logic programming owes its origins to first-order logic, functional languages originate from the lambda calculus and recursion equations [187, 188]. A program in a functional language defines an expression which is the solution to a set of problems: the program executor finds the solution to a particular problem by evaluating the corresponding expression. The earliest functional language, LISP, was introduced by McCarthy in 1958 [189]. The record of LISP applications is extensive and it is still widely used today, especially in the United States where LISP is the main language of artificial intelligence. However, such a wide range of extensions have been attached to the language (some of which, such as PROG and GO, are distinctly imperative in character) in its various implementations that functional programming proponents are highly dubious about its functional status. Turner for example has gone so far as to state his suspicion that 'the success of LISP set back the development of a properly functional style of programming by at least ten years' [190]. Examples of modern, pure functional languages are KRC, Miranda, ML and HOPE. Broad treatments of functional programming are provided by the books of Glaser et al [23], Henderson [191], and Darlington et al [192].
The term 'declarative' is now often used to cover both logic and functional programming languages. It is taken to refer to the self-evident, execution-independent interpretation property which programs in these languages can possess. Growing interest in declarative languages has led to increased attention on the relationship between them. A recent study by Darlington, Field and Pull [193] has discussed the differences between the two approaches and has proposed a possible form of unification. Observations from this study and others will be drawn upon in the two sections which follow.

3.2.3.1 Major differences

The following points identify major characteristic differences between logic and functional languages.

1. Relational versus functional semantics

Logic programs define relations which specify many-to-many transformations. Functions on the other hand specify many-to-one transformations. Hence a functional program is generally limited to the output of a single solution to a problem, where logic programs may output multiple solutions. It should be noted that the single-solution property of functional programs does not preclude the possibility that the solution computed might be a set. Nevertheless, it is a mathematical property that functions are special cases of relations. In particular, it seems reasonable to allow program specifications to be expressed as relations, at least in the first instance, even although some of these relations may later be identified as functional. (Hoare's observation that programs are predicates [141] is relevant here.) Hence, the arguments in favour of executable specifications would appear to support logic as the formalism of choice for this stage, although other considerations (such as those of efficiency) may become dominant in the later stages of software development. In fact this is one of the conclusions reached by Darlington et al. (op. cit.).

2. Execution mechanism

Where logic programs are executed by applying top-down resolution inference, functional programs are executed by expression reduction. For instance, the following is a HOPE program which calculates the length of a given list:-

```
dec length : list alpha -> num;
--- length(nil)  <= 0 ;
--- length(x::l) <= 1 + length(l) ;
```

An execution to calculate the length of the list [1, 2, 3] would go through the following rewrites:-

```
length([1, 2, 3])
        -> 1 + length([2, 3])
        -> 1 + (1 + length([3]))
        -> 1 + (1 + (1 + length([])))
        -> 1 + (1 + (1 + 0))
        -> 3
```

The standard way to implement functional languages has been with the SECD-Machine, which is based on Landin's design of an automaton for the mechanical evaluation of mathematical expressions [194]. More recently Turner has shown a remarkable implementation technique whereby functional

programs can be compiled by the use of a small group of elementary lambda-calculus functions known as 'combinators' into variable-free code [195]. The new technique, known as the SK-Reduction Machine, should have efficiency advantages. Overall, however, the problem of obtaining acceptable efficiency has been as serious for functional programming as for logic programming. Henderson has recently written that

> 'Functional languages cannot compete with conventional languages on conventional machines in terms of efficiency if that is an absolute requirement.' [191]

However, Turner quotes a study by Meira [166] which shows that for each known imperative sorting algorithm, there exists a functional sorting algorithm of the same fundamental time complexity [190]. He goes on to conjecture that for time complexity, there is no fundamental difference between imperative and applicative programming. But Turner also notes that the question of space complexity is much less clear, and in fact it has been shown that for some very simple problems it is surprisingly difficult to construct functional solutions with a reasonable space behaviour.
Ultimately, researchers in both logic and functional programming look to the development of non-Von Neumann computer architectures which can execute their programs with greater efficiency, particularly by exploiting the scope for parallelism which is inherent in both formalisms. Glaser et al have outlined a scheme for running functional programs on a data-flow architecture [23]. The main scope for parallel execution of functional programs appears to lie in the parallel evaluation of sub-expressions: Kowalski has compared this with the and-parallelism of logic programs (as described earlier) [40]. Logic programs, however, also provide (as was noted earlier) an opportunity for or-parallelism. Whether in practice this means that logic programs can be executed more efficiently on parallel architectures than functional programs is not yet certain.

3. Typing

Logic programming languages are usually untyped. Most modern functional languages are strongly typed and the typing includes polymorphism.
As noted earlier, the experience of software engineering provides strong support for typed programming languages. There seems no fundamental reason why logic programming languages should not be typed and proposals for adding typing to PROLOG were discussed earlier. Development in this area appears to be a priority.

4. Notation

The contrast between the two notations, one functional and the other relational, is obvious.
Since relations include functions, logic programming should be able to utilise functional notation. It was noted earlier that such notation is sometimes more natural and that it is in fact provided by some existing logic programming languages.

5. Invertibility

Logic programs make no commitment as to which variables of a relation are to be considered inputs and which are to be outputs. Hence, a logic program defining (say) the append relation for lists can be used to find splittings of a list as well as to concatenate lists. Functional programs on the other hand have fixed input/output semantics, and a functional definition of append could only have one form of use.
As Darlington et. al. (op. cit.) recognise, the invertibility property makes logic programs more expressive than functional ones in that one logic program can represent many functional ones. This is particularly advantageous in the early specification stage of program development. It was noted earlier, however, that many practical logic programs are written to support only one mode of use and that they can support other modes only inefficiently or not at all. Consequently, it seems desirable that logic programming systems should support mode declarations or some other means of indicating the allowed forms of call. Functional languages do not require such mechanisms, since the single mode of use of a function is implicit in its declaration.

6. Logical variables

Function applications may return only constants or functional applications to constants. In contrast, the output of a logic program may be a data structure which includes uninstantiated variables. For example, in micro-PROLOG the call

```
which(Z: append((1 2) Y Z))
```

will produce the answer (1 2¦x). Darlington et al. (op. cit.) have noted that this capability has on several occasions led to programs which are more abstract and more efficient than would otherwise be possible.

7. Determinism

Functional languages are deterministic. The output of a functional program is completely determined by its input and can be obtained without any searching. Logic programs however are non-deterministic in that searching for solutions (which may be multiple, as already noted) is generally unavoidable and furthermore the search can be conducted in many different ways.
Turner has noted that if functional languages are to be applied to such problems as the construction of operating systems, a form of non-determinism appears to be a prerequisite [190]. Unfortunately, the implementation of non-determinism in functional languages would seem to destroy their referential transparency. This point is accepted by Glaser et al [23]. Henderson, in describing a possible implementation of non-deterministic primitives within his LISPKIT LIST system, notes the price which is to be paid in the reduced transparency of programs and suggests that the programmer must use his experience in deciding whether to use them [191]. Turner (op. cit.) expresses the hope that further research will produce simplifications in this aspect.

8. Higher-order expressions

A distinction between the two approaches which is cited by several authors concerns the higher order application of functions and relations. For example, Turner states that

> 'Functional programming ... has the important advantage of being higher order, i.e. it permits the manipulation of functions as data objects, in a way that respects the principle of extensionality, whereas nothing quite equivalent to this exists in logic programming.' [190]

Evidently the analogue in logic programming is metalogical programming, in which relations describe relationships between other relations. A discussion of metalogical programming appeared earlier: it will be recalled that, whilst it is true that the metalogical facilities of most PROLOG systems have been problematic, recent work in this area such as that by Bowen and Kowalski [92] appears to hold considerable promise for logic programming.

3.2.3.2 Future directions

As the above discussion shows, there are significant differences between the two approaches. However, they are both declarative and well-rooted mathematically and this reflects iteself in many similarities. In particular, the functional programming view of software development is strikingly similar to that of logic programming as described earlier, where executable but possibly inefficient specifications are mathematically transformed into a correct and efficient program. A recent informal account of the development methodology for declarative languages in general is given by Darlington [197].
A number of proposals have been advanced for some form of unification between the two approaches. One direction is to provide facilities for both, together with an interface betwen the two within the same general programming environment; this is exemplified by systems such as LOGLISP [70], FUNLOG [198], and POPLOG [199]. A different direction is to attempt to subsume one approach within the other, possibly by making extensions to the subsuming language. An inspection of the differences listed above might suggest that a satisfactory subsumption of functional programming within logic programming is more likely than the reverse, but proposals have in fact been advanced in both directions. Thus McCabe's Lambda PROLOG [200] is reported to fully support functional equations within a logic programming framework and Darlington et al [193] have proposed an extension of functional programming to incorporate unification and non-determinism.

3.2.4 Human perceptions of logic programming

All the theoretical benefits of logic programming may be of little practical value if the human mind is in some fundamental way ill-suited to it. It is useful then to consider the evidence which relates to the human perception of logic programming.
A recent review of behavioural research into the effects of programming languages and programming methods in general on programmer performance has been conducted by Sheil [201]. Unfortunately, the results are far from conclusive. Shiel suggests that much of the research suffers from the lack of an adequate understanding of the programming process. Consequently, claims for the superiority of one method over another should be treated with caution. Similarly, a paper by Brooks [202] on the problems of experimentally studying programmer behaviour concludes that models of the cognitive processes involved in programming must be developed before any substantial progress can be made.

In the absence of firm experimental evidence from research psychologists, some informal observations might still be useful notwithstanding their somewhat anecdotal flavour. The first of the following sections describes some of the reactions of learners to (invariably, some version of) PROLOG. The second presents some more fundamental criticisms which have been made of the problem-solving capabilities of logic programming.

3.2.4.1 Experiences of PROLOG learners

Learners can be conveniently grouped into two categories: non-programmers and experienced programmers who are approaching PROLOG from a background of imperative programming.
A major project at Imperial College, London, known as the 'Logic as a Computer Language for Children' project, has been aimed at introducing logic programming to children aged ten to thirteen [203]. Using a subset of the SIMPLE interpreter for micro-PROLOG running on microcomputers, children have been taught to construct and query elementary logic databases expressing their knowledge of history and other subjects. Programs have been viewed declaratively, virtually to the exclusion of any procedural interpretation. Essentially, this has been possible because the (computationally) trivial nature of the problems has usually guaranteed PROLOG's good behaviour.
The Imperial College project suggests that the separation of logic from control which is central to logic programming is helpful to learners, in that they can concentrate on specifying the logic of their problems and rely solely on the control provided by the logic interpreter for their execution. PROLOG can be initially presented as though it was some kind of relational database language. It can be noted in passing that relational database languages have generally been found quite accessible to naive users (a survey of them is provided by Pirotte [204]).
For non-trivial problem-solving, however, learners must know at least something of the procedural semantics and the books which introduce PROLOG programming by Clark and McCabe [96], by Clocksin and Mellish [95], by De Sarem [205] and by this author [206] all make some attempt to explain the PROLOG execution mechanism. The explanations vary in both method and extent and a clear consensus on the optimum pedagogical strategy has yet to evolve. Although Kowalski suggests that the presentation of the procedural interpretation can be limited to three points [203], it is clear that a backtracking execution in particular is a potential source of confusion. Ennals et al. point to further difficulties which PROLOG's strategy may cause to learners over the order of conditions in left recursion and in negation-by-failure [207]. Indeed, virtually all the extensions to Horn clause logic are potential snares in that, whilst they provide

expressive power which could especially benefit learners, these extensions typically have special operational restrictions, as noted earlier.

The inadequacies of PROLOG as a logic programming language are a source of frustration for instructors. An extreme response, of which De Sarem's text (op. cit.) is an example, is to virtually abandon the logical view in favour of a presentation of PROLOG as an imperative programming language with a highly unusual execution strategy. Thus, all the ambitions of logic programming are dismissed. More usual and more constructive, however, has been an instructional approach which in the words of Sergot 'recognises PROLOG's close relationship with logic, and views it as a primitive but efficient implementation of the logic programming ideal' [103]. This approach presents learners with the logical reading of program clauses first, but also provides the PROLOG control reading, drawing attention where necessary to any restrictions. In this researcher's experience of teaching PROLOG programming to beginners, both adults and children, this approach is quite feasible.

As this thesis has shown, there is a great deal of scope for developing logic programming languages with an execution strategy which is different from standard PROLOG's. Kahn has warned that these languages might actually be inferior from the learner's viewpoint, because a more sophisticated execution strategy may be more difficult to predict [208]. Presumably, however, the extent to which a new strategy will be judged successful will be determined at least partly by the decline in the practical need to predict its behaviour. Morever, a more sophisticated execution mechanism than a depth-first implementation of SLD-resolution is not necessarily less comprehensible, as the LOGLISP example has indicated. However, Kahn's point stands as a valid warning against an over-complex autonomous execution strategy since, as was shown earlier, no matter how much it is improved there will always be problems which it cannot solve and hence there will always be a requirement for human beings to understand its behaviour. More intriguing is the question of whether human beings will trust, or indeed should trust, a machine which delivers solutions by a process which it is difficult or even impossible to explain. Of course, this anxiety applies also to solutions delivered by imperative programs: in fact, arguably more so, because the controlled assignment method of computing solutions is arguably less penetrable to human intelligence than the controlled inference method of logic programming.

It is in the problem-solving applications of logic that some of the greatest benefits should be seen, and this is where imperative languages are particularly inadequate. Hogger's recollection: '... as a science undergraduate in an introductory FORTRAN course, being able to accept descriptions of the effects of individual statements upon the machine but uncertain as to how they should be knitted together in a manner consistent with the problem's logical

structure' [87], seems quite typical and it illustrates well the lack of a clear problem-orientation in traditional programming. The theory of logic programming program development permits the specification of a problem in logic to be executed directly, with subsequent correctness-preserving transformations to remove inefficiencies, as described earlier. Unfortunately, it is noticeable that most PROLOG texts, including those by Clark and McCabe and by Clocksin and Mellish mentioned above, provide little or no explicit guidance on program development methodology, relying instead mainly on the presentation of examples. This researcher has published an informal framework to assist learners in the development of PROLOG programs starting with English language specifications [206]. Although presented to learners under the title of 'top-down description', it is actually based on the top-down logic programming methodology which was described earlier. English specifications of relations are translated into PROLOG clauses the bodies of which generally introduce new relations which are in turn themselves specified and translated. After each stage of decomposition comes an efficiency check: the procedural interpretation of the new clause under the PROLOG control strategy is checked for its problem-solving capability against the anticipated goal. Experience with learners using the methodology is favourable: it seems likely that something like this will be a useful complement to the more formal, verifiable and (hopefully) automatable methods of program development which are now being investigated.

3.2.4.2 Criticisms of logic for problem-solving

Potentially some of the most serious criticisms of logic programming have concerned the adequacy of logic itself for representing some of the problems which need to be solved. One example is the criticsm formulated by Minsky concerning the monotonicity of logical consequence [209]. Another is the more recent criticism by Hewitt of the choice of logic programming for the Japanese Fifth Generation project on the grounds that logic does not adequately distinguish action from description and does not cope with inconsistent information [210]. Conceptually, these and other criticisms can be regarded as implying that the model of computation which is (currently) offered by logic programming, which is intuitively that of deduction from a fixed and self-consistent theory, is inadequate for certain cases of problem-solving. Campbell mentions as examples those problems in which relationships are time-dependent; those which involve cause and effect; and those which require reasoning about uncertain or incompletely described relationships [211].

Unfortunately, no detailed study is known which has identified and analysed the problematic categories. Research reports sometimes suggest that the difficulty lies with finding the appropriate logical formulation of a problem, rather than denying that any such formulation is possible. For example, Mellish and Hardy introduce their account of the Exeter POPLOG system by writing:-

> 'Although PROLOG undoubtedly has its good points, there are some tasks (such as writing a screen editor or network interface controller) for which it is not the language of choice. The most "natural computational concepts" for these tasks are hard to reconcile with PROLOG's declarative nature. Even if some way could be found to view these tasks as naturally declarative, programming in PROLOG could still be wasteful because of the existing expertise in writing these kinds of programs in procedural languages.' [102]

In fact, PROLOG has been used to write editors. For example, the impressive structure editor which is part of the micro-PROLOG system is itself a PROLOG program, and this researcher has published a PROLOG software package which includes some quite extensive interactive input/output facilities [212]. However, in the latter case at least it is true that the top level of the task was specified in procedural terms. An algorithm was written which was then implemented in PROLOG under the assumption of the standard control strategy. It can be fairly said that this approach to some extent sacrifices the advantages of separating logic from control which are central to the claims made for logic programming.

The question of whether it is more 'natural' to express a particular specification in procedural or in declarative terms is frequently touched upon but seldom discussed in depth. The first page of Clark and McCabe's text on PROLOG programming contains the somewhat bald statement: 'While undoubtedly we sometimes think behaviourally, most often we do not' [96]. Kluzniak and Szpakowicz on the other hand state that 'programmers often find operational terms more natural' [213]. They argue morever that successful programming lies in the programmer's ability to rapidly alternate between the operational view of a problem and the formal logical view while developing the program, and that the benefit of PROLOG lies in its provision of a common notation for both viewpoints. Clearly this claim requires investigation by psychologists. It is interesting to note that the pioneer developmental psychologist Jean Piaget,

whose theory of cognitive development stresses that concrete operational thought gives rise during adolescence to formal abstract thought, has recently been criticised by other psychologists for concentrating too much on the psychological significance of abstract logical structure while tending to ignore the effect of concrete content and context [224].
The informal introductory book on problem-solving with PROLOG written by this author implicitly concedes that for some problems an algorithmic specification may be simpler to formulate [206]. It suggests that learners should attempt to classify a problem initially as either a 'problem-to-prove', a 'problem-to-find' or a 'problem-to-do', where the latter category (alone) is characterised as comprising 'non-logical' problems which are primarily concerned with controlling the computer's behaviour. The specification which is formulated for a 'problem-to-do' is a procedural one. However, it is worth noting that although the top-level specification of a 'problem-to-do' is non-declarative, it is typically the case that lower levels of the problem are in fact straightforwardly specifiable in logic so that there are still clear benefits to be had in a logic programming approach. Other researchers have also supported an approach to PROLOG programming which admits to breaches of declarative purity. For example, Kluzniak and Szpakowicz have written:-

> '... the stress is on the dynamics of programming and on the local validity of dual interpretation. It is not as important to maintain the purity of - uniform - declarative reading throughout a big program. The problems of programming as a whole are better dealt with in terms of modularisation and abstraction ...' [213]

But there seem to be two main objections to adhering to a language such as PROLOG whilst taking recourse to procedurally based problem specifications. The first is that PROLOG lacks the imperative expressiveness of a modern procedural language, so that procedural specifications do not always find a convenient PROLOG translation. (On the other hand, the fact that it actually can be done seems to support Kowalski's point, made in [203] for example, that logic programming reconciles the old conflict between the procedural and the declarative representations of knowledge). The second is that the logic programming software development methodology is subverted: the problems associated with the development of correct, reliable, maintainable software by procedural methods are to some extent re-introduced.
Some researchers have tried to resolve the difficulties by using forms of logic which either extend standard predicate

calculus or which actually replace it altogether. A review of these non-standard logics is given by Turner [214]. The category of logics which extend predicate calculus includes modal logic, in which the truth value of relationships takes account of different possible 'worlds', and temporal logic, in which truth values take account of different possible times. The category of logics which essentially rival predicate calculus include multi-valued logic, which permits more than two truth values, fuzzy logic, which introduces 'vague' predicates and which interprets 'true' and 'false' as themselves imprecise, and intuitionist logic, which is based upon the constructivist view of mathematics. Turner points out that much is known about these logics and he suggests that for AI, in particular, they provide formal tools with which to develop theories of knowledge representation and plausible inference. He predicts that although they are as yet not understood by most AI researchers, the use of non-standard logics will become a very commonplace phenomenon.

If Turner is right about the significance of the non-standard logics then some important issues are raised for logic programming. An obvious question concerns what control mechanism would be required (if indeed any were possible) to supplement or substitute for top-down resolution in order to interpret a logic program which made use of a non-standard logic. So far, however, the non-standard logics have largely been ignored by logic programming researchers. Hogger for example recalls that Horn clauses have been shown to be a universal computing formalism, one which posseses equivalent computational potential to the other formalisms normally studied in the theory of computability, and he concludes that:-

> 'Because of this, and because logic programming is now so well-established in both theory and practice, it is strictly unnecessary - and possibly of no particular benefit - to deploy non-standard logics (such as fuzzy, temporal or modal logics) for computational purposes; it is more worthwhile to seek first to achieve their desired function by formulating and implementing them in standard logic.' [87]

However, it might be said that in the 'strict' sense of what can be computed, FORTRAN (say) is no less adequate than logic. It is surely relevant to seek any route towards a more problem-oriented programming technology, and one which could perhaps be accomodated into a (suitably extended) logic programming framework must merit serious investigation.

Certainly however there is evidence that at least some

problems in the areas which are difficult for standard logic can be formulated so as to be tractable for existing logic programming. For example, the papers by Clark and McCabe [166] and by Hammond [167] both indicate ways in which PROLOG can cope with some forms of uncertainty in expert systems applications. Kowalski has shown how time and event information can be dealt with in the problem of database maintainance [66]. Although no specific study from a logic programming perspective of the problems associated with such applications as screen editors or network interface controllers is known, computer graphics has been the subject of contributions by Kowalski [109] and by Julian [108].
At a recent meeting of the Royal Society held to discuss programming languages, Kowalski was invited to reply to the following contribution to the discussion:-

> 'Many people are reluctant to discard procedural programming concepts; perhaps this is because they see the execution of a program as primarily a simulation of a succession of events in the world, rather than as a process of deduction about what holds in one particular state of the world.' [40]

In response, Kowalski said that this identified what is probably the most important, unresolved problem in logic programming. In fact, however, it seems that there are two unresolved problems here rather than one: there is the problem of how human beings 'most naturally' interpret the world, and there is the further problem of whether and how logic programming can be applied to problems which appear to go beyond the model of deduction from a fixed self-consistent theory. It can readily be anticipated that both problems will require considerable research for their solutions.

3.3 Conclusions

Logic programming represents a major departure from traditional methods and it is to be expected that the response of many computing professionals will be one of suspicion. It is appropriate then to repeat that, as was shown in Part One, the traditional methods have failed to solve the software crisis. This message is underlined in a recent report on the 'state of the art' in programming technology, in which Wasserman concludes of current practices that

> 'Software design and development is the weakest link in the system development process. It has become extraordinarily and prohibitively expensive, and remains unpredictable in terms of economically and dependably producing processes that run reliably, correctly and efficiently.' [225]

In logic, computing science has a well-founded and versatile software formalism, one which can be used to express specifications, programs, and proofs of programs. As a declarative language its programs have the highly desirable property of referential transparency. Formal methods of development, which may be machine-assisted and perhaps ultimately automated, are much more easily applied, as the existing research on program synthesis, transformation and verification has shown. The potential contribution of logic programming towards the consistent, efficient development of totally correct and verifiable software seems likely to be substantial.

The record of experience of PROLOG applications is encouraging. However, it has also shown that there are problems which have not yet been satisfactorily solved. The inadequacies of PROLOG's autonomous control strategy and of its control primitives have been exposed. In addition to the control problems there are problems with the language of logic, such as those associated with the provision of satisfactory Horn clause extensions and of metalanguage facilities. Furthermore, there is a need to establish a logic programming perspective of those types of problems which seem not to fit so naturally within the framework of deduction from a fixed theory. It should be recognised too that logic programming has something to learn from the experience of the structured programming school, for example in the implementation of modularity.

The development of new computer architectures will be of major significance. It can be speculated that the current situation in which logic and functional languages are generally at a disadvantage in efficiency terms relative to

imperative languages will be changed by the advent of parallel processing hardware. The development of languages such as PARLOG shows that logic programming has credibility as a technology for the 'fifth generation' of computing machines. Furthermore, it demonstrates conclusively that logic programming specifies not one but an entire family of programming languages, and that PROLOG should rightfully be seen as but the first realisation of the logic programming concept. In this sense, the description of PROLOG as 'the FORTRAN of logic programming' represents not so much an adverse criticism of PROLOG as an astute perception of the development of computing itself.

REFERENCES

1 Backus, John Can programming be liberated from the von Neumann style? A functional style and its algebra of programs.
CACM August 1978 pp 613 - 640.

2 Burks, A. D., Goldstine H. H., von Neumann, J
Preliminary discussion of the Logical design of an electronic instrument
Princeton, 1946.

3 Booth, A. D., Britten, K. H. V. General considerations in the design of an all-purpose electronic digital computer.
Princeton, 1947.

4 von Neumann, J. The computer and the brain.
Yale University Press, 1958.

5 Report of the Ad-Hoc Committee on Universal Languages: The problem of programming communication with changing machines - a proposed solution.
CACM Aug 1958, p12.

6 Dijkstra, E. W. A primer of Algol 60 programming.
Academic Press 1962.

7 CODASYL COBOL: Final Report.
U.S. Government Printing Office, June 1960.

8 Wulf, William A. Languages and Structured Programs.
In: Raymond T. Yeh, (Ed), Current Trend in Programming Methodology. Prentice-Hall 1977.

9 David, E. E. The Production of Software for Large Systems.
Infotech State of the Art Reports, 1971.

10 McKeag, R. M. Operating Systems.
In: R. H. Perrott, (Ed), Software Engineering. Academic Press 1977.

11 Brooks, F. P. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley 1975.

12 Hoare, C. A. R. Software Engineering: A Polemical Prologue. In: R. H. Perrott, (Ed), Software Engineering. Academic Press 1977

13 Dijkstra, E. W., Dahl and Hoare Structured Programming
Academic Press 1972.

14 Mills, H. D. Top-Down Programming in Large Systems.

In: Debugging Techniques in Large Systems Prentice-Hall 1971, pp 41 - 55.

15 Wirth, N. Programming Development by Stepwise Refinement.
CACM 14, 4 (April 1971) pp 221-227.

16 Sommerville, I. Software Engineering. Addison-Wesley 1983.

17 Dijkstra, E. W. Goto Statement Considered Harmful.
CACM 11, 3 (March 1968) pp 147 - 148.

18 Bohm and Jacopini. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules.
CACM, May 1966.

19 Wirth, N., Jensen, K. Pascal User Manual and Report.
Springer-Verlag 1974.

20 Woodward, P., Bond, S. G. Algol 68-R Users Handbook.

HMSO, London, 1974.

21 Bohl, M., Walter, A. Introduction to PL/1 Programming and PL/C.
Science Research Associates, 1973.

22 Jackson, M. A. Principles of Program Design.
Academic Press 1975.

23 Glaser, H., Hankin, C. and Till, D. Principles of Functional Programming.
Prentice/Hall 1984.

24 Floyd, R. W. Assigning Meaning to Programs.
Proc. Symp. in Applied Maths, Vol 19. Americam Mathematical Society 1967, pp 19 - 32.

25 Hoare, C. A. R. Proof of a Program: FIND.
CACM 14, 1 Jan 1971 pp 39 - 45.

26 Manna, Z., Waldinger, R. J. Towards Automatic Program Synthesis..
CACM 14, 3 March 1971 pp 151 - 165.

27 Liskov, B., Zilles, S. An Introduction to Formal Specifications of Data Abstractions.
In: Yeh, R.T. (Ed) Current trends in Programming Methodology. Prentice-Hall 1977.

28 Burstall, R.M., Darlington, J. Some Transformations for Developing Recursive Programs.
Proc. Int. Conf. on Reliable Software, Los Angeles, Californmia pp 465 - 472.

29 Hoare, C. A. R. The Emperor's Old Clothes.
CACM 24, 2 Feb 1981 pp 75 - 83.

30 U.S. Department of Defense. Requirements for Ada Programming Support Environments: 'Stoneman' 1980

31 Dijkstra, E. W. The Humble Programmer.
CACM 15, 10 Oct 1972 pp 859 - 866.

32 Wirth, N. Programming Languages: How To Assess Them.
In: R. H. Perrott, (Ed), Software Engineering. Academic Press 1977.

33 Treleaven, P. C., Brownbridge, D.R., Hopkins, R. P. Data-driven and Demand-driven Computer Architecture.
ACM Computing Surveys 14, 1 pp 93 - 143 1982.

34 Gurd, J., Watson, I., Glauert, J. A Multilayered Data Flow Computer Architecture.
Department of Computing Science, Univ. of Manchester 1980.

35 Dijkstra, E. W. Co-operating Sequential Processes.
In: Genuys, F., (Ed) Programming Languages. Academic Press, 1968.

36 Hoare, C. A. R. Communicating Sequential Processes.
CACM 21, 8 pp 666 - 677 1978.

37 Brinch-Hansen, Per The Programming Language Concurrent Pascal.
In: Bauer, F. L., Samelson, K. (Eds) Language Hierarchies and Interfaces. Springer-Verlag, 1976.

38 Campbell, R. H., Kolstad, R. B. Path Expressions in Pascal.
In: Forth International Conference on Software Engineering. pp 212 - 215 1979.

39 Chamberlin, D. D. The 'Single-Assignment' Approach to Parallel Processing.
In: Fall Joint Computer Conference p 263 - 269 1971.

40 Kowalski, R. The relation between logic programming and logic specification.
In: Hoare. C.A.R., Sheperdson. J.C (Eds) Mathematical Logic and Programming Languages. Prentice/Hall 1985.

41 Robinson, J.A. A machine oriented logic based on the resolution principle. Journal of ACM 12, 23 - 41, 1965.

42 Hodges, W. H. Logic. Penguin Books 1977.

43 Nilsson, N. J. Problem-Solving Methods in Artificial Intelligence.
McGraw-Hill, New York 1971.

44 Horn, A. On sentences which are true of Direct Unions of Algebras.
Journal of Symbolic Logic 16, pp 14-21, 1951.

45 Kowalski, R. Logic for Problem Solving.
Academic Press 1979.

46 Church, A. A Note on the Entscheidungsproblem.
Journal of Symbolic Logic 1, pp 40-41 (correction ibid. p101-102), 1936.

47 Robinson, J. A. Automatic Deduction with Hyper-Resolution. International Journal of Computer Math. 1, pp 227-234. 1965.

48 Robinson, J. A. Logic: Form and Function.
Edinburgh University Press 1979.

49 Bundy, A. The Computer Modelling of MAthematical Reasoning.
Academic Press 1984.

50 Robinson, J.A. Computational Logic: The Unification Computation.
Machine Intelligence 6, Edinburgh University Press, pp 63 - 72 1971.

51 Chang, C. L., Lee, R. Symbolic Logic and Mechanical Theorem Proving.
Academic Press 1973.

52 Tarski, A. Truth and Proof.
Scientific American 220(6), 63-77, 1969.

53 Herbrand, J. Researches in the theory of Demonstration.
In: van Heijenoort, Ed. From Frege to Godel: a sourcebook in mathematical logic 1879 - 1931. pp 525-81.
Harvard Univ. Press, 1930.

54 Godel, K. Uber Formal Unentscheidbare Satze der Principia Mathematica und verwandter System 1.
English translation in: van Heijenoort, Ed. From Frege to Godel: a sourcebook in mathematical logic 1879 - 1931.
Harvard Univ. Press pp 596-616, 1930.

55 Sheperdson, J. C. The Calculus of Reasoning.
In: Michie (Ed), Intelligent Systems.
Ellis-Horwood 1984.

56 Kowalski, R., Kuehner, D. Linear Resolution with Selection Functiion.
Artificial Intelligence Vol 2, pp227 - 260, 1971.

57 Loveland, D. W. A Linear Format for Resolution.
Symposium on Automatic Demonstration, Lecture Notes in Math 125.
Springer-Verlag pp1-162 1970.

58 Siekmann, J., Stephan, W. Completeness and Soundness of the Connection Graph Proof Procedure.
Innterner Bericht Nr 7/76, Inst. fur Informatik I, Universtat Karlsruhe. 1976.

57 Kuehner, D. Some Special Purpose Resolution Systems.
In: Meltzer & Michie (Eds), Machine Intelligence 7, Edinburgh University Press, pp 117-128. 1972.

58 Bledsoe, W. W. Non-resolution Theorem Proving.
Artificial Intelligence, vol 9, 1977.

59 Clark, K. L., McKeeman, W. M., Sickel, S. Logic Program Specification of Numerical Integration.
In: Clark, K. L., Tarnlund, S. (Eds) Logic Programming.
Academeic Press 1982.

60 Gilmore, P. C. A Proof Method for Quantification Theory.
IBM Journal Res. Dev. 4:28-35, 1960.

61 Wang, H. J. Towards Mechanical Mathematics.
IBM Journal Res. Dev. 4:28-35, 1960.

62 Robinson, J. A. Logical Reasoning Machines.
In: Michie, D., (Ed) Intelligent Systems.
Ellis-Horwood 1984.

63 Prawitz, D. An Improved Proof Procedure.
Theoria 26, pp 102-139, 1960.

64 Sickel, S. A Search Technique for Clause Interconnectivity Graphs.
IEEE Trans. Comptrs. (Special issue on automatic theorem proving) Aug. 1976.

65 Reiter, R. Two Results on Ordering for Resolution with Merging and Linear Format.
J. ACM 18 pp 630-646, Oct 1971.

66 Kowalski, R. A. Logic as a Database Language.
Research Report DoC 82/25, Dept of Computing, Imperial College, London. Revised May 1984.

67 Brand, D. Analytic Resolution in Theorem Proving.
Artificial Intelligence Vol 7 pp 285-318, 1976.

68 Kowalski, R. Predicate Logic as Programming Language.
Proc IFIP 74, North-Holland Publishing Co. Amsterdam, pp569-574, 1974.

69 Roussel, P. PROLOG: Manuel de Reference et d'Utilisation.
Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy, Sept 1975.

70 Robinson, J.A., Sibert, E.E. LOGLISP: Motivation, Design and Implementation.
In: Clark, K.L., Tarnlund, S. (Eds) Logic Programming.
Academic Press, 1982.

71 Nilsson, N. J. Problem-Solving Methods in Artificial Intelligence.
McGraw-Hill, 1971.

72 Gallaire, H., Lasserre, C. Metalevel Control for Logic Programs.
In: Clark, K.L., Tarnlund, S. (Eds) Logic Programming.
Academic Press, 1982.

73 Pereira, L., Pereira, F., Warren, D. User's Guide to DECsystem-10 Prolog.
DAI Occasional Paper 15, Department of Artificial Intelligence, University of Edinburgh. 1979.

74 Clark, K. L., McCabe, F. Programmer's Guide to IC-PROLOG.
CCD Report 79/7, Imperial College, University of London 1979.

75 Clark, K. L., McCabe, F. The Control Facilities of IC-PROLOG.
In: Michie, D (Ed) Expert Systems in the Micro-Electronic Age.
Edinburgh University Press 1979.

76 Hill, R. LUSH Resolution and its Completeness.
DCL Memo No. 78, School of Artificial Intelligence,
August 1974.

77 Dowson, M. A Note on Micro-PLANNER.
In: Campbell, J. A. (Ed): Implementations of PROLOG.
Ellis-Horwood 1984.

78 Sussman, G., Winograd, T., Charniak, E. Micro-PLANNER
Reference Manual (Revised).
MIT AI LAB Memo 203A. 1971.

79 Bruynooghe, M., Pereira, L. M. Deduction Revision by
Intelligent Backtracking.
In: Campbell, J. A. (Ed): Implementations of PROLOG.
Ellis-Horwood 1984.

80 Cox, P. T. Finding Backtrack Points for Intelligent
Backtracking.
In: Campbell, J. A. (Ed): Implementations of PROLOG.
Ellis-Horwood 1984.

81 Pereira, L. M., Porto, A. Selective Backtracking.
In: Clark, K.L., Tarnlund, S. (Eds) Logic Programming.
Academic Press, 1982.

82 McDermott, D., Sussman, G. The CONNIVER Reference
Manual.
MIT AI LAB Memo 259A, January 1974.

83 Davies, J. POPLER - Implementation of a POP-2 based
PLANNER.
In: Campbell, J. A. (Ed): Implementations of PROLOG.
Ellis-Horwood 1984.

84 Davies, J. POPLER 1.5 Reference Manual.
TPU Report no. 1, Edinburgh, 1973.

85 Griswold, R. E., Poage, J. F., Polansky, I. P. The
SNOBOL4 Programming Language.
Prentice-Hall International (Second Edition) 1971.

86 Dijkstra, E. W. A Discipline of Programming.
Prentice-Hall International, 1976.

87 Hogger, C. J. An Introduction to Logic Programming.
Academic Press 1984.

88 Clark, K. L. Negation as Failure.
In: Gallaire, H., Minker, J. (Eds) Logic and Data
Bases.
Plenum Press, New York, pp 293-322. 1978.

89 Turner, S. J. W-Grammars for Logic Programming.
In: Campbell, J. A. (Ed): Implementations of PROLOG.
Ellis-Horwood 1984.

90 Wise, M. J. EPILOG: Re-Interpreting and Extending
PROLOG for a Multiprocessor Environment.
In: Campbell, J. A. (Ed): Implementations of PROLOG.
Ellis-Horwood 1984.

91 Jaffar, J., Lassez, J-L, Lloyd, J. W. Completeness of
the Negation as Failure Rule.
Proc. 8th Int. Joint Conf. on Artificial Intelligence,
Karlsruhe, Germany, 1983.

92 Bowen, K. A., Kowalski, R. Amalgamating Language and
Metalanguage in Logic Programming.
In: Clark, K.L., Tarnlund, S. (Eds) Logic Programming.
Academic Press, 1982.

93 Warren, D. H. D. Higher-order Extensions to PROLOG: Are
They Needed?
In: Michie, D. (Ed) Machine Intelligence 10. pp
441-453. Ellis-Horwood, 1983.

94 McCabe, F. G., Clark, K. L., Steel, B. D. micro-PROLOG
3.1 Programmer's Reference Manual.
Logic Programming Associates Ltd, Fourth Edition. 1984.

95 Clocksin, W. F., Mellish, C. S. Programming in PROLOG.
Springer-Verlag 1981.

96 Clark, K. L., McCabe, F. G. micro-PROLOG: Programming
in Logic.
Prentice-Hall 1984.

97 Colemerauer, A. PROLOG - The Fifth-Generation Language.
Interview in: Europa Management Report, June 1985.
Digital Equipment Corporation, Europe.

98 Kluzniak, F. The 'Marseille Interpreter' - A Personal
Perspective.
In: Campbell, J. A. (Ed): Implementations of PROLOG.
Ellis-Horwood 1984.

99 Fogelholm, R. Exeter PROLOG - Some Thoughts on PROLOG
Design by a LISP User.
In: Campbell, J. A. (Ed): Implementations of PROLOG.
Ellis-Horwood 1984.

100 Clark, K. L., McCabe, F. G., Gregory, S. IC-PROLOG
Language Features.
In: Clark, K.L., Tarnlund, S. (Eds) Logic Programming.
Academic Press, 1982.

101 Kowalski, R. Logic as a Computer Language.
In: Clark, K.L., Tarnlund, S. (Eds) Logic Programming.
Academic Press, 1982.

102 Mellish, C., Hardy, S. Integrating PROLOG in the POPLOG Environment.
In: Campbell, J. A. (Ed): Implementations of PROLOG.
Ellis-Horwood 1984.

103 Sergot, M. A Query-the-User Facility for Logic Programming.
In: Yazdani, M. (Ed) New Horizons in Educational Computing. Ellis-Horwood 1984.

104 Ennals, R., Briggs, J., Brough, D. What the Naive User Wants from PROLOG.
In: Campbell, J. A. (Ed): Implementations of PROLOG.
Ellis-Horwood 1984.

105 Hammond, P. APES: a User Manual.
Department of Computing Report 82/9, Imperial College, London University. 1983.

106 McCabe, F. G., Clark, K. L., Brough, D. R. ZX Spectrum micro-PROLOG Programmer's Reference Manual. Sinclair Research, Cambridge, 1984.

107 McCabe, F. G., Clark, K. L., Brough, D. R. BBC Micro micro-PROLOG Programmer's Reference Manual. Acornsoft Ltd, Cambridge, 1985.

108 Julian, S. Graphics in micro-PROLOG.
M.Sc. Thesis, Dept. of Computing, Imperial College.
London University 1982.

109 Kowalski, R. Logic as a Computer Language for Children.
In: Yazdani, M. (Ed) New Horizons in Educational Computing. Ellis-Horwood 1984.

110 Ross, P. LOGO Programming.
Addison-Wesley 1984.

111 Campbell, J. A. (Ed) Implementations of PROLOG.
Ellis-Horwood 1984.

112 Clark, K.L., Tarnlund, S. (Eds) Logic Programming.
Academic Press, 1982.

113 Boyer, R. S., Moore, J. S. The Sharing of Structure in Theorem Proving Programs.
In: Machine Intelligence 7, Edinburgh University Press.
1972

114 Mellish, C. S. An Alternative to Structure Sharing in the Implementation of a PROLOG Interpreter.
In: Clark, K.L., Tarnlund, S. (Eds) Logic Programming. Academic Press, 1982.

115 Bruynooghe, M. The Memory Management of PROLOG Implementations.
In: Clark, K.L., Tarnlund, S. (Eds) Logic Programming. Academic Press, 1982.

116 Kahn, K. M., Carlsson, M. How to Implement PROLOG on a LISP Machine.
In: Campbell, J. A. (Ed): Implementations of PROLOG. Ellis-Horwood 1984.

117 van Emden, M. H. An Interpreting Algorithm for PROLOG Programs.
In: Campbell, J. A. (Ed): Implementations of PROLOG. Ellis-Horwood 1984.

118 Robinson, J. A. Logical Reasoning in Machines.
In: Michie, D. (Ed) Intelligent Systems. Ellis-Horwood 1984.

119 Lloyd, J. W. Foundations of Logic Programming.
Springer-Verlag 1984.

120 Plaisted, D. A. The Occur-Check Problem in PROLOG.
Int. Symp. on Logic Programming, Atlantic City, pp 272-280. 1984.

121 Bruynooghe, M. Garbage Collection in PROLOG Interpreters.
In: Campbell, J. A. (Ed): Implementations of PROLOG. Ellis-Horwood 1984.

122 Warren, D. DEC-10 PROLOG Efficiency.
In: Michie, D. (Ed) Expert Systems in the Micro-Electronic Age. Edinburgh University Press 1979.

123 Martelli, A., Montanari, U. An Efficient Unification Algorithm.
In: ACM Transactions on Programming Languages and Systems, Vol 4 No 2, April 1982.

124 McCabe, F. G., Gregory, S. Getting Started with IC-PROLOG.
DOC 81/29, Dept of Computing, Imperial College, London. January 1981.

125 Byrd, L., Pereira, F., Warren, D. A Guide to Version 3 of DEC10-PROLOG and Prolog Debugging Facilities.
DAI Occasional Paper 19, Dept of Artificial Intelligence, University of Edinburgh. 1980.

126 Warren, D. An Improved PROLOG implementation which Optimises Tail Recursion.
Proc. of Int. Workshop on Logic Programming. von Neumann Comp. Sci. Soc., Debrecen, Hungary. July 1980.

127 Warren, D. Implementing PROLOG - Compiling Predicate Logic Programs.
Research Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh. 1977.

128 Robinson, J. A, Sibert, E. Logic Programming in LISP.
School of Comp. and Inf. Sci., Syracuse University. 1980.

129 Durham, T. The Best of Both Worlds?
Article in: Computing the Magazine, May 23, 1985.

130 Lloyd, J. W. Foundations of Logic Programming.
Springer-Verlag 1984.

131 Naish, L. An Introduction to MU-PROLOG.
Technical Report 82/2, Dept of Computer Science, University of Melbourne.

132 Gabbay, D. M., Sergot, M. J. Negation as Inconsistency.
Research Report DoC 84/7, Imperial College. 3rd Draft December 1984.

133 Manna, Z. The Correctness of Programs.
J. Computing and System Science, Vol. 3 pp 119-127, 1969,

134 Hoare, C. A. R. An Axiomatic Basis for Computer Programming.
CACM, Vol 4, pp 321, 1969.

135 Dijkstra, E. W. A Discipline of Programming.
Prentice-Hall 1976.

136 Davies, R. E. Runnable Specification as a Design Tool.
In: Clark, K. L., Tarnlund, S. A. (Eds) Logic Programming.
Academic Press 1982.

137 Clark, K. L. The Synthesis and Verification of Logic Programs.
Research Report DoC 81/36, Imperial College, University of London. Revised Sep. 1981.

138 Clark, K. L., Sickel S. Predicate Logic: a Calculus for Deriving Programs.
Proc. 5th Int. Joint Conf. on Artificial Intelligence, Cambridge, Massachusetts, 1977.

139 Hogger, C. J. Derivation of Logic Programs.
Ph. D. Thesis. Imperial College, University of London. 1979.

140 Hogger, C. J. Derivation of Logic Programs.
Journal of the ACM 28(2), 372-422, 1981.

141 Hoare, C. A. R. Programs are Predicates.
In: Hoare, C. A. R, Shepherdson, J. C. (Eds)
Mathematical Logic and Programming Languages.
Prentice-Hall, 1985.

142 McDermott, D. The PROLOG Phenomenon.
SIGART Newsletter 72, July, pp 16-20, 1980.

143 Hansson, A., Tarnlund, S. A. Program Transformation by Data Structure Mapping.
In: Clark, K. L., Tarnlund, S. A. (Eds) Logic Programming.
Academic Press 1982.

144 Brough, D. R., Walker, A. Some Practical Properties of Logic Programming Interpreters.
Research Report 83/34, Department of Computing, Imperial College. University of London December 1983.

145 Warren, D. H. D. Natural Language the PROLOG way.
Interview with Tony Durham in: Computing The Magazine, April 11th 1985.

146 Kriwaczek, F. A Critical Path Analysis Program.
In: Clark, K. L., McCabe, F. G. Micro-PROLOG: Programming in Logic.
Prentice-Hall International, 1984.

147 Warren, D. H. D. Logic Programming and Compiler Writing.
Software Practice and Experience, 10, 2. 1980.

148 Bendl, J., Koves, P., Szeredi, P. The MPROLOG System.
Proc. of Int. Workshop on Logic Programming, von Neumann Comp. Sci. Soc., Debrecen, Hungary, July 1980.

149 Cuadrado, C. Y, Cuadrado, J. L., PROLOG Goes to Work.
Byte Magazine, pp 151-158, August 1985.

150 Campbell, J. A., Hardy, S. Should PROLOG be List or Record Oriented?
In: Campbell, J. A. (Ed) Implementations of PROLOG.
Ellis-Horwood 1984.

151 Burstall, R. M., McQueen, D., Sannella, D. T. HOPE: An Experimental Applicative Language.
Report CSR-62-80, Dept. of Computer Science, University of Edinburgh. 1980.

152 Milner, R. A Theory of Type Polymormhism in Programming.
Journal of Computer and System Sciences 17(3), pp 348-375. December 1978.

153 Mycroft, A., O'Keefe, R. A Polymorphic Type System for PROLOG.
DAI Research Paper No. 211, Department of Artificial Intelligence, University of Edinburgh. 1983.

154 Clark, K. L., Gregory, S. PARLOG: Parallel Programming in Logic.
Research Report DOC 84/4, Department of Computing, Imperial College, University of London. Revised June 1985.

155 Dijkstra, E. W. A Discipline of Programming.
Prentice-Hall International 1976.

156 Clark, K. L., Tarnlund, S. A First-order Theory of Data and Programs.
Proc. of IFIP-77, Toronto, pp 939-944. North-Holland Publ., Amsterdam, 1977.

157 Turner, D. An Admired Combination.
Interview in: Computing the Magazine, May 10th, 1984.

158 Cunningham, R. J., Zappacosta-Amboldi, S. Software Tools for First-order Logic.
Research Report DOC 82/19, Department of Computing, Imperial College, London University 1982.

159 Balogh, K. On an Interactive Program Verifier for PROLOG Programs.
Proc. of Colloquium on Mathematical Logic in Programming. Republished North-Holland Publ., Amsterdam, 1981.

160 Bergman, M., Kanoui, H. Application of Mechanical Theorem-Proving to Symbolic Calculus.
Third Int. Symp. on Advanced Methods in Theoretical Physics. C.N.R.S., Marseille, 1973.

161 Belovari, G., Campbell, J. A. Generating Contours of Integration: An Application of PROLOG in Symbolic Computing.
Proc. 5th Conf. on Automated Deduction. Lecture Notes in Computer Science, Springer-Verlag 1980.

162 Coelho, H. A Program Conversing in Portugese Providing a Library Service.
Ph.D. Thesis, University of Edinburgh. December 1979.

163 Colmerauer, A. Metamorphosis Grammars.
In: Natural Language Communication with Computers. No. 63, Lecture Notes in Computer Science, Springer-Verlag pp 133-189, 1978.

164 Quantification in a three-valued Logic for Natural Language Question-Answering Systems.
Proc. 6th IJCAI, Tokyo, pp 182-187, 1979.

165 McCord, M. L. Using Slots and Modifiers in Logic Grammars for Natural Language.
Technical Report 69A-80. Department of Computer Science, University of Kentucky. 1980.

166 Clark, K. L., McCabe, F. G. PROLOG: A Language for Implementing Expert Systems.
In: Michie, D. (Ed) Machine Intelligence 10.
Ellis-Horwood 1983.

167 Hammond, P., Sergot, M. A PROLOG Shell for Logic Based Expert Systems.
Dept. of Computing, Imperial College, London. 1983.

168 Brough, D., Parfitt, N. An Expert System for the Ageing of a Domestic Animal.
Logic Programming Group, Department of Computing, Imperial College, London. 1984.

169 Yazdani, M. Knowledge Engineering in PROLOG.
In: Forsyth, R. (Ed) Expert Systems. Chapman and Hall, 1984.

170 Hardy, S. PROLOG for Knowledge Engineers.
Tecknowledge Internal Memo. 1983.

171 Bundy, A., et al. MECHO: A Program to Solve Mechanics Problems.
DAI Working Paper No. 50. University of Edinburgh, 1979.

172 Sergot, M. Prospects for Representing the Law as Logic Programs.
In: Clark, K. L., Tarnlund, S. A. (Eds) Logic Programming.
Academic Press 1982.

173 Cory, H. T., et al. The British Nationality Act as a Logic Program.
Logic Programming Group, Department of Computing, Imperial College. 1984.

174 Van Emden, M. H. Chess End-Game Advice: A Case Study in Computer Utilisation of Knowledge.
In: Michie, D. (Ed) Machine Intelligence 10.
Ellis-Horwood 1983.

175 Kriwaczek, F. Some Applications of PROLOG to Decision Support Systems.
M.Sc. Thesis. DoC Report 85/9, Dept of Computing, Imperial College, London. (First pub. 1982), May 1985.

176 Clark, K. L., Van Emden, M. H. The Logic of Two-person Games.
In: Clark, K. L., McCabe, F. G. micro-PROLOG: Programming in Logic. Prentice-Hall International 1984.

177 Gallaire, H., Minker, J. (Eds) Logic and Data Bases.
Plenum Press, New York. 1978.

178 Santane-Toth, E., Szeredi, P. PROLOG Applications in Hungary.
In: Clark, K. L., McCabe, F. G. micro-PROLOG: Programming in Logic. Prentice-Hall International 1984.

179 Roberts, G. M. An Implementation of PROLOG.
M.Sc. Thesis, Department of Computer Science, University of Waterloo. 1977.

180 Michie, D. (Ed.) Introductory Readings in Expert Systems.
Gordon and Breach, New York. 1982.

181 Hammond, P. APES (A PROLOG Expert System Shell): A User Manual.
DoC Report 82/9, Department of Computing, Imperial College, London.

182 Kowalski, R. Logic Programming.
In: Byte Magazine, pp 161-177. August 1985.

183 Sergot, M. J. Programming Law: LEGOL as a Logic Programming Language.
Logic Programming Group, Department of Computing, Imperial College, London. 1980.

184 Michie, D. (Ed.) Intelligent Systems: the Unprecedented Opportunity.
Ellis-Horwood 1985.

185 Kowalski, R. Letter in 'Computing' Newspaper. 13th December 1984.

186 Coelho, H., Cotta, J. C., Pereira, L. M. How to Solve it with PROLOG.
Laboratorio Nacional de Engenharia Civil, Lisbon, Portugal. 2nd Edition 1980.

187 Church, A. The Calculi of Lambda Conversion.
Princetown University Press, N. J. 1941.

188 Kleene, S. C. General Recursive Functions of Natural Numbers.
Mathematical Annals 112, pp727-742, 1936.

189 McCarthy, J. et al. LISP 1.5 Programmer's Reference Manual.
MIT Press, 1962.

190 Turner, D. A. Functional Programs as Executable Specifications.
In: Hoare, C. A. R, Shepherdson, J. C. (Eds) Mathematical Logic and Programming Languages.
Prentice-Hall, 1985.

191 Henderson, P. Functional Programming: Applications and Implementations.
Prentice-Hall International, 1980.

192 Darlington, J., Henderson, P., Turner, D. A. (Eds) Functional Programming and its Applications.
Cambridge University Press, 1982.

193 Darlington, J., Field, A. J., Pull, H. The Unification of Functional and Logic Languages.
Research Report DoC 85/3. Dept of Computing, Imperial College, London University. Feb. 1985.

194 Landin, P. J. The Mechanical Evaluation of Expressions.
Computer Journal 6 (4), pp308-320, 1963.

195 Turner, D. A. A New Implementation Technique for Applicative Languages.
Software Practice and Experience, 9, pp31-49, 1979.

196 Meira, S. R. L. Sorting Algorithms in KRC: Implementation, Proof and Performance.
Computing Lab. Rep. no 14, University of Kent at Canterbury.

197 Darlington, J. Program Transformation.
In: Byte Magazine, pp 201-216, August 1985.

198 Subrahmanyam, P., You, J. H. FUNLOG = Functions + Logic.
Intern. Symp. Logic Programming, IEEE, pp 144-153, 1984.

199 Hardy, S. The POPLOG Programming Environment.
Cognitive Studies Memo 82-05, University of Sussex, 1982.

200 McCabe, F. G. Lambda PROLOG.
Internal Report, Dept. of Computing, Imperial College, London University. (In preparation, Feb. 1985).

201 Sheil, B. A. The Psychological Study of Programming.
Computing Surveys vol 13 no 1 pp 101-120. March 1981.

202 Brooks, R. E. Studying Programmer Behaviour Experimentally: The Problems of Proper Methodology.
Comms. ACM vol 23 no 4, pp207-213. April 1980.

203 Kowalski, R. A. Logic as a Computer Language for Children.
Research Report no 82/23. Dept of Computing, Imperial College, University of London. Feb. 1982.

204 Pirotte, A. High Level Data Base Query Languages.
In: Gallaire, H., and Minker, J. (Eds) Logic and Data Bases. Plenum Press, New York, pp409-436, 1978.

205 De Sarem, H. Programming in micro-PROLOG.
Ellis-Horwood 1985.

206 Conlon, T. Start Problem-Solving with PROLOG.
Addison-Wesley 1985.

207 Ennals, R., Briggs. J., Brough, D. What the Naive User Wants from PROLOG.
In: Campbell, J. A. (Ed) Implementations of PROLOG.
Ellis-Horwood, 1984.

208 Kahn, K. A Grammar Kit in PROLOG.
In: Yazdani, M. (Ed) New Horizons in Educational Computing.
Ellis-Horwood 1984.

209 Minsky, M. L. A Framework for the Representation of Knowledge.
In: Winston, P. (Ed) The Psychology of Computer Vision.
McGraw-Hill, New York, pp211-280, 1975.

210 Article by Hedley Vosey in 'Computing' newspaper, Sep. 29 1985.

211 Cambell, J. A. Three Uncertainties of A.I.
In: Michie, D., Hayes, P. (Eds) Intelligent Systems.
Ellis-Horwood 1985.

212 Conlon, T. Expert Systems: A Resource Pack for Standard Grade Computing Studies.
Department of Computer Education, Moray House College of Education, Edinburgh. July 1985.

213 Kluzniak, F., Szpakowicz, S. PROLOG - A Panacea?
In: Campbell, J. A. (Ed) Implementations of PROLOG.
Ellis-Horwood, 1984.

214 Turner, R. Logics for Artificial Intelligence.
Ellis-Horwood 1984.

215 Tarnlund, S. A. Logic Information Processing.
Report TRITA-IBADB 1034, Dept of Information Processing, University of Stockholm, Sweden. 1975.

216 Hogger, C. Concurrent Logic Programming.
In: Clark, K. L., Tarnlund, S. A. (Eds) Logic Programming.
Academic Press 1982.

217 Clark, K. L., Gregory, S. A Relational Language for Parallel Programming.
Research Report DOC 81/16, Dept of Computing, Imperial College, London University. July 1981.

218 Clark, K. L., Gregory, S. PARLOG: Parallel Programming in Logic.
Research Report DOC 84/4, Dept of Computing, Imperial College, London University. Revised June 1985.

219 Shapiro, E. Y. A Subset of Concurrent PROLOG and its Interpreter.
Technical Report TR-003, ICOT, Tokyo. February 1983.

220 Ueda, K. Guarded Horn Clauses.
Technical Report TR-103, ICOT, Tokyo, June 1985.

221 Gregory, S. How to use PARLOG.
Unpublished Report, Dept of Computing, Imperial College, Londin University.

222 Darlington, J., Reeve, M. J. ALICE: a Multi-Processor Reduction Machine.
In: Proc 5th Conf. on Functional Programming Languages and Computer Architecture, Portsmouth, NH, pp 65-75.
October 1981.

223 Gregory, S. Design, Application and Implementation of a Parallel Logic Programming Language.
PhD Thesis (in Preparation). Dept of Computing, Imperial College, London. 1985.

224 Boden, Margaret A. Piaget.
Fontana Press 1979.

225 Wasserman, I. J. New Directions in Programming.
In: Wallis, P. J. L. (Ed) Programming Technology.
Pergamon Infotech State of the Art Reports. Pergamon 1982.

226 Turner, D. A. Prospects for Non-Procedural and Dataflow Languages.
In: Wallis, P. J. L. (Ed) Programming Technology.
Pergamon Infotech State of the Art Reports. Pergamon 1982.

227 Tarnlund, S-A, Horn Clause Computability.
BIT 17, 215 - 226. 1977.

228 Andreka, H., Nemeti, I. The Generalised Completeness of Horn Predicate Logic as a Programming Language.
Research Report 21, Dept. of AI, Univ. of Edinburgh. 1976.

229 Sebelik, J., Stepanek, P. Horn Clause Programs for Recursive Functions.
In: Clark, K. L., Tarnlund, S. A. (Eds) Logic Programming.
Academic Press 1982.