THE PERFORMANCE EVALUATION OF INTERPRETER BASED COMPUTER SYSTEMS.

SIMON BERNARD JONES.

PhD. Thesis.

June 1981.

Computing Laboratory,
University of Newcastle upon Tyne.

NEWCASTLE UPON TYNE
UNIVERSITY LIBRARY
ACCESSION No.
81-[illegible]105
LOCATION
L2441

ABSTRACT.

This thesis explores the problem of making accurate assessments of the performance of high level language interpreter programs which are embedded in some more complex system. The overall system performance will be determined by all the software and hardware components present; but in order either to analyse and improve particular components, or to select between alternative versions of components, the concept of the performance of individual components is important.

A model is developed for the abstract behaviour of software components playing the role of an interpreter by considering their interaction with the program code which is being interpreted and with the underlying virtual machine which is, in turn, interpreting them. This model enables a flexible definition of performance by relating the interactions in which an interpreter takes part. A methodology is recommended for assessing experimentally the performances defined within such a framework.

The performances of an interesting selection of pseudo-machine and high level interpreter implementations of Lispkit and Prolog are then assessed and conclusions drawn.

ACKNOWLEDGEMENTS.

I would like to express my thanks to Dr. Peter Henderson for his supervision of this research, and for frequent and valuable constructive criticism.

I would also like to thank my parents for their financial and moral support during my long years of study, both undergraduate and postgraduate, and to thank my wife Pethong for her unfailing confidence and encouragement.

This research was financed by the United Kingdom Science Research Council, and latterly by the University of Newcastle upon Tyne.

CONTENTS.

		Page.
1.	Introduction.	0
2.	The Lispkit language.	6
2.1	Expressing computations as functions	8
2.2	An operational model for Lispkit evaluations	18
3.	The Prolog language.	23
3.1	Expressing computations as relations in a logic framework	26
3.2	An operational model for Prolog computation	44
4.	Multi-level structure in interpreter based systems.	51
4.1	The structure of multi-level systems	52
4.2	Two models for the behaviour of components within multi-level interpreter systems	62
4.3	Comments	75
5.	The performance of components of multi-level systems	77
5.1	Defining the performance of a component	81
5.2	A methodology for empirical performance assessment	90
5.3	Comments	96
6.	Lispkit and Prolog machines: Structure and performance	97
6.1	A Lispkit pseudo-machine	100
6.2	A Prolog pseudo-machine	135
6.3	Conclusions and comparison of the Lispkit and Prolog pseudo-machines	162

Page.

7.	Higher level interpretation of Lispkit and Prolog.	165
7.1	Interpreting Lispkit	165
7.1.1	A preliminary note on the Lispkit program syntax required by the interpreters	166
7.1.2	First Lispkit interpreter, LISPINT1	168
7.1.3	Second Lispkit interpreter, LISPINT2	183
7.1.4	Comments	191
7.2	Interpreting Prolog	192
7.2.1	The syntax of interpreted programs	194
7.2.2	The interpreter, PROLOGINT	194
7.3	Conclusions and comments on the Lispkit and Prolog interpreters	204
8.	Summary and conclusions.	205

Appendices.

A.	A brief history of Lispkit	211
B.	A pseudo-machine implementation of Lispkit	213
B.1	An s-expression syntax for Lispkit	214
B.2	Compiling Lispkit programs for the LM	215
B.3	The Lispkit Machine	218
B.4	An AlgolW software realisation of the LM	222
C.	A brief history of Prolog	227
D.	A pseudo-machine implementation of Prolog	230
D.1	An s-expression syntax for Prolog	231
D.2	Compiling Prolog programs for the PM	233
D.3	The Prolog Machine	236
D.4	An AlgolW software realisation of the PM	247
E.	Tabulated results from test executions on the Lispkit Machine	256
E.1	Assessing the Lispkit Machine	256
E.2	Assessing LISPINT1 and LISPINT2	257

Page.

F. Tabulated results from test executions on the Prolog Machine 266

F.1 Assessing the Prolog Machine 266

F.2 Assessing PROLOGINT 267

G. Bibliography. 274

Chapter 1 - INTRODUCTION.

Introduction.

The observations, theories, experiments and results reported in this thesis arose from an informal exploration of the implementation of Lispkit and Prolog by means of interpreters and pseudo-machines. The exploration was primarily intended to enlarge my personal understanding of these and related languages and of their particular implementation problems. However, the urge to pass judgement on the merits and demerits of individual systems, and to compare and contrast different systems, is very strong, and my thoughts turned in this direction. It became clear that it is not a simple matter to make accurate assessments and fair comparisons of the performance of interpreters and pseudo-machines; the influences of different hardware, different languages (for implementation of the interpreters and pseudo-machines), different programming styles within the same implementation language, different overall system structures, and so on, must ideally be factored away, and the characteristics of the interpreters and pseudo-machines must be presented in some abstract but meaningful form. Great care must be taken in the design of experiments, and in the statement of conclusions, but I was not aware of any other work which attempted to clarify this matter; ad hoc techniques prevail in assessments and comparisons reported in the literature, and to me this felt less than satisfactory in such an important subject.

In particular my attention was drawn to the performance comparison of implementations of DEC-10 Prolog and Stanford Lisp, reported in Warren (1977) and Warren et. al (1977). The comparison arrives at some surprising conclusions concerning the relatively high efficiency of Prolog, and I did not feel satisfied with the simple benchmarking technique used in the assessment. On the other hand, both the comparison of SASL and Lisp reported in Turner (1979), and the comparison of various Prolog systems reported in Moss (1980), show attempts to factor out the influences of the environment; in the former case this is by counting high level operations within the software, and in the latter case by performing initial experiments to "normalise" the execution speeds of different machines. However, in neither case are the considerations underlying such techniques developed as an independent topic.

Thus this thesis has grown to be an attempt to identify and to clarify some of the issues in the analysis and assessment of the performance of interpreter systems. The results of the investigation will show that a comprehensive and reliable performance assessment is, in general, very difficult to achieve.

Lispkit and Prolog are two representatives of the large number of experimental (very) high level programming languages currently aiding research in many branches of computer science; for example, they serve as testbeds for the development of advanced programming techniques, as prototypes for future programming languages, and as languages for use in experiments on automatic program synthesis, transformation, verification and the definition of semantics.

For experimental high level languages it is common to make implementations which consist either of an interpreter (usually written in some high level language, and accepting programs in "source code" form), or a software "pseudo-machine" (usually written in some high level language, and accepting programs for execution in an intermediate compiled form); pseudo-machines are clearly also interpreters of some language. The interpreter and pseudo-machine approaches expedite the implementation process in the research and teaching environments, and may themselves lead to a deeper understanding of the implemented languages. In the applications, or systems engineering, environment these two approaches may also ease the modification and maintenance of high level language systems.

Of course, an interpreter for a high level language may itself be written in some high level language (possibly the very one it is designed to interpret) and be executing on some interpreter or pseudo-machine. Thus the general picture which arises is that of programming systems which are constructed as multiple levels of interpreters and pseudo-machines. This has been true for some time in the hardware field with the concept of microprogrammed processor architectures.

It is to be expected that the price to be paid for an extra layer of interpreter software between a program and the hardware is a loss of anticipated execution speed of the program. Perhaps the best that can be expected is that the time for interpretation of a program, ti, is a linear function of the time for execution of the program (if suitably compiled) directly on the hardware, td:

$$ti = a*td + b$$

But, with one or more layers of potentially sophisticated interpretation, the relationship might very well be much worse than linear, for example quadratic or exponential:

$$ti = a*td*td + b*td + c$$

or $$ti = a*exp(td) + b$$

The design decisions taken in individual interpreters and pseudo-machines will critically determine the performance characteristics. The decisions and their consequences are of theoretical interest in the research environment, but they are of practical importance in all environments; the commercial user is paying money for CPU time, and the researcher or student is trying to obtain useful results whilst maintaining patience and interest.

Hence, whether for the sake of research interests, or of systems engineering, it is necessary to be able to make accurate assessments of the impact of design decisions on the performance of systems and their components, and to be able to make fair and meaningful comparisons between alternative versions of components.

The complexity of the software of multi-level interpreter systems makes an empirical assessment of performance more feasible than an theoretical approach, though experimental results can be expected to yield insight into mechanisms to be considered later by a theoretical analysis. In such assessments of the performance of individual interpreters it is important to obtain characteristics which are not distorted by the inclusion of the characteristics of other components in the system; for example, the results of assessing the performance of an interpreter (and hence the interpretation scheme which it realises) should depend neither on the test programs which were being interpreted, nor on the machine which was executing the interpreter.

To this end, what is necessary is some way of defining the performance of an interpreter component in isolation from the remainder of any system in which it is used, and to have some practical method of obtaining good assessments of such performances. Subsequent chapters suggest some solutions to these problems, and the suggestions are put into practice to assess a selection of Lispkit and Prolog implementations.

Chapters 2 and 3 set the scene for a discussion of high level language interpretation by presenting the Lispkit and Prolog languages in some detail.

Chapters 4 and 5 examine one approach to the treatment of multi-level interpreter systems and their performance, and present a formal model for defining the performance of individual, isolated components of systems. The model

also clarifies the relationship of the performance of individual components to the performance of compound (multi-level) components, and of the system as a whole.

Chapter 5 also describes a methodology for the empirical assessment of the performance of components of systems, and in Chapters 6 and 7 the methodology is applied to pseudo-machine implementations of Lispkit and Prolog, and to high level interpreters for Lispkit and Prolog.

The appendices contain a little more background information on the origins of Lispkit and Prolog, details of the two pseudo-machine implementations, and the tabulated results of experiments described in Chapters 6 and 7.

CHAPTER 2 - THE LISPKIT LANGUAGE.

The Lispkit Language.

In this chapter I shall describe the Lispkit Lisp variant, which can be considered as representative of the family of functional, recursive languages for manipulating symbolic data structures.
The origins of Lispkit are outlined in Appendix A.

The first part of the chapter is concerned with the use of Lispkit as a notation for expressing computations as functions from symbolic data structures to symbolic data structures. The second part of the chapter will cover an operational model for the evaluation of Lispkit programs.

Lispkit has been implemented as a high level pseudo-machine, for which Lispkit programs are compiled into an intermediate machine code. Details of the design of the machine and of the compilation are to be found in Appendix B. Chapter 6 outlines the behavioural characteristics of the Lispkit machine, and an empirical performance assessment is made. In Chapter 7 two Lispkit interpreters are programmed in Lispkit and their performance is assessed.

A much fuller exposition on the subject of functional programming is to be found in Henderson (1980). Indeed, this chapter and Appendix B are little more than a personal reiteration of the same material, and in particular I must give credit for the design of the Lispkit language and machine to Henderson.

The Lispkit language is based on the concepts of using symbolic data structures as basic values, of organising a computation as nested expressions rather than as a sequence of assignments, and of using recursive functions to handle the tree-like symbolic data structures.

The following is an example of a complete Lispkit program. It illustrates the basic constructs of the language, and can serve to focus attention in the language description which follows:

```
find  whererec
      find ( x,l) = if eq(l, NIL) then  1
                    else
                    if eq(head(l),x) then 1
                    else 1 + find ( x, tail (l))
```

This program searches a list to find the first occurrence, if any, of a particular value. The result of the program is the position of the value in the list, expressed as an integer, or, if the value is not found, then an integer which is one greater than the length of the list. The program requires two input arguments, the value and the list, called x and l respectively. The second to fifth lines constitute a function definition, of the recursive function "find". Simple expressions appear throughout the program, for example "eq(l, NIL)" and "1+find (...)", and in fact the entire program is an expression. In line 5 the function "find" is applied explicitly to some arguments : "find (x, tail (l))".

The action of the function find is programmed to cover three cases: If the list l is empty (equal to NIL) then x is certainly not contained in it, and result is 1 (since the length of an empty list is 0). Otherwise if the first member of l (head(l)) is equal to x then the result is 1. Otherwise the first member is not equal to x, and the result must be one more than the result of finding x in the remainder of the list, that is 1 + find (x, tail(l)).

2.1 Expressing computations as functions.

2.1.1 Symbolic data structures.

The first feature of Lispkit which it is necessary to describe is also one of the key factors contributing to the great programming power of this and a host of other experimental very high level languages. The class of data values which a Lispkit program is designed to manipulate is a particular form of symbolic expressions (<u>s-expressions</u>) (McCarthy (1960)). S-expressions subsume the integers and simple string constants (<u>symbols</u>) as atomic values (<u>atoms</u>), and include all binary trees (acyclic) which have atoms at the leaves. All compound objects (data structures) are represented as such binary trees. In the representation on paper of s-expression values integer atoms will be written as expected, symbols will be short unquoted strings of upper case letters (and possibly digits in other than the first position), and a compound object (a tree) will be represented as (a.b) where a and b are the representations of the left and right subtrees respectively. Hence the following simple syntax:

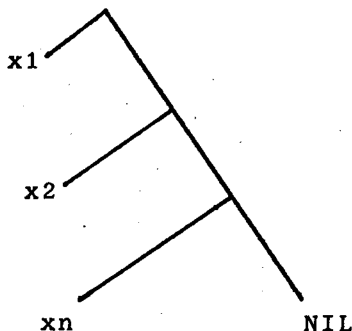
```
s-expression :: = atom | (s-expression.s-expression)
 atom ::  = integer | symbol
```

In practice the above "dot notation" will be cumbersome and visually confusing, so we shall adopt the usual convention that certain s-expression structures may be simplified (unambiguosly). The following two rules for rewriting s-expressions will apply:

(i) An s-expression of the form
    (a b ... d.(e f ... g.h))
    may be written as
    (a b ... d e f ... g.h)

(ii) An expression of the form
(a b ... d.NIL)
may be written as
(a b ... d)

One consequence of these simplifying rules is that we shall choose to represent a <u>list</u> of objects x1, x2,..., xn by the binary tree



which can be written as (x1.(x2.(...(xn.NIL)...)))
and conveniently rewritten as (x1 x2 ... xn).

Some examples of s-expressions (in various degrees of simplification):

(1 2 3 4)   (A B. (C D.NIL))   - "two lists"
((1 2 3) (A B C) (X Y Z)) - "a list of lists"
((A.1) (B.4) (H.J)) - "a list of dotted pairs"
(((X1. Y1). (X2.Y2)). ((X3.Y3). (X4.Y4)))
   - "a complete binary tree"

2.1.2 Simple computational expressions.

Lispkit provides a variety of computational <u>expressions</u> which may be used in a program to construct new s-expressions from previously available ones, to extract subexpressions from s-expressions, to create new integer atoms by arithmetic, to compare atoms for equality and inequality, to test for atomicity of an s-expression, and to select between two alternative expressions depending on a logical test.

In order to construct new s-expressions we have the computational equivalent of the dot notation:

```
          cons (a, b)
```

builds a new s-expression whose left and right subtrees are a and b, respectively, so

```
          cons ((1 2), (3 4)) = ((1 2).(3 4))
                              = ((1 2) 3 4)
```

The car operator will extract the left subtree of an s-expression and cdr the right subtree. Car and cdr are undefined if their operand is an atom. If the operand represents a list then car and cdr correspond to the intuitive operations "head" and "tail" respectively. I shall use car, head, cdr and tail interchangeably, and as appropriate. Also it will be convenient to use "the car/head" and "the cdr/tail" in place of the phrases "the right subtree" and "the left subtree". Hence

```
          car (((X.Y).Z))= (X.Y)
          head ( (A B C ) ) = A
          cdr ((X.(Y.Z))) = (Y.Z)
          tail ((A B C )) = (B C)
          tail ((A)) = tail ((A.NIL)) = NIL
```

Arithmetic facilities are basic, but comprehensive. The operators +, -, *, div, rem are available (defined only for integer atom operands), and are written in the familiar infix notation. Examples do not seem necessary.

The logical operator eq is used to test for the equality of atoms. The expression eq (a,b) has, as value, the atom T if a and b are identical integers or symbols, and the atom F otherwise (including if either a or b is not an atom at all). The operator $\leqslant$ tests for that inequality between two integers. $a \leqslant b$ has the value T or F, as appropriate, and is undefined if either a or b is not an integer.

There is a single unary predicate atom which has the value T if its argument is an atom, and F otherwise.

Examples of logical expressions and their values:

```
  eq((1 2), (1 2)) = F          eq(head((1 2)), 1) = T
  eq(tail ((1)), NIL) = T
    1≤2 = T                     2≤1 = F
    atom (A) = T                atom ((1 2)) = F
    atom (tail ((1))) = atom (NIL) = T
```

To select between two alternatives a conditional expression construct is used:

<u>if</u> econd <u>then</u> etrue <u>else</u> efalse

has the value of etrue if the value of econd is T, and the value of efalse if econd is F. For example

<u>if</u> eq(A,B) <u>then</u> car ((X.Y)) <u>else</u> cdr((X.Y)) = Y

<u>if</u> atom (1) <u>then</u> 2 <u>else</u> 3 = 2

<u>if</u> car((T F F)) <u>then</u> THIS <u>else</u> THAT = THIS

It will often be the case that an expression will contain two or more identical subexpressions. The use of a <u>where</u> construct enables common subexpressions to be replaced by instances of the same <u>variable</u> name, and this variable is associated with the value of the subexpressions. This technique will be familiar from common mathematical practice. For example, if we have a 3-list (list of 3 values) in which the first value is T or F, and indicates whether we are to select the second or third value respectively, then the selection operation for (T SECOND THIRD) can be written as

```
if car (threelist) then head(tail(threelist))
                   else head(tail(tail(threelist)))
where threelist = (T SECOND THIRD)
```

This is an expression, having the value SECOND, in which the variable "threelist" has replaced 3 occurrences of (T SECOND THIRD) in the conditional expression.

A *where* expression may introduce more than one variable - variables will be written in lower case letters (possibly with digits in other than the first position). The variables introduced are the *defined variables*, and the expressions which give their values are the defining expressions. The expression to which the *where* is attached is the *qualified expression*. A restriction introduced with *where* expressions is that the scope of use of the defined variable names is within the qualified expression only. The defining expressions may not refer to locally defined variables, but only to variables in more global scopes (enclosing *where* expressions). In the following example

```
      (... x ... where  x = e1
                  and   y = ... z ... x ...)
      where  z = e2
       and   x = e3
```

the first occurrence of x is resolved to the variable which is associated with the value of e1. In the definition of y, the variables z and x are resolved to e2 and e3 respectively.

Note that parentheses may be used freely in order to disambiguate nested expressions, as I have done in the example above to indicate the nesting of the *where* expressions.

2.1.3 Defining functions.

There is one form of expression, not mentioned above, which has the special property that its "value" represents a

function, or operator, which may be applied to constant, input or computed values. This value should not be thought of as an s-expression, which would normally be expected as the result of an expression evaluation,but simply as an object which can receive arguments, perform a computation with them and return a result.

Two new forms of expressions are necessary : for function definition and function application.

A function definition has the form

λ(x, y, ..., z)e

and is known as a lambda expression (commonly the function value of such an expression is also known as a lambda expression or simply a function). λ is the operator symbol which signals the type of expression. The parenthesised list of variable names x, y, ..., z is the ordered list of formal parameters or arguments; when the function is applied it will require an ordered list of actual parameters of exactly the same length. The final component, e, stands for any Lispkit expression (including a lambda expression or constant), which may contain occurrences of the argument variables. e is the body of the lambda expression.

A function application has the form

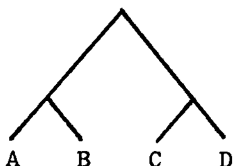
f(a, b, ..., d)

in which f is an expression whose value is a lambda expression, and a, b, ..., d are expressions whose values will be computed in order to provide the lambda expression with its actual parameters. Note that f is not restricted to being an explicit lambda expression (though this is one obvious possibility). The value of a function application expression is found by evaluating the arguments a, b, ..., d and f, substituting a, b, ..., d into the body of the function for occurrences of the corresponding formal parameter variables, and evaluating the substituted body. This particular parameter mechanism is known as "calling by value".

As a simple example consider a function of 4 values which builds a balanced binary tree with these values at its leaves. The defining lambda expression is

λ(w, x, y, z) cons(cons(w, x), cons(y, z)).

Referring to the function which this expression represents as f, we can build the tree



with the application

f(A,B,C,D).

The form of an application is similar to that of various Lispkit primitive expressions, such as cons(a,b), but no confusion will arise if we are careful about function names.

Despite the fact that function values should not be thought of as s-expressions, they may be treated in a computation in many of the same ways. In particular they may be built into data structures and extracted at a later stage, they may be associated with local variable names in <u>where</u> expressions (useful if a function is needed in several places in an expression), and they may be given as actual parameters to applications and returned as results.

To demonstrate the local naming of a function we may write a lambda expression representing a function which builds an 8 leaf balanced tree, and incorporate the 4 leaf tree building function from above:

λ (s,t,u,v,w,x,y,z)(cons(f(s,t,u,v),f(w,x,y,z))

<u>where</u>
f = λ(w,x,y,z)cons(cons(w,x),cons(y,z)))

Here the body of the 8 leaf function is an expression qualified by the definition of f as the 4 leaf function.

This example has introduced the need for a more refined description of variable scope. The variable names in the formal parameter list of a function have scope only within the body of the function. When resolving which formal parameter or defined variable a variable occurrence refers to, the variables in a formal parameter list have the same status as defined variables qualifying the body of the function. A variable is resolved to the nearest enclosing definition in the textual levels of <u>where</u> and lambda expressions. Hence, in the example above, the variables w, x, y, z mentioned in the body of f refer to the formal parameters of f, and the variables s, t, ..., z in the qualified expression of the 8 leaf function all refer to the parameters of this function.

Definitions of functions may be mixed with other definitions in a <u>where</u> expression since function names are no different from other variables, but retaining the previous restriction that the definitions may not refer to each other.

To conclude this description of function definition facilities I must show how recursive functions are treated. Recursive functions are a natural way of expressing computations involving tree structures.

The scope rule for the names defined in a <u>where</u> expression implies that recursive functions cannot be defined directly since the body of a function would need to mention the name of the function explicitly. This rule also prevents groups of functions defined together from referring to each other (mutual recursion).

To overcome this restriction a variant of the *where* expression is available. The *whererec* expression is used to qualify an expression by definitions in a similar way to a *where* expression but with the extra feature that the scope of the defined variables has been expanded to include the defining expressions in addition to the qualified expression. Thus function definitions may refer explicitly to themselves, to each other, and to any non-function variables defined in the same *whererec* expression. One restriction remains, the defining expressions for non-function variables may not mention *any* variables (function or otherwise) defined in the local group. Note that it appears that *whererec* subsumes the purpose of *where* but in the implementation to be described in Chapter 6 (and in Appendix B) it is slightly more expensive to execute *whererec* than *where*.

For example, the following lambda expression accepts as input a list of values and returns a similar list in which each value has been "doubled up" into a "consed" pair:

```
λ(l) (doubleall (l)
        whererec doubleall = λ(l) if eq(l,NIL) then NIL
                                   else cons(double(head(l)) ,
                                                 doubleall (tail(l)))
        and  double =  λ (x)   cons(x,x) )
```

The body of doubleall refers to itself and to double.

A notational convenience which I shall allow myself is to represent function name definitions in a more familiar style. In a *where* or *whererec* expression

f = λ (x,y,...,z)e

will frequently be written as

f(x,y,...,z) = e

For example double (x) = cons(x,x).

2.1.4 Complete programs.

The most recent three complete examples, constructing a 4 leaf balanced tree, an 8 leaf balanced tree and doubling the members of a list, have each had the form of a lambda expression in which all variable occurrences could be resolved either to the parameters of the lambda expression or to a definition (or further parameter list) contained within the body of the lambda expression.

Thus the three examples satisfy the requirements to be complete Lispkit programs. Each is self-contained (no further definitions are necessary in some more global context to give values to any variables), and each is an expression whose value is a function.

A Lispkit program execution then consists of two steps; firstly the program is evaluated to produce the function which it represents, and secondly this function is applied to an actual parameter list constructed from the desired input data values.

Two common examples of simple functional programs:

1) A program which maps a list of items (x y ... z) onto a list with the items in reverse order (z ... y x)

```
reverse
whererec  reverse (l)= if eq(l,NIL) then NIL
                       else append(reverse(tail(l)),
                                   cons(head(l),NIL))
and append (l1,l2) = if eq(l1,NIL) then l2
                     else cons(head(l1), append(tail(l1),
                                                l2))
```

This program embodies the list reversal algorithm frequently called "naive reverse", since it requires a rather large number of function applications and cons operations to perform its task. The program does not have the form of a lambda expression, but it nevertheless does represent a function, since the value

of the program is the value of the variable reverse, and reverse is defined to be a function.

2) A program to sum the integers in the range from m to n inclusive (assuming m $\leq$ n):

```
sum
wherererec  sum(m,n) = if eq(m,n) then  m
                       else m+sum(m+1,n)
```

To illustrate the power of the Lispkit function handling consider the following alternative definition of the doubleall function above:

```
doubleall
whererec  doubleall (l) = map(double)(l)
  and     double   (x) = cons(x,x)
  and     map   (f)    = (g whererec
                            g(l) = if eq(l,NIL) then NIL
                                   else cons(f(head(l)),
                                             g(tail(l))))
```

In this program we have a "higher order function" map, which accepts as parameter a function f of one argument, and constructs from this a function g which will apply f to each member of a list. The definition of doubleall is unusual in that it constructs a function which will double the members of a list (the subexpression map (double)), and then applies this function to l. In a larger program map could be used to provide simple definitions for a variety of more sophisticated functions such as doubleall.

2.2 An operational model for Lispkit evaluations.

In this section I shall describe a method by which Lispkit expressions may be evaluated.

This method will have two important uses; firstly it will provide an easily grasped guideline which a programmer may use when

creating and understanding his programs, secondly it will immediately suggest an implementation of Lispkit, which will thus obtain the results which the programmer expects from his programs.

An important remark which I must make before commencing with the description is that the operational model is essentially what is known as a "call-by-value" method. The general approach, which will become obvious later, is that first the subexpressions of an expression are evaluated completely, and then the main expression operator is applied to these results. This is a straightforward approach, but unfortunately it makes impossible certain programs which would yield useful results with a more subtle evaluation mechanism. However, the class of programs which will succeed with this call-by-value model contains a very large number of useful and interesting members. The primary group of programs missing concerns the direct representation of the processing of finite portions of infinite data structures - but this omission will not generally be a problem.

2.2.1 Evaluating in an environment.

If we look into a Lispkit program and extract an arbitrary expression then within that expression we will be able to identify two different categories of variables. One category contains those variables which are defined (or appear in formal parameter lists) within the expression. The other category contains variables which are associated with values somewhere within the more global context from which the expression has been extracted.

When a Lispkit expression is evaluated it will be arranged that the values of global variables are contained within an

environment description. Hence an expression should be thought of as being "evaluated in an environment". The value of a variable, when required, will be looked up in the environment. *Where* expressions, *whererec* expressions and function applications will each, as part of their action, modify the environment in which their subexpressions are evaluated.

An environment is an ordered set of *associations* between variable names and values, with the notion of "more recent" determining the ordering of the associations. Associations are added to the environment at the most recent end, and the search for the value of a variable commences at the most recent end. This mechanism ensures adherence to the scope rules for defined variables and formal parameters.

2.2.2 Simple computational expressions.

The very simplest expression is a variable name, which is evaluated by looking up its value in the environment.

The next simplest expressions to evaluate are those with the operators cons, car, cdr, +, -, *, *div*, *rem*, eq, ⩽ and atom. To evaluate one of these expressions in environment E, the operand expressions are evaluated in E and then the operator is applied to the values obtained. To evaluate the conditional expression *if* e1 *then* e2 *else* e3 in environment E, we first evaluate e1 in E and then, depending on its value, evaluate either e2 in E or e3 in E.

Where expressions involve extending the environment. To evaluate a *where* expression in environment E the defining expressions are evaluated in E, the values are associated with the corresponding defined variables and these associations are added to E to give E'. The qualified expression is then evaluated in E'.

2.2.3 Lambda expressions, function applications and *whererec* expressions.

Evaluating a lambda expression in environment E results in an object called a *closure*, which contains a record of the lambda expression itself and of E. The environment recorded by the closure is used again when the function is applied. A closure is the representation of a function value.

In order to evaluate an application f(a,b,...) in environment E the expressions f, a, b,... are evaluated in E. The value of f will be a closure containing an environment Ec, a formal parameter list and a body. The values of a,b,... are associated with the corresponding variables in the formal parameter list and these associations are added to Ec to give E'. The body expression is evaluated in E' to give the result of the application.

To evaluate a *whererec* expression in environment E requires rather a curious action, but one which nevertheless is accomplished in the software implementation of Lispkit described in Appendix B. We construct a new environment E' by adding to E the associations obtained from evaluating the defining expressions *in E'*. Note the use of E' and not E - it is this which enables recursive function bodies to access themselves, since the closures will have recorded E' and not E. The qualified expression is then evaluated in E'.

2.2.4 Complete programs.

A programmer initiates a Lispkit program execution by supplying a program text P and some data values A. The program P is evaluated in the empty environment (hence the requirement that

it must be self-contained). The closure which results is then applied to A in the normal way for a function application, where A contains the values of the evaluated actual parameters. The value returned by this application is the final result of the program execution.

CHAPTER 3 - THE PROLOG LANGUAGE.

The Prolog language.

In this chapter I shall describe Prolog, which is one particular variant in the family of languages based on statements in logic. The origins of Prolog are outlined in Appendix C - these lie in work on automatic theorem proving. In recent years there has been a growing community of research prototypes for such programming systems, of varying degrees of complexity and purity, and with differing fields of application in mind. From an examination of the fundamental properties of programming in a logic style, and from a general knowledge of the state of the art in designing and implementing logic programming systems I have made an attempt to capture the essential aspects in the design of a language and implementation for direct computation ("symbolic structure crunching", rather than data base handling or an expert system). I have chosen to use the name Prolog to refer to the language in order to maintain the link with common terminology, although I should actually use "my variant in the family of logic programming languages".

This chapter will cover the expression of computations in the logic style of Prolog, and an operational model for the execution of Prolog programs.

Prolog, like Lispkit, has been conveniently implemented as a high level pseudo-machine, for which Prolog programs must be compiled into an intermediate machine code. Details of the design of the machine and of the compilation are to be found in Appendix D. Chapter 6 includes an outline of the behavioural properties of the implementation, and an empirical performance assessment is made. In Chapter 7 a Prolog interpreter is written in Prolog and its performance is assessed.

The style of description of logic programming in Prolog which follows is, I believe, a novel one for this particular subject. I have attempted to produce a "bottom-up" approach to the required concepts and tools, in the hope that this may make it easier for a novice to grasp the use of Prolog for computation. The presentation follows the same pattern as that of Lispkit; firstly a description of data objects (for this purpose I have introduced the notion of su-expressions), secondly the operations by which new values and results are generated, and lastly the control of these operations by a program. Kowalski (1979) gives a very broad coverage of the application and reasoning behind logic programming styles.

Prolog follows a significantly different style of programming from Lispkit. In Lispkit a computation is expressed as a function which is used to map input values to result values. In Prolog, on the other hand, a computation is expressed as a relationship which states how acceptable input values and output values are intended to be related to each other. Equivalently, this relationship can be viewed as a specification or predicate which is satisfied by acceptable patterns of input and output values. An important distinction is that the functional style of Lispkit embodies the notion of an implicit direction of computation, we supply some input values and the result is computed. However, the precise assignment of input and output roles to arguments in Prolog programs does not occur until the program is executed. Hence Prolog programs may be written largely independently of the flexible input/output roles of their arguments.

Prolog is based on the concepts of symbolic data structures as basic values, pattern matching between data structures, and clauses which are implications enabling relations between data structures to be defined in terms of other relations. Clauses are grouped together as the cases of predicates, and each relation between data structures is defined by one such predicate. Hence a Prolog program is a collection of mutually recursive predicates, and it is the task of program execution to discover instances of data structures which satisfy certain specified predicates. The formal approach to Prolog takes a slightly different viewpoint, and this is mentioned briefly in Appendix C.

The following complete Prolog program is a direct reprogramming of the Lispkit "find" example from the start of Chapter 2. It illustrates the basic constructs available in Prolog, and will focus attention for the language description which follows:

```
query (x,l,n) ← find(x,l,n)
find (x, NIL, 1) ←
find (x,(x.l),1) ←
find (x,(y.l),n) ← ¬eq(x,y),find(x,l,m),add(m,1,n)
```

The resulting position of x in the list l has been assigned a name, n. The query in the first line states that there are three input/output arguments, x,l and n, and that they are related by the predicate find. Find is defined by three cases, each of which is an implication with the antecedents (or conditions) on the righthand side of ←, and the consequent on the lefthand side. The first case states that when any value

x is sought in the empty list, the position value must be 1. The second case states that when a value x is sought in a list whose first value is equal to x, the position value must be 1. The final case states that if a value x is sought in a list whose first member is not equal to x, then the position value n must be 1 greater than the position m when x is sought in the remainder of the list. Note the use of the symbolic data structure patterns (x.l) and (y.l) to name and extract the component parts of values. This is in direct contrast to the use of explicit constructors and selectors (cons, car, cdr) in Lispkit.

3.1 Expressing computations as relations in a logic framework.

3.1.1 Symbolic data structures and pattern matching.

Prolog programs are intended to handle binary tree data structures of exactly the same form as those of Lispkit, with one small, but important, change. The change is to introduce a rather unusual form of atom into the s-expression syntax. The new values, which may appear at the leaves of binary trees, represent subtrees for which values are not known - although more information may become available concerning the values of these subtrees from other sources during a computation. The new values I shall call "unknowns"; they will be represented on the written page as an asterisk prefixing a positive integer, and the new type of symbolic data structures I shall call "su-expressions" (for "s-expressions with unknowns"):

```
su-expression ::= atom | (su-expression.su-expression)
atom ::= integer | symbol | *positiveinteger
```

I shall adopt the same simplifying rules for su-expressions as for s-expressions, and the same use of the terms car, cdr, head and tail (although these will not appear as parts of the Prolog language).

Some comments on what we gain from the introduction of unknowns are appropriate:

(i) An implication of the explicit numbering of unknowns is that two or more occurrences of the same unknown are intended to represent the same (unknown) value. For example

(*1.*1)

represents a tree for which the car and cdr are identical. Note that different unknowns do not necessarily represent different values - we can explicitly specify equality, but not inequality.

(ii) An su-expression containing one or more unknowns can be seen in two lights; it can be viewed as a partial description of some particular data structure, or as a representation of the entire set of values which have the same structure. Both views may be useful at different times.

The primary facility in Prolog for the computation of new values is through the elaboration of the subtrees represented by unknowns, thus extending the known part of the su-expressions. Note that this is very similar to the concept of assignment, the difference being that the operation is a very controlled one - unknowns are extended, but known values are never replaced by other knowns. Elaboration, or extension, of data structures is achieved by unification, which is a sophisticated form of pattern matching.

Two su-expressions are unified by finding elaborations for some or all of the unknowns in the expressions, such that the elaborated expressions are identical. The simplest set of elaborations is a substitution usually called the <u>most general unifier</u> of the expressions. It may be the case that there is no most general unifier for two expressions and so the attempted unification <u>fails</u> and the expressions are not elaborated - for example, if a number or symbol in one expression is at a position corresponding to a different number or symbol in the other expression.

The unification of two su-expressions may be described conveniently by the following informal recursive procedure. The procedure is a simultaneous prefix order walk over the two expressions to be unified ("visit the roots, then the left subtrees in prefix order, then the right subtrees in prefix order"):

> To unify su-expressions se1 and se2:
>
> If both se1 and se2 are dotted pairs then unify the car of se1 with the car of se2, and then unify the cdr of se1 with the cdr of se2;
> Otherwise, if either se1 or se2 is a dotted pair and the other is a number or symbol then the entire unification fails,
> Otherwise, if se1 and se2 are identical numbers or symbols then there is nothing to be done,
> Otherwise, if se1 and se2 are unequal numbers or symbols then the entire unification fails,
> Otherwise, if se1 is an unknown and se2 is a dotted pair, number or symbol (or vice versa) then the unknown is elaborated to be identical to the non-unknown,

Otherwise, the remaining case, both se1 and se2 are unknown, and se2 is elaborated to be the same unknown as se1.

(Note that whenever a unification fails, then the values of the su-expressions remain as they were before unification started)

I must mention several more straightforward properties of unknowns which are essential to the unification operation:

(i) Unknowns have scope covering both the trees which are being unified, and in general an unknown represents the same value in all trees in which it appears. In particular a series of unifications may cause subtrees of a large number of su-expressions to become "linked" by equality.

(ii) An obvious property of an unknown which appears in several su-expressions is that all the occurrences should be elaborated as a result of a unification which elaborates the unknown in one of the trees.

The description of unification in the preceding paragraphs may give a complex appearance to an operation which is actually very simple. However, despite the simplicity, unification is very powerful, and its consequences are subtle - it is this which makes a long discussion desirable.

A series of examples will make the unification operation clearer:

(i) Unifying (A.*1) and (*2.B)
yields the matches *1 = B and *2 = A
and the su-expressions are elaborated to (A.B).
If our interest was in the su-expression (A.*1)
then our knowledge of it has been extended
to (A.B).

(ii) Unifying (A.*1) and (B.C) fails, since the
symbols A and B are not equal.

(iii) Unifying (A.*1) and (A.(1 2 3.*2)) yields
the match *1 = (1 2 3.*2). So (A.*1) is
elaborated to (A.(1 2 3.*2)), but *2 is
not elaborated.

(iv) Unifying (*1 A) and (*2 *2)
yields the matches *1 = *2 and *2 = A
hence both su-expressions are elaborated to
(A A).

(v) Unifying (*1 A) and (*2 *1)
yields the matches *1 = *2 and *1 = A
and hence both su-expressions are elaborated
to (A A).

(vi) Unifying (*1 A) and (B *1) fails, since *1
cannot match both A and B simultaneously.

(vii) A more interesting example:
Suppose that we are interested in the
su-expression

(A.*1)

and we unify firstly (A.*1) and (A.(B.*2)),
and then (X.*2) and (X.(C.*3)).
The first unification elaborates the expression
of interest to

(A.(B.*2)).

The second unification matches *2 with
(C.*3), hence elaborating (X.*2) to (X.(C.*3)),
and simultaneously elaborating the expression
of interest to

(A.(B.(C.*3)))

This final example shows how several unifications, matching su-expressions with common unknowns, can extend our knowledge of particular su-expressions by incorporating information indirectly from other sources.

3.1.2 Computational use of unification to satisy specifications and relations.

The preceding examples have shown, in a rather contrived way, how unifying an su-expression of interest with another pattern may extend our knowledge of that expression, either by failing to match or by elaborating unknowns (if any).

To illustrate the power of unification in a useful computational context consider unifying various expressions (assumed not to contain the unknown *1) with the pattern (*1.*1). This pattern is a template for an su-expression whose car and cdr are equal. The template can behave as a generator of symmetrical trees, for example

((A B).*2) is elaborated to ((A B).(A B))

and ((A *2).(*3 B)) is elaborated to ((A B).(A B)),
or as a verifier of symmetry for example

((A B).(A B)) will succeed with no further elaboration,

and ((A B).(C B)) will fail.

Thus the template (*1.*1) can be considered as a specification of symmetrical trees, or as an equality relation between the two members of a dotted pair, and can be employed as a data verifier or as a result generator.

It is clear that unification provides a very powerful means of transferring information and generating results within a computation, and for this reason Prolog

requires little more than unification in order to perform useful computations.

One final point about unification is that it should be used with great care when unifying two su-expressions which contain the same unknown. Some of the examples above were of this form, but they were safe. However, consider the following example.

Unifying (A.*1) with *1 yields the match *1=(A.*1).

The unknown *1 thus becomes a self-referential data structure, which could be considered as either an unbounded tree, or as a cyclic structure. That is not a problem, but if the structure were subsequently unified with another unbounded tree then the unification procedure might be thrown into a non-terminating recursion.

Unification can be defined in such a way as to detect and warn against the creation of a cyclic structure, using an "occur check", but it is expensive to implement. Since many useful programs will not encounter this problem, the occur check will be omitted from further consideration.

A program in Prolog is a description of su-expression patterns, and of unifications which are required between these internally generated patterns and a list of externally supplied su-expressions (the input). The result of such a computation is an elaborated form of the input expressions.

3.1.3 Conditions - requests for unification.

The basic unit of a program which requests a unification operation is a <u>condition</u>, which has the form

    p(a,b,...,d)

where p is the name of a <u>predicate</u>, and a,b,...,d are <u>actual argument</u> su-expression patterns.

Each su-expression pattern represents the value of an su-expression, but <u>variable</u> names are used wherever an unknown would otherwise be shown in asterisk notation. The reason for this is quite simple. Most parts of a Prolog program will be executed several times during a computation (for example, if a predicate is recursive), and, as with languages such as Algol, the local variables in a section of code need to be fresh ones at each execution. If unknowns were represented in the asterisk notation then this would not be the effect at all. Predicates and their cases are described a little later, and the scope of each variable is exactly the case in which it appears- at each execution of a case, each variable is associated with a fresh, unique unknown.

The use of su-expression patterns provides notational simplicity in the extraction of the components of input arguments, and the combination of results to form output arguments (for example, the find program at the beginning of the chapter uses the pattern (x.l) to separate the head and tail of a list).

A predicate is itself a collection of su-expression patterns and unification requests, which define a relation or specification that the actual arguments are required to satisfy. The definition of predicates is described below.

A condition is a <u>predicate</u> <u>call</u>, and is a request that the named predicate apply unifications to the actual argument values in order to elaborate and/or verify their values. Three outcomes are possible for the condition:

(i) The predicate call may succeed without elaborating the arguments - they satisfy the specification which it represents.

(ii) The predicate call may succeed with elaboration of the arguments - our knowledge of the argument values has been extended.

(iii) The predicate call may fail - the arguments have not matched successfully during some unification, and their values have not satisfied the specification represented by the predicate. The arguments are never elaborated in this case.

Some examples of conditions:

(i) Suppose that the predicate named "sympair" verifies that its single argument is a symmetrical dotted pair, then

sympair (((A.B).(A.B))) will succeed without elaboration,

sympair (((A.B).x)) will succeed, elaborating x to (A.B),

sympair (((A.x).(x.B))) will fail.

(ii) Suppose that the predicate named "swap" verifies that its two arguments are each dotted pairs, but with the car and cdr swapped, then

swap ((A.B),(B.A)) will succeed without elaboration,
swap ((A.B),x) will succeed, with x elaborated to (B.A),
swap ((A.B),(C.A)) will fail.

3.1.4 Calling primitive predicates, and negated conditions.

Conditions, su-expression patterns and defined named predicates form the necessary core of Prolog programming, and they are adequate for a variety of interesting programs.

However, for many more practical applications further facilities are desirable. Arithmetic, for example, could be programmed explicitly as relations on lists of digits representing numbers, but it is much more convenient to have arithmetic available directly.

For this purpose a selection of primitive (or "evaluable") predicates are provided. These are called upon from conditions in exactly the same way as defined predicates.

Integer arithmetic is provided by predicates for addition (add), subtraction (sub), multiplication (mul), division (div), and remainder (rem). Each of these is a three argument predicate which relates its arguments in the obvious way. Informally:

add (x,y,z) succeeds if x+y=z and fails otherwise,
sub (x,y,z) succeeds if x-y=z and fails otherwise,
mul (x,y,z) succeeds if x*y=z and fails otherwise,

div (x,y,z) succeeds if x _div_ y=z and fails otherwise,
rem (x,y,z) succeeds if x _rem_ y=z and fails otherwise.

Clearly the arguments x,y and z, must be either numbers or unknowns. There are, however, practical restrictions on the use of these predicates which are necessary to permit a straightforward implementation. The five arithmetic predicates can be used in their verification role with no problem, but the restrictions apply to the generation of results. Add and sub can compute any one unknown from two knowns. Mul will compute z given x and y, and will compute x(or y) given y and z (x and z) provided that the ratio y/z (x/z) is an integer. Div and rem will compute z if given x and y. In all other cases the result is undefined and the computation cannot proceed.

Note that a computation which _cannot proceed_ is different from a unification which _fails_, as the latter determines a definite choice between alternatives. This will become clearer in the operational model.

Leq is a two argument predicate which terminates the computation if either argument is unknown. If both arguments are numbers and satisfy the inequality $\leqslant$ then leq succeeds, otherwise it fails:

leq (x,y) succeeds if $x \leqslant y$ and fails otherwise.

The predicate eq is used to test explicitly for the equality of two arguments which must be atoms. If the arguments are equal atoms then eq succeeds, if either argument is unknown the computation is terminated, otherwise eq fails. Eq is not strictly necessary since equality constraints are easily expressed in Prolog. However it is convenient in conjunction with negated conditions, which are described below.

The final primitive predicate is atom, which succeeds only if its single argument is a number or symbol - an unknown is not considered to be an atom for this purpose. Atom fails if its argument is a dotted pair, and terminates the computation if it is an unknown:

atom(x) succeeds if x is a number or x is a symbol, and fails otherwise,

eq(x,y) succeds if atom(x) and atom(y) and x=y, and fails otherwise.

The final tool necessary for practical programming in Prolog is the capability to negate the success or failure result of a condition. Such a condition will be prefixed by the negation symbol, ¬ .

Execution of an unnegated condition can have three distinct types of result (excepting a termination of the computation). These are described above.

A negated condition in some sense reverses the success or failure interpretation of the unnegated condition. If a condition c succeeds with no elaboration, then ¬c fails, if c fails then ¬c succeeds with no elaboration, and if c succeeds with elaboration then the computation cannot proceed reliably as either success or failure as the result is inconclusive in determining the truth of ¬c (see Appendix C).

Examples of negated conditions:

¬ mul (2,4,7) will succeed,

¬ add (1,2,x) will terminate the computation,

and assuming the same definition of sympair as above ¬sympair (((A.B).(A.B))) will fail.

3.1.5 Defining predicates.

The enigmatic circularity of this description of the Prolog language must now be closed with a discussion of the definition of the named predicates which are called upon by conditions.

A predicate must impose on given actual argument values a particular specification, and must verify and/or elaborate those values by unification.

A predicate has a name and consists of a group of one or more alternative cases, where each case has the form of a logical implication:

p(f1,f2,...)← c1,c2,...

p is the name of the predicate, (f1,f2,...) is a list of formal argument su-expression patterns, and c1,c2,... are a group of conditions. Variables appearing in the su-expressions of the formal arguments or conditions have scope which is exactly the case in which they appear - so unknowns throughout the case may be linked by the constraints of equality. Thus elaborations occurring in conditions may modify the formal argument values.

A predicate call will succeed if the actual argument values from the call satisfy any one of the cases of the predicate. Actual arguments satisfy a case if they unify successfully with the formal arguments, and if each of the conditions in the case also succeeds. Hence the nature of a case as an implication becomes clear:

"The arguments f1,f2,... satisfy (are related by) the property p if the conditions c1,c2,... are satisfied".

If a case has no conditions on the right hand side

```
p(f1,f2,...)←
```

then the actual/formal unification succeeds or fails without further qualification . This form of case is referred to as an <u>assertion.</u> "The arguments f1,f2,... unconditionally satisy p".

The term "predicate" reflects the role of a program as a verifier of su-expression values, though the term "relation" would also be appropriate. In fact it will be convenient to refer to predicates as "relating the values of their arguments", thus reflecting the duality of the programs.

Predicates in Prolog serve a parallel purpose to functions in Lispkit. They provide a means of abstracting meaningful specifications from a body of code and of describing them separately, and a powerful method for the description of computations involving tree-like data structures. Unifications parallel the actions of passing arguments and results, and the case structure of a predicate parallels conditional testing in Lispkit.

Examples of predicate definitions:

(i) The pattern (x.x) can be used to define the predicate sympair which is satisfied by a symmetrical dotted pair:

```
sympair((x.x))←
```

This is a predicate consisting of only one case, an assertion, with only one argument. It should be read as "All dotted pairs whose car and cdr are identical are symmetrical pairs".

(ii) The predicate

revsympair((x.y),(y.x)) ← sympair(x), sympair(y)

states that "For all su-expressions x and y, if x and y are both symmetrical pairs, then (x.y) and (y.x) are related by the reversed symmetrical pair property".

(iii) If the 2x2 matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ is represented by the list of lists ((a b)(c d)) then the assertion

transpose (((a b)(c d)),((a c)(b d))) ←

could be used to compute or check the transpose of a matrix. The condition

transpose (((1 3)(2 4)),x)

causes x to be elaborated to ((1 2)(3 4)).

(iv) Predicates with more than one case are appropriate when several alternative argument structures are acceptable to the specification. To specify that two arguments are related if either of them is a symmetrical pair:

eitherpair (x,y) ← sympair (x)
eitherpair (x,y) ← sympair (y)

(v) The most common situation in which a multicase predicate is appropriate is when defining a recursive predicate. At least two cases will be required - one is a non-recursive (base) case, and the other contains the recursive call. To verify that each member of a list is a symmetrical pair:

```
eachsym (NIL) ←                  "base case"
eachsym ((first.rest)) ← sympair (first),eachsym(rest)
```

These two cases state, respectively, that

Each member of the empty list is certainly a symmetrical pair (following the usual convention for empty conjunctions),

and Each member of the list (first.rest) is a symmetrical pair if first is a symmetrical pair and each member of rest is a symmetrical pair.

(vi) As an example containing many of the Prolog features which have been described, consider the following predicate "remove" which relates three arguments x (assumed atomic), l1 and l2 (both lists of atoms) if l2 could be obtained from l1 by deleting all occurrences of x:

```
remove (x, NIL, NIL) ←
remove (x,(x.l1),l2) ← remove(x,l1,l2)
remove (x,(y.l1),(y.l2)) ← ¬eq(x,y),remove(x,l1,l2)
```

The cases state, respectively, that

Removing (any) x from an empty list NIL must result in the empty list,

and If the result of removing x from the list l1 is l2, then similarly the result of removing x from (x.l1) must be l2,

and If x and y are different atoms, and removing x from the list l1 gives l2, then the result of removing x from (y.l1) must be (y.l2).

In its present form the remove predicate, given A and (A B A C) as its first two arguments, specifies that the third argument must be (B C). The negated condition ¬ eq(x,y) is crucial to this.

If it were deleted then for A and (A B A C) the predicate is satisfied if the third argument has <u>any combination</u> of the As removed - the results possible are (A B A C), (B A C), (A B C) and (B C).

The latter example shows clearly how Prolog programs may be regarded less as algorithms, and more as statements in logic which specify the properties and relationships of data structures.

3.1.6 Complete programs.

A program in Prolog consists of a complete set of the predicate definitions required for a particular computation (complete in the sense that each of the non-primitive predicates named in conditions is defined), plus a distinct single case predicate with the name "query" which specifies the computation to be performed.

A computation is requested by supplying as input a list of su-expression values. The query predicate is called to verify and/or elaborate the input values (actual arguments).

If the query cannot be satisfied by the input values then the final result is an indication of failure.

If the query succeeds then zero or more elaborations of the inputs exist which satisfy the query. The final results are these elaborated forms of the input values.

The following complete program can be used to verify that two lists are the reverses of each other, or can generate elements of either list from the other:

```
query (x,y)<-reverse (x,y)
reverse (NIL,NIL)<-
reverse ((x.l1),l2)<- reverse(l1,l3),append(l3,(x),l2)
append (NIL, l, l)<-
append ((x.l1),l2,(x.l3))<- append(l1,l2,l3)
```

With inputs (1 2 3), *1 the unknown *1 will be elaborated to (3 2 1).

With inputs (*1 2 *3),(1 *2 3) the unknowns *1, *2 and *3 will be elaborated to 1,2 and 3 respectively.

If the query is modified to

```
query (x)<-reverse (x,x)
```

then the program will check for palindromic lists:

With input (1 2 3) the query will fail.

With input (1 2 *1) the unknown *1 will be elaborated to 1.

With input (1 2.*1) the query will succeed with an infinite number of elaborations for *1, each of which gives a structure which will make the list (1 2.*1) palindromic. The simplest of these values are

(1), (2 1), (*2 2 1), (*3 *3 2 1),
(*4 *5 *4 2 1), (*6 *7 *7 *6 2 1), etc

in which the unknowns *2,*3,*4,*5,*6,*7 have been generated by the su-expression patterns of the program, to stand in place of unknown values which are required for the structure of the results.

3.2 An operational model for Prolog computation.

The execution of a Prolog program with given input values has the nature of a controlled exploration of a space of alternative unifications and elaborations. Alternative paths exist due to the presence of predicates with multiple cases, and the notion that a predicate call succeeds if the actual arguments satisfy any one of the cases independently.

Thus there are two rather independent aspects of Prolog execution: the construction of any particular alternative computation, and the control of the exploration of alternatives. The former is more direct and intuitive, and I shall describe this first. The latter is more abstract, it is a backtracking process, and will be brought into the description gradually - a detailed grasp of this is not necessary for successful programming.

In the construction of a computation we shall be concerned with the local variables of predicate cases, the creation of su-expression values from su-expression patterns, and the order of unifications within cases.

3.2.1 Local variables and environments.

As a computation proceeds predicate cases are entered and su-expressions must be built from patterns which contain variables. Actions within the execution of a case take place in the context of a local environment which contains associations between the local variables of the case and values for them.

When a local environment is created, as the execution of a case is started, each variable has its value initialised to some unknown which is <u>not yet in use</u> in the computation. Subsequent unifications may elaborate these values or use them in the elaboration of other unknowns.

Su-expression values are built from patterns (in the formal arguments or conditions) by including the values from the current local environment of variables named in the patterns.

Note that, unlike Lispkit, environments are only local to cases, and patterns cannot refer to variables in any other cases. External values can only be accessed by the unification between actual and formal arguments.

3.2.2 Executing conditions (predicate calls).

A condition p(a,b,...) is executed in a local environment E by building a list of actual argument su-expressions according to the patterns a,b,... (and referencing the values of variables in E), and then calling the predicate p to verify or elaborate the actual argument values. The call to p may either succeed (with or without elaborating the arguments) in which case the condition succeeds, or it may fail (the arguments are not elaborated) and the condition fails also. If the arguments have been elaborated then the values in E have also been elaborated, and results have been returned by the predicate call.

For example, if E associates x,y and z with (A.B), *2 and *3 respectively then the condition sympair((x.(y.z))) will build an actual argument ((A.B).(*2.*3)) and will call sympair, which will succeed and elaborate y to A and z to B.

As may be expected from the description earlier, a negated condition has a very similar effect, but if the predicate call fails then the condition succeeds, if the call succeeds without elaboration then the condition fails, and if the call succeeds with elaboration then the entire execution is terminated prematurely. As a consequence of this it can be seen that a negated condition can never return results, it can only be used to verify that particular values do not satisfy a predicate.

3.2.3 Primitive predicates and defined predicates.

The execution of a primitive predicate call is very straightforward. The values of the actual arguments are examined to check that they satisfy the specification of the predicate (as described previously), any unknowns which can have their values computed are elaborated, and the call succeeds, fails or terminates the computation, as appropriate.

The execution of a call of a defined predicate is rather more complicated. For any particular alternative computation <u>one</u> of the predicate cases is selected for execution - when the predicate is first called the first case is selected, and subsequent alternative computations will select successive cases. Hence cases are selected in <u>the order in which they appear</u>. This selection sequence is an important rule to remember when writing programs.

A selected case proceeds by selecting a fresh, unique unknown for each variable in the case, and associating these with the variables in the new local environment E. The formal argument su-expressions are built and are unified with the actual argument values. If this unification fails then the selected case fails and the next alternative computation is tried. If the unification succeeds then the conditions of the case are executed, in E, from left to right until either they are all satisfied (and so the case and predicate call succeed), or any condition fails and the next alternative computation is tried. The left to right rule for conditions is also very important to remember.

When a condition fails, the "next alternative computation" will be found by trying any remaining choices in a previous condition of the case. If there are no previous conditions or remaining choices then the case itself fails.

When a case fails, the "next alternative computation" will be found by selecting the next case of the predicate for execution. If there are no more cases, then none of the cases has been satisfied, and the predicate call itself fails.

3.2.4 Decision points and backtracking.

The pattern of execution described is a depth first scanning of the nested predicate calling structure of the program, and a left to right scanning of the predicate calls in each predicate case. However, at each call of a defined predicate there is a choice of cases and execution has reached a decision point at

which only one case can be selected for execution.

As execution proceeds each decision point is noted as it is passed, together with a record of the entire computation state at that point. This may seem to be an extremely extravagant and complex operation; however the language will be implemented using a garbage collected list space (as is usual for very high level languages such as Lispkit and Prolog) and thus saving the computation state is not expensive in time or effort, as it amounts to noting a small collection of pointers.

Whenever a predicate case or condition fails, and hence demands that an alternative computation be found, the record of decision points and states can be used to backtrack - that is to roll the computation back to a recent decision point. The state of the computation is restored to that recorded at the most recent decision point, and computation proceeds with the next choice of predicate case. If all the choices at the most recent decision point have been exhausted then that decision point is discarded and the computation backtracks to the next most recent decision point - this corresponds to a predicate call failing completely and then the condition which called it being failed.

By this method of backtracking the computation does not have to return to the beginning each time a failure occurs. The method implements what is essentially a parallel execution, of the alternative cases in a predicate, by a sequential execution.

3.2.5 Complete programs.

An execution is started by supplying actual argument values to a call of the query predicate. At this point there is no backtrack record, as no decision points have been passed.

If the query fails then the input values cannot satisfy the program. In this case all decision points have been discarded from the backtrack record, as the program tries all alternative computations before failing. Hence the computation must terminate with only failure as a result.

On the other hand the query may succeed (either with or without elaborating the arguments), and there may be decision points remaining in the backtrack record (if there are none then the computation has finished). Any remaining backtrack records may point to alternative elaborations of the input arguments, or maybe only to an exhaustion of the search space with no further solutions. In general more solutions will be of interest, and these can be coaxed from the program by artificially backtracking to the most recent decision point in the normal way.

Care must be employed in the ordering of predicate cases and conditions, as may be deduced from the first to last case selection rule, and the left to right condition execution rule. The problems which may arise concern calling primitive predicates with an impermissible combination of unknowns, or unexpected infinite recursions. As an example of the latter problem consider activating the reverse and append definitions, given earlier, by the query

query (l) ← reverse(l,(1 2 3)).

The first recursive call of reverse has two unknowns as arguments and behaves thenceforth as an inexhaustible source of ever larger mutually reversible templates for lists. After a few tries the correct elaboration for 1, (3 2 1), will be found to satisfy the query. However, if we backtrack in order to exhaust the search space then the source of templates never empties, the append call continually fails and backtracks, and no second solution or eventual failure is forthcoming!

This concludes the discussion of the operational model for Prolog execution.

CHAPTER 4 - MULTI-LEVEL STRUCTURE IN INTERPRETER BASED SYSTEMS.

Multi-level structure in interpreter based systems.

In this chapter I shall outline an approach to analysing the run-time structure and behaviour of complex computer programs. I shall call the structural models "multi-level interpreter systems". The name arises from the types of systems of programs to which I shall be primarily interested in applying the analysis; these programs will fall into two general classes: firstly, compiled high level language programs executing on a high level machine simulator (often implemented by a program written in lower level language), and secondly, high level programs in source code executing on an interpreter (often itself written in a high level language and executing as a program in the first class). Chapter 1 discussed the reasons why it is significant to study programming systems of this form.

Clearly the components of such systems and their relationships could become quite intricate. In order to overcome this potential complexity I propose to introduce in this chapter constrained structural and behavioural models for multi-level interpreter systems.

In the first part I shall introduce a simple diagrammatic notation with which to represent the structure and name the components of systems. The diagrams should be treated as little more than instructions for assembling systems from their component hardware and software blocks, though the precise arrangement of the blocks is determined by some knowledge of their run-time interactions; consequently, the description of the notation makes much reference to the concept of the interpretation of one component by another.

In the second part of the chapter I shall present two semi-formal analyses of the behaviour of multi-level interpreter systems. These are the Interpretation model and Abstract Execution model, which clarify the interpretative details of the first part of the chapter. Subsequent chapters will exploit the properties of the two models in order to define the concept of the performance of a component of a multi-level system, and to devise experimental procedures for assessing performance empirically.

4.1 The structure of multi-level systems.

In practice, programming systems are invariably composed of at least two components: one hardware component, a machine, and one software component, a machine language program residing in the memory of the machine. We can distinguish an active component, the machine (a performer of actions), from a passive component, the program (a source of instructions for the active component). Often the system will be composed of a hardware component and several software components, within which there will be a discernible hierarchy of services provided and interpretative actions performed; this will be especially true of the systems to be studied in later chapters of this thesis. A software component which interprets and carries out the commands of another component is similarly active in some sense, and the component which is being interpreted is passive.

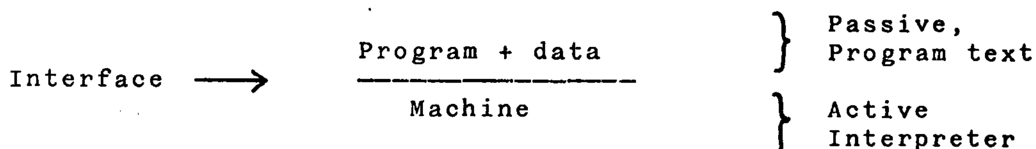
For the purposes of structuring a multi-level system, each active/passive pair of components is separated by an interface. Each interface usually separates two physically distinct components which have been brought together in constructing the system; for example a hardware machine and its software program, or an interpreter and the program which it interprets.

In general a complex programming system will contain several interfaces of interest, and a notation is necessary which allows the representation of such systems.

4.1.1 The single interface.

An obvious choice for a notation to represent a single interface is to draw a horizontal line for the interface, and to name the passive and active entities, above and below the line respectively.

For example:



Bearing in mind that the diagram must cover the entire structure of the system, I have, in the example, included both a program and its data above the line, and all the machinery (microprogram, hardware, etc) required to execute the program below the line. No other information is required in order to complete the execution.

The operational view that I would like to take of this configuration is that the program and data together describe some computation in terms of concrete actions which can be performed only by the lower level machine. The program can take no action of its own - everything must be done by the machine, and the machine is responsible for interrogating the program to obtain its directions.

The active entity below an interface will be referred to as an interpreter, a machine, or a virtual (or high level) machine.

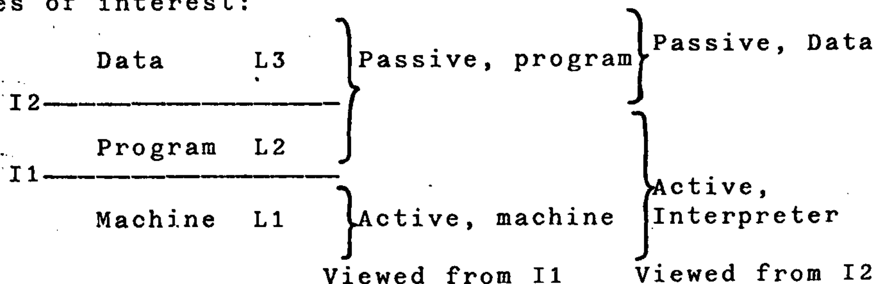
The passive entity above an interface will be referred to as a program (since it will usually correspond intuitively to just that), or occasionally as data (since it is exactly that to the interpreter below the interface).

4.1.2 Multiple interfaces.

For complex systems in which there are several obvious choices for interfaces it may be constructive to consider more than one at a time. The interfaces will then clearly delimit the parts of the system which we are interested to treat as single components.

The diagrammatic notation can be extended simply to include one horizontal line per interface. Note that the multiple division of the system follows a strict vertical stacking scheme - no networks are allowed.

For example, the typical execution of a program, on some machine's hardware with some data, has two interfaces of interest:



The interfaces and levels have been given labels to assist in discussion.

Each interface divides the system into passive and active entities - the single interface philosophy applies separately at each interface. Hence, from the point of view of I2 everything below it (program + machine) forms a single interpreter for what lies above (the data).

This suggests that a typical program can be considered to be a very high level language interpreter for programs in a language consisting, maybe, of only a few numbers.

Operationally, the data contains instructions for work to be done by the program, and this work is achieved by instructing the machine. Hence the concept of multi-level interpretation.

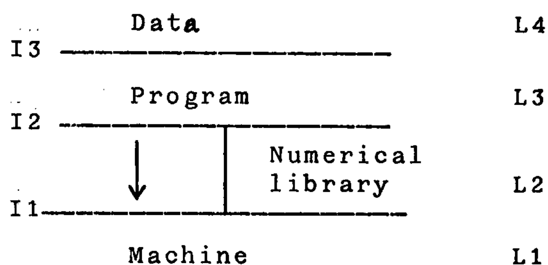
4.1.3 A refinement - "interpreter extensions".

I will introduce one small refinement into the notation scheme.

It will be useful to distinguish one special case of an interpreter level. This is an interpreter in which some of the actions made available at the upper interface are implemented by the interpreter as a complex series of actions at the lower interface, but in addition some of the actions available at the upper interface are already known to, and are provided by, the machine below the lower interface; the lower machine performs the latter actions directly.

This is intended to represent the rather common situation in which a program (compiled into machine language form) is loaded onto a machine together with a collection of "library subroutines" which act as a run-time support package. The program then proceeds mainly by executing machine instructions directly, but occasionally by calling one of the subroutines to perform some more sophisticated action.

A level of interpreter of this nature will be shown diagrammatically by dividing the level into left and righthand sides. The lefthand side will contain a vertical arrow to show that some instructions are executed directly at the lower interface, and the righthand side will name the "package" of non-primitive facilities, for example:



Levels of interpretation of this special kind will not prove to be particularly significant in the following discussion; for example they will receive no special treatment in the second part of this chapter, or in the definition of performance. However, it will be necessary to bear them in mind during practical performance assessment if the characteristics of the active part of the level are to be assessed.

4.1.4 Comments.

Although conceived separately, and for a different purpose, the multi-level interpreter scheme which I have described has a great similarity to the multi-level system description given by Anderson, Lee and Shrivastava (1978). They consider multiple levels of interpreters with well defined interfaces, and also the concept of interpreter extension which corresponds to the refinement outlined above. However, their treatment of interpreter extensions differs somewhat from mine; the new level is considered to be an extension of the next interpreter level below, and to be under the control of that level, whereas I wish to treat it as an independent interpreter. Despite the slight dissimilarity I shall adopt the term "interpreter extension".

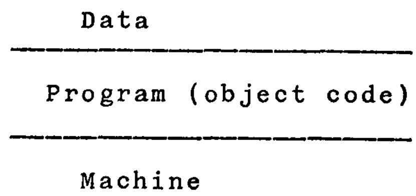
In general, suitable choices for single components of multi-level systems will be individual software programs, designed and developed as integral units. However, it will also prove to be convenient to group together such programs to make compound components, or to identify interfaces within programs and to divide the programs into two or more levels (for example, isolating a group of subroutines providing a common service to a main program).

4.1.5 Examples.

Before proceeding to give a formalised description of the behaviour within multi-level interpreter systems, I shall show a range of multi-level decompositions of typical systems. The examples illustrate the styles of decomposition which will be used in subsequent chapters.

4.1.5.1 Simple program.

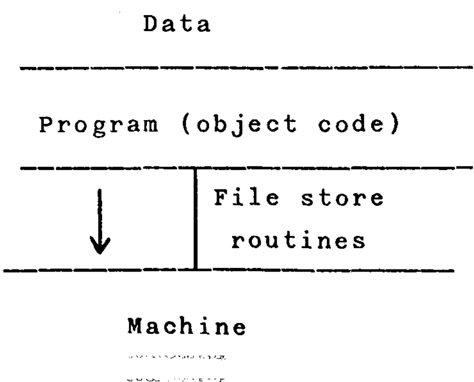
The first example is almost the simplest situation conceivable. A compiled program, given data and executing on the bare hardware of a machine.



This system decomposition is appropriate in all normal programming configurations. In general the program might include several levels of interpretation, which we are not interested to distinguish. Also the data might include further levels of program text and data which are to be interpreted by the program.

4.1.5.2 Executing with a file store.

The normal working environment rarely makes it possible to run programs on completely bare hardware. Usually there is some form of operating system providing greater control over the machine, and extending the machine's facilities. Typically the operating system will provide file storage and possibly a virtual memory. Considering only the provision of a file store, this may be represented by an interpreter extension above the hardware level:



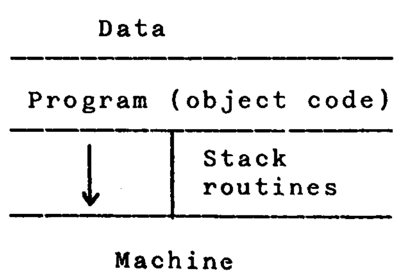
The implication here is that the program executes machine instructions, or calls file store routines which execute machine instructions on its behalf. The machine is unable to tell which instructions are executed by the program, and which by the file store extension level.

As a development from the simple example above, this new decomposition can be seen in two ways; either as the separation of two levels in what was previously a rather sophisticated machine, or as the removal from the program of a set of basic utilities which were independent of the remainder of the program.

4.1.5.3 A simple Algol-type program.

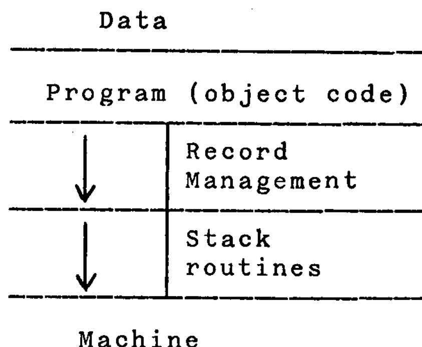
This example is of a similar complexity to the last, but it introduces the possibility of interpreter levels arising from degrees of sophistication of implementation of high level languages. I choose to illustrate this in the context of an Algol-type language, executing on a machine which has no intrinsic stack handling abilities.

A program of this type will be compiled into object code which includes calls on stack manipulation routines. These routines will be present in the object code, but the programmer did not include them explicitly in his design, and so I would like to isolate them. The configuration can be represented as follows:



4.1.5.4. An advanced Algol-type program.

Further complexity arises in the previous example if a program makes use of some more advanced facility of the language. For example, a system of records and pointers may require record allocation and accessing routines, heap management and garbage collection. This will be present in addition to the necessary stack manipulation routines, and indeed the record management will almost certainly rely on the stack routines. Hence the following configuration:



Of course, if we were not interested in distinguishing between the stack and record management levels then they could be merged into a single "run time support" extension level.

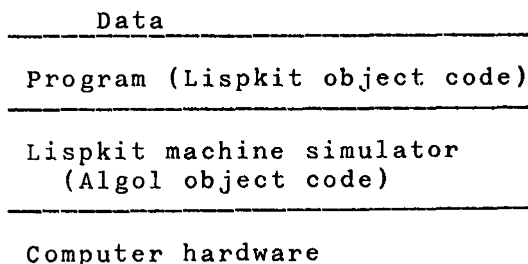
The problem arises, when extracting several supporting levels of interpreter, of the order in which the levels should be placed in the configuration. The order will be determined by the dependence between the levels. For example, if two levels A and B are extracted from a program then the order is determined by whether A uses the facilities of B, and vice versa. If A and B are independent (they only require the machine below in order to execute) then they may occur in either order.

If A uses B but B does not rely on A, then A is placed above B (I have treated the record and stack, above, in this way as A and B respectively). If A and B rely on each other then they must be merged to form a single level - maybe a different factorisation of the system is possible. With three or more levels more care may be needed, but I do not anticipate that this problem will be of sufficient significance to prevent useful progress.

4.1.5.5 A high level machine simulator.

Consider a program written in a very high level language, such as Lispkit, which has been translated into a data structure containing the object code for running the program on a special purpose machine simulator. The machine simulator is itself an Algol object code program for executing on a particular computer's hardware. The Lispkit program requires some data in order to execute.

An execution of such a system of programs will have the following structure:

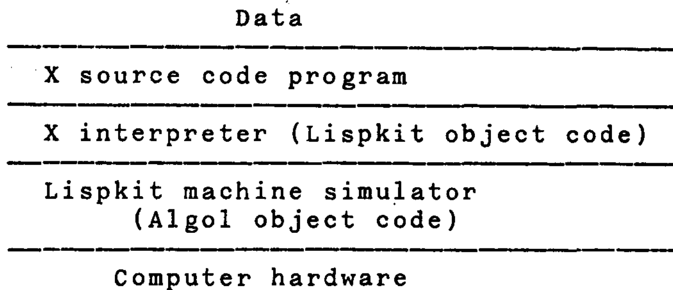


Again, each level of the configuration could be decomposed into further levels if required. The next, and final, example shows the interesting case where the data is split into another program plus data. This will be of significance later, as will the case

when the simulator has levels of mechanism distinguished within it.

4.1.5.6 Higher level interpretation of a program.

This final example is a development of the previous one. The program has become more specifically an interpreter for the source code programs of an experimental language X. Above the interpreter we have a source code X program and some data for it:



4.2 Two models for the behaviour of components within multi-level interpreter systems.

The notations described in the following pages will play a descriptive role in the formulation of concepts of software behaviour, and in the subsequent discussion of performance. The notations will not be developed into a complete formal theory. The system configuration diagrams introduced above are useful as reminders of the identities of system components. However, the diagrams say nothing formal about the way in which levels of machines, interpreters, programs and data interact.

I have two, apparently conflicting, demands of a formal description of the behaviour of multi-level interpreter systems. Firstly, the software components of a system are simply chunks of text, whether they are in a source or compiled form, and only acquire the

ability to be active and perform computations when "loaded" onto some active machine; the machine may itself be a virtual machine formed by loading the passive text of an interpreter onto some lower level machine. It should be possible to model this activation of passive program texts. Secondly, despite the previous observation, a program does seem to have behavioural properties of its own which are independent of the (virtual) machine on which it is executing; it has a certain "algorithmic complexity" which determines the amount of work required to complete the computation specified by the program's data. It should be possible to assign some active character to a program without including the characteristics of the (virtual) machine below, and it is this active character which a performance assessment must assess.

In the following sections I shall attempt to capture these two rather different aspects of system behaviour in two models, the Interpretation and Abstract Execution models respectively. The two models are orthogonal, but can be related to each other, and the requirement for consistency will constrain the models.

4.2.1 The Interpretation model.

In this model the components of a system are of two types; each software component is a passive Base text object, this covers the programs and data which form every level of a system except the lowest; and the lowest level of a system is an innately active object of type Machine which accepts the text of programs and data and generates results.

To represent the combinations of programs and data which a Machine expects to receive (and, later, interpreters will expect to receive), the more embracing type Text is necessary

Text = Base + Base x Text

An object of type Text is a sequence of Base objects, each of which is the passive text of some program or data. When a Text object is a sequence of more than one Base item I shall use square brackets to delimit the sequence, e.g. [p1, p2, d] , in which the last item will be of type Text although it will frequently be a Base object.

The results of a computation are conveniently grouped with programs and data as Base text:

program, data, results: Base

The type Machine can now be elaborated. A Machine is a function which accepts some software (in general a combination of programs and data), and produces some results:

Machine = Text $\longrightarrow$ Base

For example, the execution represented by the configuration

d : Base
p : Base
i : Base
mc: Machine

in which the names of the components have been annotated with their types, can be modelled by applying the Machine mc to a compound data structure:

results = mc ([i,p,d])

If mc:Machine then this gives a general purpose character to mc with which it may execute self-contained programs

results = mc(p) where p:Base

or more complex software systems, as above. This can be seen by substituting the definition of Text into that of Machine:

Machine = (Base + Base x Text) $\rightarrow$ Base

When a Machine is acting purely in the role of supporting a multi-level software system then its properties can be discussed in the restricted domain M':

M' = Base x Text $\rightarrow$ Base

or the "Curried" form of this, M":

M" = Base $\rightarrow$ (Text $\rightarrow$ Base)

This is the way in which hardware machines are normally used; the machine is given Base text as a program to execute, and some Text as data for the program. The separation of program and data shows that they are changeable components, the program can be executed with various sets of data. The two types M' and M" have been derived from a consideration of hardware machines, but they also describe precisely the usual concept of an interpreter as an active object which accepts program and data and produces results. I shall define the Curried domain as being exactly that of Interpreters:

Interpreter = Base $\rightarrow$ (Text $\rightarrow$ Base)

and I shall denote the member of Interpreter corresponding to the Machine mc, acting in the restricted role M', by mkInt(mc). mkInt converts a component of type Machine to one of type Interpreter, simply by modifying its type:

mkInt: Machine $\rightarrow$ Interpreter

and mkInt may be defined (in the notation of Chapter 2) as

$$mkInt(mc) = \lambda(base)(\lambda(text)mc(\lfloor base, text \rfloor))$$

Hence the following equality holds:

$$results = mc(\lceil p,d \rceil) = mkInt(mc)(p)(d)$$

where mc:Machine, p:Base, d:Text

Now, the range of the Interpreter mapping is (Text $\rightarrow$ Base), which has already been defined as Machine, and hence

Interpreter = Base $\rightarrow$ Machine.

Thus it is possible to generate a new active Machine, a <u>virtual machine</u>, by applying an active Interpreter to a passive program Base text. This corresponds to "loading" a program onto a machine, but not executing it until we provide some data; the loaded machine has become a new and different virtual machine. The active Machine obtained by loading a program text onto an Inpterpreter has the character which we usually attribute to the program itself - it accepts data and produces results. Hence the distinction has been made between the text of a program, which simply <u>describes</u> some computational ability, and an active program, which is a composition of the program's text and lower levels of hardware and software, and which <u>can perform</u> computation.

The analysis of a system can be continued to greater levels of detail, since if the Base text which is loaded onto an Interpreter also describes the function of an interpreter, then the resulting Machine is of the restricted type M' and can again be treated as an Interpreter. For example, consider the system

```
 d
---
 p2
---
 p1
---
 mc
```

The following analysis can be made of the results from such an execution:

```
results = mc([p1,p2,d] )
        = mkInt(mc)(p1)( [p2,d] )
        = p1'( [p2,d] )
        = mkInt(p1')(p2)(d)
        = p2'(d)
where  p1' = mkInt(mc)(p1)
       p2' = mkInt(p1')(p2)
```

The objects p1' and p2' are virtual machines which implement the functions described by the program texts p1 and p2 respectively:

```
 d
---      
 p2          }
---          }
 p1   } p1'  } p2'
---   }      }
 mc   }      }
```

Hence it has become clearer why I wish to treat complex programming systems as "multi-level interpreter systems".

4.2.2 The Abstract Execution model.

In the Interpretation model, described above, each software component of a system is passive, and only acquires an active characteristic when executed (or interpreted) by a Machine (or Interpreter). This seems to suggest that it is fruitless to enquire about the behaviour or performance of an isolated component, as the behaviour of the component is intricately

involved with the behaviour of the Machine or Interpreter on which it is executing. However, the notion that a program can be assigned behavioural attributes independent of its execution environment is in common usage; in some sense a program describes work to be performed, and it is the business of some underlying components how the work is performed. This notion will be of key importance in the treatment of performance in Chapter 5, and I shall attempt to suggest here a formal basis for that treatment.

The Abstract Execution model considers explicitly the interactions between each pair of components which are vertically adjacent in a configuration diagram; the interaction may be in the form of subroutine calls (or other operations) which the upper component demands of the lower, or in the form of accesses which the lower component makes into the data structure representing the upper component - these are simply convenient and suggestive intuitive interpretations of the events which occur during execution of the system. Each component is then modelled by a function which maps an interaction at the upper interface of the component into an interaction at the lower interface.

An alternative view is to consider that, at each interface, the component above the interface hands to the component below, some description of the complete computation; the description is at a level of abstraction which the lower component is prepared to expand in more detail and hand to the next lower component.

I shall restrict myself to consideration of interactions which are sequences of events; this is certainly adequate and clear for analysing systems executing on current conventional sequential processors, though I do not

know whether it is a necessary restriction in the following analysis, nor whether the theory will need modification to handle parallel processing systems.

Having attempted to bridge the intuitive gap between the Interpretation and Abstract Execution models, I shall now elaborate the latter.

Observation at an interface of a system for the duration of a computation will yield a record of the interaction, which is a sequence of events. The record is an <u>abstract execution</u> of type <u>Trace</u>:

Trace = sequence of interaction events

Each component of a system, with the exception of the highest level component (usually simple data), will be modelled by an active entity, an abstract execution machine of type AEMachine, which transforms a Trace at its upper interface into a Trace at its lower interface:

AEMachine = Trace $\rightarrow$ Trace

There are three types of system component which must be described in terms of AEMachines and Traces.

Firstly, the lowest level component, computer hardware, is innately active, as in the Interpretation model. A hardware component mc is a member of type AEMachine:

mc : AEMachine.

In this case the upper interface Trace will be a sequence of machine instructions and their operand values.
A hardware component, being at the lowest level, has no true lower interface, but the Trace produced by the component can be conveniently interpreted as the manifestation of the computation to an outside observer -

it will consist of such effects as power consumption, time, and input/output actions, and hence consists, in part, of the desired "results" of the computation.

Secondly, each software component, except the highest, is a Base text object representing a program which must be modelled as an AEMachine. For this purpose I shall postulate the availability of a function aep (for "abstractly execute program") which takes the text of a program and produces an appropriate AEMachine:

$$aep : Base \longrightarrow AEMachine.$$

aep is universal in the sense that it is capable of accepting any program, in any language and producing the AEMachine appropriate to the environment in which it will execute; I shall assume, for simplicity, that the program's text contains relevant clues about these matters for use by aep. This somewhat extravagant claim for the properties of aep will be pursued further below.

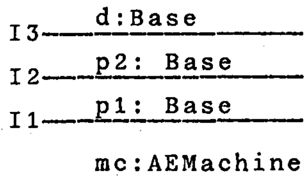
Thirdly, the highest level component, data, is a Base text object which is inspected by, and directs the actions of, the component below. A data component is not active in transforming Traces, but it interacts with the component below, and hence will be modelled by a Trace of this interaction. As for the case of programs, above, I shall postulate the availability of a function aed (for "abstractly execute data") which given a data text produces the Trace at its lower interface

$$aed: Base \longrightarrow Trace$$

Again I assume that the text contains clues about the nature of the data and the program which is to process it (the component below), and that aed is universal in a similar way to aep, above. In this case the Trace

could simply be the pattern of accesses to the data which the program below the interface makes, or even simpler, the data itself; on the other hand the data might be the text of a self-contained program, to be interpreted by the component below, and a useful trace of the interaction would be more complex.

Now that we can model each component of a system, the computation performed by an entire system can be modelled by composing the functions modelling each component. For example, the configuration of components



is modelled by

result trace = mc(aep(p1)(aep(p2)(aed(d))))

= mc(p1'(p2'(d')))

where p1', p2' and d' are the representations of p1, p2 and d:

p1' = aep(p1)

p2' = aep(p2)

d' = aed(d)

The result trace contains the desired results of the computation.

Alternatively the interaction at any interface can be inspected by abstractly executing the components above the interface. In the example above

trace at interface I2 = aep(p2)(aed(d))=p2'(d')

The Interpretation model allowed the construction of virtual machines by the combination of the lower level components of a system. The virtual machines then had the innately active Machine characteristic. Interpretation model virtual machines always occupy the lowest level

of a system. The Abstraction Execution model also enables the construction of virtual AEMachines by combining components, but not only at the lowest level of a system. For example, in the system above, the components may be combined in various ways:

```
result trace = mc(p1'(p2'(d')))
             = mcp1'(p2'(d'))
             = mc(p12'(d'))
             = mc(p1'(p2d')), and others

where mcp1' = λ (trace) mc(p1'(trace))
            = λ (trace) mc(aep(p1)(trace))
       p12' = λ (trace) p1'(p2'(trace))
            = λ (trace) aep(p1)(aep(p2)(trace))
       p2d' = p2'(d') = aep(p2)(aed(d))
```

and mcp1', p12': AEMachine, but p2d':Trace.
In this example mcp1' clearly corresponds to the virtual machines of the Interpretation model, as it combines the lowest levels of the system. However, p12' may be better referred to as a "virtual program", and p2d' is "virtual data" (not a particularly useful concept).

An important characteristic of the construction of virtual programs in the Abstract Execution model is that it permits adjacent levels of programs to be combined (and indicates how single levels may be factorised into two or more levels) without disturbing the structure and analysis of the remainder of the system. In the above example

```
mc(p1'(p2'(d'))) = mc(p12'(d'))
```

where the composition of two functions, p1' and p2', has been replaced by a single function, p12', which can be used as if p1' and p2' had never been separate.

This should be contrasted with the same transformation in the Interpretation model, where p1 and p2 can be factored out by exploiting the lambda expression notation:

```
        result = mc( [p1, p2, d] )
                     where, now, mc:Machine
                 = mkInt(mkInt(mc)(p1))(p2)(d)
                 = p12" (mc,d)
        where p12" = λ (mc,d)mkInt(mkInt(mc)(p1))(p2)
```

p12" now formally isolates p1 and p2, but it does not fit naturally into the Interpretation model scheme, as p12' does into the Abstract Execution scheme. p12" has an unusual type

        p12" : Machine×Base → Base

and cannot be used in place of p1 and p2 as if they had never been separate.

Hence the Abstract Execution model has improved on the Interpretation model by enabling abstract views of the structure of a system to be generated more naturally; but this has occurred at the expense of the introduction of the hazy notion of a Trace, and two powerful, but not formally defined, functions aep and aed. I shall attempt to remedy this unsatisfactory situation in the next section.

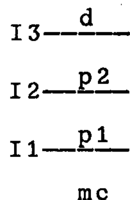
4.2.3 Relating the Interpretation and Abstract Execution Models.

The functions aep and aed are extremely powerful. They are marvellous tools to have available, but the validity of their postulated existence must be examined. This problem is intimately associated with the question "What exactly is a Trace?" I cannot answer that question satisfactorily in the

general case, but by relating the Interpretation and Abstract Execution models the functions aep and aed will be specified in terms of Traces, and only the latter problem will remain.

Ultimately I consider that the validity of the Abstract Execution model will be verified by the existence of useful experimental results which fit within the framework of the model, although, in common with most such models, that verification will not necessarily imply that the model is the best possible formalisation of the problem.

In the Interpretation model interfaces occur between Machines and their data. For example, in the system



the interface I1 separates the Machine mc and the data [p1,p2,d] , the interface I2 separates the Machine mkInt(mc)(p1) from the data [p2,d] , and I3 separates mkInt(mkInt(mc)(p1))(p2) from d.

At each of these interfaces it is the interaction of the Machine below and the data above which is recorded by a Trace. The practical act of observing an interaction can be achieved by taking the Machine and data, possibly modifying one or both to include monitoring mechanisms, and recording the events which occur. I shall use the function <u>tracing</u> to represent this experimental action formally:

tracing: Machine x Text $\longrightarrow$ Trace

and in the above example:

trace at I2 = tracing (mkInt(mc)(p1), [p2,d]).

In itself the introduction of tracing is fairly trivial, but it allows the gap between the Interpretation and Abstract Execution models to be bridged. The functions aep, aed, and tracing, in order to be useful, must give the same Traces at interfaces.

For example, in the configuration

```
I2——d——
I1——p——
     mc
```

the following equalities must be satisfied:

aep(p)(tracing(mkInt(mc)(p),d)) = tracing(mc, [p,d])
since aep(p) maps the trace at I2 into the trace at I1, and

trace at I1 = tracing (mc, [p,d])
trace at I2 = tracing (mkInt(mc)(p),d)

and
aed (d) = trace at I2 = tracing (mkInt(mc)(p),d)

These equalities amount to the definition of aep and aed by the empirical observation of Traces.

4.3 Comments.

The Interpretation model is more realistic and more immediately credible than the Abstract Execution model, which relies on fictitious Traces that never exist as palpable data structures within a system. In Chapter 5 I shall sidestep the issue of the general nature of Traces when, in the context of performance evaluation, experiments which monitor interactions yield only extracts of Traces and not the Traces themselves. The extracts will be credible statistics concerning the

execution of a system, and Traces will exist only in the background, in the formal description of a system. In the discussion of performance evaluation the Abstract Execution model will provide the structure of a performance analysis, and the correspondence with the Interpretation model will show how empirical assessments can be organised.

CHAPTER 5 - THE PERFORMANCE OF COMPONENTS OF MULTI-LEVEL SYSTEMS.

The performance of components of multi-level systems.

This chapter is concerned with the determination and analysis of the performance of the individual components of multi-level interpreter systems. The Abstract Execution model of Chapter 4 will be built upon to provide a framework for the study of performance.

Following the discussion in Chapter 4, programming systems are comprised of component levels of software and hardware. The component at the lowest level, which requires no support, is a hardware machine, and is generally of fixed characteristics in the sense that it cannot be modified in the short term to suit different applications. On the other hand, the software components of the system will be programs and interpreters which have been designed to cooperate in the solution of certain problems, and which can be changed comparatively easily to suit the circumstances.

Since the systems are intended to solve real-world problems, as judged to be important by the designers of the systems, those designers cannot ignore the engineering goal of finding the "best" systems to solve the given problems. This reasoning applies equally well in both the commercial and academic fields, as has been discussed in Chapter 1. The solution of a problem, by the execution of one or more software components on a hardware component, demands that resources be made available - time, memory, power and so on. The "best" system will be one which optimises (according to some criterion) the use of resources, the required balance depending on the particular application.

Hence the task is to be able to take programming systems, to examine relevant aspects of their performance, to make meaningful judgements of the adequacy of individual systems, to make meaningful comparisons of different systems, and, perhaps most important of all, to obtain feedback of useful conclusions to the designers of the systems and their components.

This task is more complex in the case of multi-level interpreter systems than for simple systems, as each component will have its own behavioural properties. A useful performance study of such a system should be able to yield conclusions and feedback on the individual components of the system - this is obviously especially valuable if the components have been designed independently.

For example, the system below has a typical multi-level interpreter structure:

Data
___
Program
___
Interpreter
___
Machine

The most common measurement to be made of the behaviour of such a system is the time taken to execute particular computations. The time for execution is itself a measure of the total amount of work performed by the <u>machine</u> during the computation, and hence reflects the contribution of all components of the system to the overall performance. Consider

executing the system above several times, with different data on each occasion:

Data 1	Data 2	...
Program	Program	
Interpreter	Interpreter	
Machine	Machine	

This is the experiment which is usually carried out in order to assess experimentally the performance of the program; the interpreter level is not usually explicitly noted, but it is almost invariably present, often in the form of a library of subroutines supporting the features of the language in which the program is written. If the performance parameter measured is the execution time then a relationship will be sought between this measure and some relevant characteristic of the data, the length of a list of items, for example. However, any observed relationship, a distinct curve on a graph or an approximate formula, will not necessarily yield the desired conclusions on the properties of the program, as the characteristics of the interpreter and machine have also been included in the measurements.

Obviously the ability to distinguish the properties of individual components from the properties of their environments is very important; if the conclusions of a performance study are to be returned to the designers of the components of a system, then it would not be reasonable or constructive to blame a program for inefficiency if the fault actually lies with a bad design decision in a lower level of interpretation.

The design of experiments for the comparison of two alternative versions of a component is an important problem. Clearly care is needed to ensure a fair comparison between the components, especially if they are executing in different environments, on different machines for example. The multi-level interpreter model suggests a straightforward definition of a fair comparison; the comparison must be between the performance properties of the components in isolation from the properties of their respective overall systems. Care must be taken in factoring out the influence of underlying levels of interpretation. For example, Turner (1979) has compared implementations of SASL and Lisp on different machines and has factored out the influence of the hardware by examining the requestsfor record storage cells that the implementations make on their storage management support routines; also Moss (1980) has compared several Prolog implementations, on several different machines, by initially performing simple experiments to "normalise" the machine execution speeds. Undoubtedly there are many other instances of these types of investigations.

Performance assessments may be carried out either experimentally or analytically. The latter may be quite appropriate for relatively straightforward algorithms, and for the restricted use of more complex algorithms, but in general such complex algorithms will not be susceptible to a complete analytical treatment. The complex algorithms in which I shall be interested will be interpreters for high level languages, and performance assessments will have to be obtained experimentally.

In the subsequent sections of this chapter I shall give a definition of the performance of a component in a multi-level system, and also the outline of a methodology for the assessment of performance which arises from the definition when applied to real systems. The definition of performance is based on the Traces of the Abstract Execution model of Chapter 4, and thereby provides a solution to the problem of isolating the properties of components from the remainder of the systems in which they execute. The treatment of performance will concern itself with the relationship of the performance of components to each other, and to the system as a whole, and will not concern itself with the internal details and machanisms of individual components.

5.1 Defining the performance of a component.

The Abstract Execution model of multi-level interpreter systems treats each component of a system as a mapping between interactions at the upper and lower interfaces of the component; the interaction, or Trace, at the upper interface is a complete description of the computation to be performed, and the component transforms this into a (presumably) more detailed Trace describing the computation at the lower interface. The Trace at an interface defines the work to be performed by the lower levels of the system.

This suggests that the crucial characteristic of a component is the way in which it transforms the workload between its upper and lower interfaces. In the discussion which follows, this observation

will form the basis for a definition of performance. Before proceeding to the definition, however, it is necessary to introduce a more practical means of observing an interaction than the tracing function introduced in Chapter 4.

5.1.1 Observing interactions at interfaces.

In Chapter 4 there was some difficulty with the precise nature of the objects of type Trace, although they were a valuable abstraction in enabling the Abstract Execution model to construct representations of multi-level systems which had useful properties. The operation tracing was introduced to link the Abstract Execution Traces to observations of real systems described by the Interpretation model. tracing was defined as

being of type Machine x Text ⟶ Trace, and in the system

$$\begin{array}{l} I2 \ \frac{d}{\quad} \\ I1 \ \frac{p}{mc} \end{array}$$

the trace at, for example, I2 can be obtained from the Abstract Execution model

trace at I2 = aed (d)

or by tracing the Interpretation model

trace at I2 = tracing (mkInt(mc)(p),d)

However, tracing is not a representation of the practical act of observing an interaction; when the interaction at an interface of a system is observed, usually by modifying the software, the information collected is never a complete Trace, but some <u>extract</u> of the Trace which represents some particularly interesting aspect of the interaction.

Similarly, when discussing the Traces at upper and lower interfaces and how they are related by the component between, it will be the relationships between particular aspects of the interactions which convey most understanding.

Hence I shall define the type <u>Extract</u> to cover the interesting statistics which can be distilled from Traces, and the deliberate act of choosing an interesting statistic to extract from a Trace will be represented by the functions <u>Select</u>:

$$\text{Select} = \text{Trace} \rightarrow \text{Extract}.$$

Note that Extracts of Traces are now more credible objects; for example, they include such useful statistics as the number of machine instructions executed at the upper interface of a hardware component, the number of times that particular instructions or particular sequences of instructions are executed, the number of statements or expressions interpreted at the lower interface of a program, and the number of times that certain elements of data are accessed by the program below the lower interface of the data. In short, Select functions allow us to consider any aspect of an interaction that we consider to be relevant, although, of course, some statistics may be easier to obtain than others.

The practical act of observing an interface, in order to obtain an Extract concerning a particular aspect of the interaction, is modelled by the function <u>monitor</u>:

monitor: Select x Machine x Text ⟶ Extract

which may be defined in terms of tracing:

monitor (sel,mc,d) = sel(tracing(mc,d))

For example, in the system

I2 —d—

I1 —p—

mc

if selaccess: Select can extract a record of the accesses which p makes into d at I2 then

record of accesses = monitor(selaccess, mkInt(mc)(p),d)

The definition of monitor shows that the complete action of modifying a system, observing an interaction, and extracting useful information is modelled by the composition of the functions sel and tracing; the composition represents the practical experimental measurement of a system.

Of course Select functions can be applied equally well to the Traces of an abstractly executed system. In the example above, the record of accesses can be obtained trivially

record of accesses = selaccess(aed(d)).

5.1.2 A definition of performance.

The Abstract Execution model represents a component by a mapping between Traces, but Traces are more of a formal than a practical tool; instead it is usual to consider a program as determining a mapping between events of interest at its upper and lower

interfaces, in other words between Extracts at the interfaces. For example, "a program makes x calls of a particular subroutine at its lower interface when the data at its upper interface contains y items".

It is in these terms which the performances of programs, interpreters and machines are usually expressed, and so I shall define a Performance as a mapping between Extracts:

$$\text{Performance} = \text{Extract} \rightarrow \text{Extract}$$

Any component may be characterized by a number of Performance functions, each relating different aspects of the interactions at the upper and lower interfaces. It is the aim of the study of the performance of a component to characterise, as completely as possible, the mapping which the component implements between selected events of interest; the study may be carried out analytically, thus arriving at a precise formula for the Performance, or experimentally, yielding particular points in the Performance mapping and presented as maybe a table, graph, or approximate formula.

To assess the Performance of a component with both upper and lower interfaces it is necessary to know the Base text of the component, and the Select functions for monitoring the upper and lower interface interactions:

$$\text{assess} : \text{Base} \times \text{Select} \times \text{Select} \rightarrow \text{Performance}$$

The assess operation is not constrained to being either analytical or experimental, but in either case the resultant Performance mapping must be consistent in both the Interpretation and Abstract Execution models. Consider assessing the performance of p in the simple system

```
I2——d——
I1——p'——
    mc
```

and let sel1 and sel2 be interesting Select functions for the interactions at I1 and I2 respectively. The assessment of the performance of p, perfp, is

```
    perfp = assess (p, sel2, sel1)
so that perfp(extract2 ) = extract1.
```

By the Abstract Execution model the performance perfp relates the selected features of Traces at the two interfaces:

```
    assess (p,sel2,sel1)(sel2(aed(d))) = sel1(aep(p)(aed(d)))
    since extract1 = sel1(aep(p)(aed(d)))
                   = sel1(trace1)
          extract2 = sel2 (aed(d))
                   = sel2 (trace2)
          trace1   = aep(p)(aed(d))
          trace2   = aed(d).
```

Alternatively, the Interpretation model can be used to show monitoring actions explicitly:

```
        assess(p,sel2,sel1)(monitor(sel2,mkInt(mc)(p),d))
                 = monitor (sel1,mc, [p,d] )
```

which can also be simplified to

```
    perfp(extract2) = extract1
```

```
since monitor(sel1,mc, [p,d] ) = sel1(tracing(mc, [p,d] ))
                               = sel1(trace1)
                               = extract1
      monitor(sel2,mkInt(mc)(p),d) = sel2(trace2)
                                   = extract2
```

The required consistency of Traces between the Interpretation and Abstract Execution models, and the applicability of Select functions to both models, has ensured that the properties of Performance mappings are the same in both models. The notion of Performance is clearly most closely related to the Abstract Execution model, as it was defined in those terms, but its direct compatibility with the Interpretation model and monitor is important, as the latter model indicates how assessments of performance can be made empirically. The equations above show that perfp, a particular Performance function characterising p, relates Extracts:

perfp(monitored Extract at upper interface)
= monitored Extract at lower interface.

This relationship shows that we can predict lower interface Extracts if perfp is already known, or that we can monitor the interfaces in particular experimental executions of the system and thus build up an impression of the perfp mapping from particular data points. In the latter case the system will be viewed initially in terms of the Abstract Execution model - components of interest will be isolated, and appropriate interfaces and Select functions will be chosen; then experimental measurements will be

made by monitoring the system as described by the Interpretation model; and finally the measurements will be presented as a Performance assessment, in tabular, graphical or algebraic form, and the precise significance of the assessment with respect to the system will be clear from the choice of components, interfaces, and Select functions.

5.1.3 Combining the performances of adjacent components.

One important consequence of the definition of Performances, above, is that the Performances of two components occupying (vertically) adjacent levels of a system can be combined in a very simple way to give the overall Performance of the pair as if they were a single component. This is related to the ease of construction of "virtual programs" in the Abstract Execution model.

Consider the following system configuration

```
I3—d
I2—p2
I1—p1
   mc
```

Suppose that sel1, sel2 and sel3 are Select functions extracting statistics of interest from the Traces at interfaces I1,I2,and I3 respectively. Hence

```
trace3 = aed(d)
trace2 = aep(p2)(aed(d))
trace1 = aep(p1)(aep(p2)(aed(d)))

sel3 (trace3) = extract3
sel2 (trace2) = extract2
sel1(trace1) = extract1
```

If the Performances of p1 and p2 are perf1 and perf2 respectively,

```
    perf1 = assess (p1,sel2,sel1)
    perf2 = assess (p2,sel3,sel2)
then
    perf1(extract2) = extract1
    perf2(extract3) = extract 2
```

and since for both perf1 and perf2 the same Select function, sel2, has been used at I2 to yield extract2 then

```
    perf1(perf2(extract3)) = extract1
or  perf12(extract3) = extract1
    where perf12 = λ(extract) perf1(perf2(extract)).
```

The Performance function perf12 is thus the mapping between the selected statistics at I3 and I1, that is the Performance of the virtual program formed by considering p1 and p2 as a single component, and is simply the composition of the Performances of the individual components.

The importance of this result is that it assures us that, when analysing a system, several components can be combined if detail is to be ignored, and single components can be separated into several levels if more detail is required, without disturbing the properties, structure and analysis of the remainder of the system.

5.1.4 Comments.

I have attempted to provide a framework for the analysis of the performance properties of components within multi-level interpreter systems. Traces have

been relegated in importance with the introduction of Extracts which represent Selected statistics that can be obtained experimentally.

However, within the framework there still remains much flexibility in the practical assessment of systems; components must be isolated, interfaces and statistics for collection selected, experimental measurements made, and any choices possibly modified due to unforeseen complexity or experimental problems.

5.2 A methodology for empirical performance assessment.

Many choices have to be made, and compromises reached, during the empirical assessment of a system within the framework outlined in the previous section. During the practical research reported in the following chapters of this thesis, I found that the 8 point programme given below was valuable in organising the work, and in highlighting the decisions to be made, and problems to be overcome, in a manageable order.

1. The first step is to divide the system to be assessed into a multi-level interpreter structure which reflects the particular interests of the performance assessment. The components to be assessed should be isolated, as accurately as possible, from their data, and any lower levels of supporting routines or interpreters whose characteristics are not to be included in the assessment. Interfaces should delimit precisely the components to be assessed.

2. A general qualitative analysis should be made of the behaviour and interactions occuring at the interfaces delimiting components of interest. This will guide the choice of Select functions in the next step.

3. The aspects of the interactions which are to be monitored at each interface must be chosen. This choice is largely determined by the particular performance characteristics which are desired, but also partially by practical considerations of what it is possible and not possible to monitor (see step 4). The choice made here, which includes a decision on which monitored statistics are to be compared with which, has a large effect on the meaning of the performance relations obtained.

The main problem is that, at typical interfaces, the choice of interaction events to monitor is very large, and the series of interaction events can be viewed at different levels of abstraction. For example, suppose that above an interface the data is a binary tree (e.g. an s-expression), and below the interface is some program which scans the tree repeatedly to ascertain whether some property is satisfied (e.g. whether any two subtrees are identical). A view of the interaction which is very "close" to the data would record only one event - that the data exists and is passed to the program. A view that is very close to the program could record each and every individual occasion on which the program accesses a node or leaf of the tree (including repeated inspections). An intermediate view would note some property of the tree, such as the total number of internal nodes and leaves, or the depth of the tree. Conventional wisdom will usually dictate that a view in the latter group is the most appropriate, but there is no reason why the other views should not be selected for some applications.

A similar situation exists when the interaction at the interface is an interpreter executing a program. A view of the interaction close to the program could count the statements to be executed, and expressions to be evaluated, according to some adequate and economical flow of control through the program. On the other hand, a view of the interaction close to the interpreter could count the number of basic evaluation steps performed by the interpreter, which would possibly include the repeated evaluation of expressions (e.g. if an interpreter of Lispkit programs re-evaluated a defining expression at each access to the defined variable, a mechanism which could be used to implement call-by-name semantics). These two views would not necessarily coincide closely.

The previous situation is complicated further if the program above the interface has been compiled, converted to some intermediate textual form, before interpretation; this is a very common situation. The performance properties may have been changed by the compilation. The choice must be made whether to monitor the interpreter below the interface, the intermediate code above the interface (both of which are present in the executing system), or the statements and expressions of the original "source text" program (which may have to be monitored indirectly, possibly by separate simulation). The three choices may not be closely related, and a decision must be made which is appropriate to the circumstances. In fact, for this type of system it may be interesting to monitor both the original program and the evaluation steps performed by the interpreter; the results may be compared to obtain a

Performance relation as if there were an extra software component between the program and interpreter, and this relation could yield an assessment of the overall evaluation scheme embodied in the compilation step and interpreter design.

In the latter two cases I referred, somewhat vaguely, to monitoring the source text statements and expressions of a program, and described this as "according to some adequate and economical flow of control through the program". I wished to capture the impression that an intelligent programmer would have of the amount of work necessarily involved in executing his program. This is obviously a subjective issue, but for most languages and constructs there is general agreement; for example, in Lispkit we might expect a call-by-value semantics and correspondingly expect each defining expression and actual argument expression to be evaluated once only. I shall call such a view of a program's execution a <u>model interpretation</u>. The important point about the use of model interpretations for gathering statistics in performance assessment, is that the resultant assessment will be with respect to the programmer's expectation, which will be consistent between different programs in the same language.

4. Make a more detailed, but still qualitative, investigation of the interactions at the interfaces. The purpose here is to decide precisely how the selected statistics are to be monitored at each interface; this will usually entail deciding how the various software components are to be modified to collect the required statistics. In order to

obtain model interpretation statistics it may be necessary to decide on manual analysis of the program, or to construct a special purpose simulator or interpreter, or to modify the program itself to collect its own statistics; some combination of techniques may be necessary as manual analysis may become intractable, and simulation or self-monitoring may be unduly inefficient (especially in purely applicative languages such as Lispkit or Prolog). In fact, the technique of modifying a program to monitor its own activity could be a good method for defining model interpretations rather more formally, as the statistics gathered would depend only on the semantics of the language and not on the details of execution, but that is a sideline I have not pursued.

It is at this step, and possibly step 5, that re-iteration through earlier steps may become necessary as difficulties of monitoring arise. Some compromise may be necessary between desired performance assessments and practical statistics collection.

5. Make appropriate changes to the system to incorporate monitoring mechanisms, and implement any special simulators if necessary. Note that several versions of a system may be required, each monitoring different aspects of the execution, if the monitoring mechanisms interfere with each other; for example, the execution of monitoring mechanisms in an upper level component will be present as a part of lower level interactions, and monitoring this lower level interaction will provide a false impression of the software above.

6. Choose a set of test executions of the system, perform the executions, and collect all the required statistics. The tests should be chosen such that, for each component whose performance is to be assessed, the range of the component's abilities is fully exercised; this is the requirement that the discrete set of data points obtained should be as accurate a representation of the desired Performance function as possible. This is a very big problem, and it does not seem likely that a truly satisfactory solution will be possible except in a very few cases.

When a component to be assessed is a "data processing" program (e.g. sorting, searching), there may be an obvious range of data values to present to the program (e.g. lists of successively greater length, or trees of successively greater depth), but if the component is an interpreter then it will almost certainly be impossible to characterise a "range" of programs. In the latter case it might be necessary to assess the Performance of the interpreter separately for different types of program and to attempt to infer some general properties of the interpreter from the results.

7. Make comparisons of statistics collected in step 6 in order to obtain the Performance relations chosen in step 3. The Performance relations can be expressed as tables of statistics, as graphs of the discrete points obtained in the test executions, or, if there is sufficient justification, as graphs of the discrete experimental points with other points added by interpolating or extrapolating along curves which fit the experimental points. In each of the cases it seems reasonable to present the relation as a formula if the observed points show an easily defined trend; this formula will be the fitted curve in the case of an

interpolated graph. Of course, it should be remembered that such fitted formulae are only hypotheses about trends.

This step may lead to reiteration through step 6 if further statistics are required to confirm or explore trends.

8. Finally, conclusions can be drawn from the empirical performance assessments, the quality of components can be judged against external criteria, and comparisons of components can be made. Again, reiterations through earlier steps may be inspired by the concluding observations.

5.3 Comments.

I have shown an approach to performance analysis, which exploits the properties of systems described by the Abstract Execution model in order to define and isolate the properties of individual components, and the Interpretation model guides the collection of experimental performance data. Practical assessment entails many complications which must be overcome by careful choices; these have been outlined in an 8 step scheme for organising the practical work.

In the following chapters the performance assessment discipline will be illustrated by application to pseudo-machine implementations of Lispkit and Prolog, and to interpreters for Lispkit and Prolog. At the same time the assessments will provide interesting results and comparisons of the pseudo-machines and interpreters.

CHAPTER 6 - LISPKIT AND PROLOG MACHINES: STRUCTURE AND PERFORMANCE.

Lispkit and Prolog machines: Structure and performance.

It is common practice to implement very high level languages, which provide for the handling of symbolic data structures, by designing a special purpose pseudo-machine. Each language will have its own style of pseudo-machine, and programs are usually compiled into the form of an intermediate machine code which is then interpreted by the pseudo-machine.

Pseudo-machines, in this context, are largish programs written in some well known, well supported, and usually reasonably efficient, conventional language (typically one of the widely varying Algol family). Each pseudo-machine, in executing an intermediate machine code program, simulates (at an abstract level) the activity of some hypothetical computer hardware which is intended to be particularly well adapted to the requirements of the very high level language in question.

I am on reasonably safe ground to assume that the only style of computer architecture, which is currently well enough understood to provide a sound basis for general purpose computing machines, is the traditional von Neumann single sequential instruction stream architecture. In the von Neumann paradigm the state of the machine is held in a memory, and each instruction in the sequential stream causes a state transition in the memory. There is, nevertheless, flexibility available in the choice of instructions (and hence the transitions), in execution of the instructions (possibly pipelined, for example), and in the

organisation of the memory (linearly addressed, paged, tree structured, for example). Thus it seems quite reasonable that the von Neumann style has been followed in the design of very high level language pseudo-machines such as Henderson's Lispkit (Henderson (1980)), Turner's SASL (Turner (1979)) and Warren's Prolog (Warren (1977)) (though Warren's Prolog machine does not exist as an independent entity, as its intermediate machine code serves only to structure a compilation of his dialect of Prolog into DEC10 machine code). It is particularly straightforward to construct the software for pseudo-machines with such architectures. My implementation of Prolog also follows the von Neumann paradigm, though it was designed independently of Warren's pseudo-machine.

Very high level languages oriented towards symbolic data structure processing are well suited by one particular memory organisation. This comprises a heap store of cells which may be linked by pointers to form lists, trees and so on. The use of such a heap store enables machine states (or parts thereof) to be saved by recording simply a small collection of pointers, and avoids the copying of large data structures when passing parameters or building new structures from old ones. There are many strategies for organising and managing heap stores, but I will not discuss them here.

The pseudo-machine style of implementation is particularly valuable for very high level languages for several reasons. By dividing the processing of a program into distinct compilation and pseudo-machine execution phases the complexity of the design exercise

is reduced. Extension of the facilities in various ways is made more straightforward. The two phase design, if carried out well, may aid in understanding the meaning and use of the language through one specific, clear implementation.

The logical design characteristics of two particular pseudo-machines, for Lispkit and Prolog, are given in Appendices B and D, with example concrete realisations in AlgolW. A performance assessment of these realisations is made in the later parts of this chapter. For the performance assessment to be made here, it is the structure of the software realisations which is important.

The powerful nature of the basic programming facilities provided by very high level languages means that the pseudo-machines will have to perform sophisticated actions in response to single intermediate machine code instructions. Examples of this include the allocation of new heap cells (possibly invoking garbage collection or other management actions), and the unification of su-expressions in Prolog. Since it is good programming practice to isolate the sophisticated facilities and to design them separately, it would seem to be a good idea to treat the implementations as multi-level interpreter systems for the purposes of performance analysis. The systems will comprise an upper level, which is responsible for shaping the interpretation of the intermediate machine code program, and one or more lower levels, which provide the supporting facilities.

The performance of each level is of interest independently, not only in the cycle of design and improvement of the pseudo-machines, but also in obtaining an indication of the performance that could be expected if the systems were executed on specially constructed hardware. For example, the heap store would be a candidate for realisation in hardware rather than software, with each cell access or allocation request forming one machine instruction, but with all domestic chores (such as garbage collection) carried out in parallel with normal execution. In this case the activity generated by the upper levels of the system alone would determine the time for execution.

Hence the framework of thought for performance assessment, which has been examined at great length in previous chapters, will be of relevance here, as well as in the next chapter (where the performance of higher level interpreters, executing on top of the pseudo-machines of this chapter, will be the topic).

6.1 A Lispkit pseudo-machine.

The operational model for the execution of Lispkit programs, given in Chapter 2, can be realised reasonably directly by a special purpose Lispkit machine (LM). The Lispkit programs are precompiled to an intermediate machine code, which consists essentially of linear sequences of instructions for the LM. The LM has a conventional sequential machine architecture, but in which all data, programs and results are held in four registers, S,E,C and D, each of which contains an s-expression. Appendix B contains a detailed

description of the roles of the four registers, the machine actions determined by each LM instruction, and the code generated during compilation of each type of Lispkit expression.

In practice Lispkit programs are represented in an s-expression syntax (also in Appendix B), and the compiler itself is a Lispkit program which executes on the LM. The compiler accepts a Lispkit program as an s-expression, and produces object code, which is also an s-expression, suitable for re-input to the LM as a compiled program.

In this section I would like to describe the structure and general properties of a particular software implmentation of the LM. The implementation is interesting for its simplicity and economy, and its performance properties will be explored in a subsequent section of this chapter.

6.1.1 The software components.

The LM is implemented as a medium sized (several hundred lines) AlgolW program, which is compiled to execute on an IBM 370/168 under the supervision of the Michigan Terminal System (MTS).

Broadly speaking the LM software can be divided into four component parts: s-expression storage management, s-expression input, s-expression output, and the central Lispkit evaluator (usually known as the "apply" routine).

The s-expression storage management software is an essential facility, relied upon by all other parts of the LM. A large amount of storage for s-expressions is provided in a collection of arrays. Without going into too much detail, each constructed node in an s-expression is allocated to one cell of storage which contains pointers to the left and right subtrees, and each atom is allocated one cell which contains the number or symbol. The cells are managed as a heap, and each cell contains administra tive information to help with this. During program execution, storage cells are explicitly allocated but implicitly released, and hence the storage management incorporates a simple mark and scan garbage collector to reclaim released cells. The garbage collector reclaims all cells which are not currently part of the s-expressions in S,E,C,D and a temporary working register W. The storage management is a self-contained component which provides a service to the remainder of the LM. Each access to one of the five registers, each access to a cell, and each request for the allocation of a storage cell is considered to be an operation provided by the service. Garbage collections are invoked by the storage management itself in response to an allocation request when no more cells are noted as available. An explicit storage initialisation procedure is also provided.

S-expression input and output does not deserve detailed exploration. Strings of characters representing the written form of s-expressions are transformed to and from s-expression storage respectively.

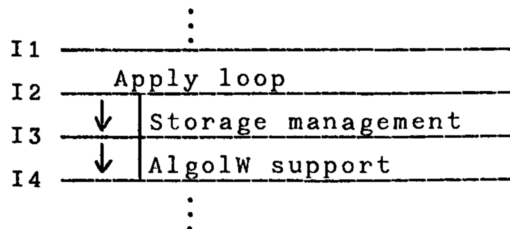
The apply routine is very straightforward. It contains statements to initialise the machine state in S,E,C and D, with the Lispkit program and its arguments, and then enters an iterative loop in which each execution of the loop body decodes the next LM instruction and performs the appropriate state transition. Iteration is terminated when the STOP instruction is encountered.

The overall pattern of activity in the LM follows a simple sequence: s-expression storage is initialised, the program and argument s-expressions are input, the program is applied to its arguments, and finally the resultant s-expression is output.

The services of the storage management are called upon throughout each stage of execution of the LM.

Performance analysis will be primarily concerned with the properties of the apply phase of LM execution, and with the storage management during this phase.

Bearing in mind the sophisticated stack manipulation, parameter passing and array access mechanisms which cannot be avoided when compiling and executing an AlgolW program, I initially propose to treat the LM as a multi-level interpreter with the following structure:



Above interface I1 there will be a compiled Lispkit program, and its data. Below interface I4 will be a (virtual) machine capable of executing IBM 370 machine code instructions, as all actions required by the components above I4 will be presented at I4 in terms of these instructions.

The components between I1 and I3 are derived directly by compiling the LM source code in AlgolW.

The AlgolW support component contains software implementing the stack, parameter and array access mechanisms which are general facilities provided by the AlgolW language.

6.1.2 General behavioural considerations.

Referring to the multi-level interpreter structure above, the nature of the interactions at each of the interfaces can be identified.

At interface I1 the LM machine instructions of a compiled Lispkit program are scanned and executed in sequence by the LM apply loop.

At interface I2 the actions of the apply loop are presented as a sequence of IBM 370 machine instructions (which are passed directly to I3), AlgolW support operations (also passed directly to I3), and storage management operations.

At interface I3 the storage management actions are executed as sequences of IBM 370 instructions and AlgolW support operations. These sequences are mixed with the instructions and operations passed directly from I2. The IBM 370 instructions are passed directly to I4.

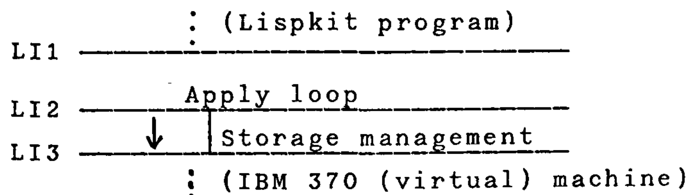
At I4 the IBM 370 instructions which realise the AlgolW operations of I3 are mixed with the instructions passed directly from I3. Hence below I4 a (virtual) machine which can execute programs in the form of IBM 370 machine instructions is required.

A simplification to the three level model will be convenient for the purposes of performance analysis, as it is impractical to gain access to the software contained in the AlgolW support level, and so it is not possible to monitor directly the activity at I3 as processed by the support software, or to monitor the internal behaviour of the level. However, it is unlikely that the basic facilities provided by the AlgolW support have been implemented without a little consideration for the users of AlgolW, and therefore it seems reasonable to assume that unsophisticated use of AlgolW will not incur excessive overheads in the supporting software. I shall assume that each simple operation performed by the supporting software is achieved by a short fixed length sequence of IBM 370 machine instructions, and hence that the performance of the supporting software is simply a linear factor. During the apply loop and in the storage management I have avoided the use of recursive procedures (substituting loops and explicit stacks where necessary), and

parameter passing is all achieved by value and by result. By these simplifications I hope to remain within the assumed behaviour of the AlgolW support. (In practice the assumption is not violated obviously, though I have no direct evidence to prove that this is so. Note that I have avoided the use of the AlgolW records and references facility, which would certainly incur large overheads in the support as it would have to manage its own heap storage).

I have discussed the behavioural contribution of the AlgolW support separately, and in some detail, in order to make its properties explicit, to show the practical difficulties which its inaccessibility creates, and to suggest how an attempt can be made to simplify these difficulties by careful planning.

As a consequence of the assumption of linear performance for the AlgolW support, empirical analysis of the LM will be based on a slightly simpler multi-level interpreter structure in which the support software has been absorbed into the apply loop and storage management components:



It will be borne in mind that the IBM 370 machine instructions at interface LI2 already contain a linear factor of overhead, and that the machine instruction contribution at LI3, from the storage management, also contains a linear factor.

Throughout the remainder of this description and analysis of the LM I shall use the simple terms LI1, LI2, LI3, Apply and Storage to refer to the interface and levels as shown in the simplified multi-level structure above.

6.1.3 Performance assessments to be made.

There are four interesting comparisons between abstract executions which can be made in order to assess the quality of the LM design and implementation. With reference to the simplified LM structure above:

(i) Comparing a model interpretation of the Lispkit program, executing above LI1, with the trace of LM instructions at LI1 scanned and interpreted by Apply, will show how effectively the evaluation scheme (embodied in the compilation step and Apply software) implements the Lispkit language. This comparison is not an assessment of a software component of the system, but an identical technique can be applied as if an extra software component were present.

(ii) Comparing a model interpretation of the Lispkit program with the sequence of IBM 370 instructions and Storage operations which Apply presents at LI2 will show how effectively the detailed code of Apply implements Lispkit language constructs.

(iii) Comparing the trace of LM instructions scanned and interpreted by Apply at LI1 with the sequence of instructions and operations which Apply presents at LI2 will show how effectively the detailed code of Apply implements the evaluation strategy which it imposes on the Lispkit program, that is how well Apply achieves its own goals.

(iv) Comparing the Storage operations which Apply executes at LI2 with the trace of IBM 370 instructions presented by Storage at LI3 will show how effectively the <u>detailed code</u> of the Storage software implements its own <u>higher level operations</u> (as made explicitly available to Apply).

Of course, comparisons (i), (ii) and (iii) will be related to each other by the property of cascading performances (discussed in Chapter 5), and also the <u>overall</u> performance of the LM could be found by direct comparison of the Lispkit program execution, above LI1, with the IBM 370 instructions at LI3.

6.1.4 Monitoring the behaviour.

The performance assessments planned in the previous section require five bodies of statistics to be collected for comparison. By what techniques should these statistics be collected?

Considering the interfaces individually:

LI1:
At LI1 the behaviour of the Lispkit program itself must be monitored (call this statistic <u>LS1</u>), and the LM instructions as executed by Apply must be monitored (call this statistic <u>LS2</u>).

For LS1 a convenient statistic to collect is the number of function applications performed by a model interpretation of the program. The count includes <u>where</u> and <u>whererec</u> expressions, which are related to function applications (both semantically and practically).

For some programs the count is obtained by manual analysis, and for some by means of a specially constructed simulator (this is inefficient, so it is not practical for large programs). In later experiments the count is obtained from the LS2 statistics (below) by noting, in earlier experiments, that the number of function applications (plus wheres and whererecs) is exactly mirrored by the number of AP and RAP instructions executed by the LM. Function applications are a convenient way of estimating the overally workload represented by a program. The true workload per function application will vary between programs, as the number of primitive expressions per function application varies. However, it is usual to find that large expressions contain embedded function calls, and so the properties observed will be approximately representative of "average" programs.

There are, of course, many other statistics which could be collected (too many!). One of the more interesting would be to count the operations which request explicitly the allocation of new s-expressions cells, that is the number of cons,+,-,*,div, and rem operations. Most programs can only make useful progress by employing these operations. I would expect them to be well scattered throughout a program, and hence they will be closely related to the function applications.

For LS2 the measurement is much easier. The number of times that each LM instructions is executed is recorded by simple modifications to Apply. This also gives the total number of instructions that are executed, but no information on the ordering of the instructions is retained.

With only a little more ingenuity more complex events, such as particular sequences of instructions, could be monitored.

Whilst discussing the behaviour at LI1 I shall note that there is little interest in a model interpretation of the compiled code (LM instructions) of a Lispkit program, as the execution of the code by Apply corresponds directly to the model interpretation. This fact could be confirmed by experimentation, but the result is not a key one in the assessment of the LM, and it can be seen by inspection of the LM machine code and the Apply software.

LI2:
At LI2 it is necessary to monitor the IBM 370 instructions and Storage operations executed by Apply (call this statistic LS3), and the operations as received for execution by Storage (call this statistic LS4).

There are two complications with these measurements. Firstly, Apply cannot monitor its own execution of IBM 370 instructions directly since it is written in AlgolW source code. Attempting to count the instructions by using the MTS timing facility would be unreliable, as the contribution from execution of Storage operations would have to be measured and subtracted, and these are sufficiently frequent that the unaccountable effects of the timing operations themselves would be significant. Secondly, the Storage operations strictly include all allocation operations (new cons cell, new number cell, new symbol cell),and all accessing operations (car,cdr,iscons, isnumber,

issymbol, extracting the value from atoms and references to the registers S,E,C,D and W). But most of the accessing operations are in the form of in-line code in Apply rather than procedure calls, and this means that monitoring LS4 from within Storage is not possible.

The first of these problems may be overcome by making the reasonable assumption that AlgolW compilation generates a simple linear sequence of IBM 370 instructions for each non-looping section of AlgolW source code, and that the lengths of the two sections of code are roughly proportional. Hence Apply can measure the number of IBM 370 instructions which it executes (to within a constant of proportionality) by counting the number of times that critical sections of the software are themselves executed (for example, repetitive loop bodies). Since the Storage operations which are executed by Apply are scattered throughout the Apply software, the "loop counting" will automatically also be proportional to the number of Storage operations executed. Apply contains three loops, the main instruction execution loop (already monitored for LS2), and two small loops for looking up the value of variables in the environment E. Each of these loops is counted to give the LS3 statistic.

The second problem is overcome by noting that the pattern of Storage operations requested by Apply is identical to the pattern processed by Storage (since Apply has explicit control via procedure calls). Apply is modified to count each of the Storage operations which it requests, in-line code as well as procedure calls, though this requires care. The LS4 statistic is obtained by this method.

Note that although the LS4 data could be used to form the Storage operation count of LS3, it would be unusual to add a loop count to a Storage operation count, and it would contribute no more useful information to LS3 than the loop count alone. However, since it is reasonably straightforward to obtain the operation count, I have chosen to use it as the precise measure of LS4.

LI3:

The execution of IBM 370 instructions by the Storage software is the only statistic of interest at this interface (call the statistic LS5).

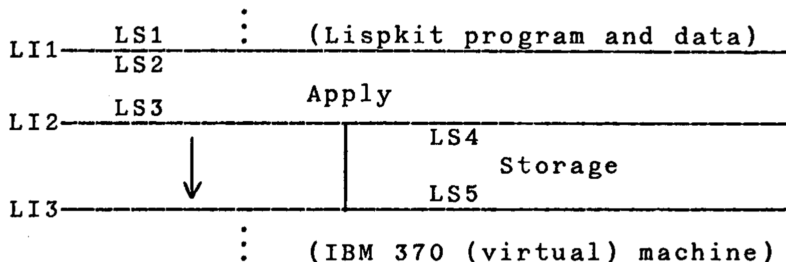
There is a problem here which is identical to that described above for the LS3 statistic. Storage cannot monitor directly its own execution of IBM 370 instructions, and using the MTS timing facility is unreliable. However, the problem can be overcome in the same way, by making the simplifying assumption about AlgolW compilation.

Each of the accessing operations at LI2 is realised at LI3 as simple AlgolW code, and each of the allocating operations consists of a simple piece of code but with a potential garbage collection.

Hence the LS5 statistic is obtained by counting the number of times that the loop bodies of the garbage collector are executed (both mark and scan phases), and adding to this the LS4 statistic, which is already counting the number of basic Storage operations executed.

The statistic obtained is (on average) proportional to the number of IBM 370 instructions executed by the Storage software.

As a visual reminder, here are the five bodies of statistics marked at the appropriate interfaces of the multi-level structure diagram.



6.1.5 Planning the test executions.

To assess the performance of the Apply and Storage software components, the statistics LS1-5 must be collected from a series of test executions of the LM.

In order to obtain a representative assessment of the performance from comparisons between LS1-5, it is necessary to execute a large variety of problems on the LM upper interface. Each problem consists of some Lispkit program and some data. A variety of programs is required since each will place different emphasis on different parts of the Lispkit language, and a variety of data is required in order to span a range of loads on the LM. A large number of

performance data points spanning a wide range of LM loads is desirable, as this will give a much better representation of performance trends than a small number of clustered data points.

I have selected a group of six Lispkit programs, which are intended to cover a range of styles of application. The programs are classified by reference to three independent attributes: list processing versus arithmetic processing, function applications nested linearly versus nested in a tree pattern, and the presence or absence of higher order functions (that is first order versus higher order).

(i) Naive reverse:

```
reverse whererec
        reverse (l)=if eq(l,NIL) then NIL
                    else append(reverse(tail(l)),
                                cons(head(l),NIL))
  and append (l1,l2)=if eq(l1,NIL) then l2
                    else cons(head(l1),append(tail(l1),l2))
```

This program requires rather a large number of function applications to reverse any given list, and hence the tag "naive". It is included partially for the sake of tradition! Naive reverse is a first order list processing program, which falls between the two extremes of linearly and tree nested function applications.

(ii) Reverse with accumulating parameter:

```
revacc whererec
revacc(l) = (rev(l,NIL) whererec
              rev(l,rl)=if eq(l,NIL) then rl
                        else rev(tail(l),
                                 cons(head(l),rl)))
```

This program is a most efficient way of reversing lists. It is a first order, list processing program with linear nesting of applications.

(iii) Quicksort (using an accumulating parameter to avoid appending);

```
λ (l) quicksort(l,NIL)
whererec quicksort(l,rl)= if eq(l,NIL) then rl else
                          if eq(tail(l),NIL)then
                                cons(head(l),rl)
                       else quicksort(lesseq(head(l),tail(l)),
                       cons(head(l),quicksort(greater(
                                      head(l),tail(l)),rl)))

and lesseq(x,l) = if eq(l,NIL) then NIL else
                  if head(l)≤x then cons(head(l),lesseq(x,
                                                tail(l)))
                  else  lesseq (x,tail(l))
and  greater(x,l) = if eq(l,NIL) then NIL else
                    if head(l)≤x then greater(x,tail(l))
                  else cons(head(l),greater(x,tail(l)))
```

This is a first order, list processing program. The nesting of function applications depends on the initial ordering of the input list l. For this experiment I intend to use data which invokes the most branching computation, in other words for each list l that quicksort receives, tail(l) will be an equal mix of values greater than and less than or equal to head(l), and also this property will hold recursively for lesseq(head(l),tail(l)) and greater (head(l),tail(l)). In this case quicksort has a tree nested structure of function applications.

(iv) Iterative summing: (Assuming m≤n)

```
sum whererec sum(m,n) = if eq(m,n) then m
                         else m+sum(m+1,n)
```

This program is first order, arithmetic and with linearly nested function applications.

(v) Powering. Computes n**k using exactly (n**k)+1 function applications:

```
λ(n,k) if eq(n,1) then count(1) else count(pow(n,k)-k)+k
whererec pow(n,k) = if eq(k,1) then n
                    else n*pow(n,k-1)
    and  count(n) = if eq(n,1) then 1 else
                     if eq(n,2) then count(1)+1
                    else count((n-1) div 2)+
                         count((n-1) div 2+(n-1)rem 2)+1
```

This program is first order, arithmetic, and has a tree structured nesting of function applications. The form of the count function was designed to ensure that the depth of nesting remains reasonable, at about (k log n), and it achieves this by imposing a branching structure on the computation.

(vi) Higher order iterative summing:

```
λ(m,n)repeat(sum)(m,n,inc,end)
whererec repeat (op) = λ(m,n,modif,finished)
                         if finished (m,n) then m
                       else op(m,repeat(op)(modif(m),n,
                                   modif,finished))
    and  inc(m) = m+1
    and  sum(x,y) = x+y
    and  end(m,n) = eq(m,n)
```

This program was designed simply to make heavy use of higher order functions, in contrast to the sum function, (iv) above. It is a higher order, linearly nesting, arithmetic program.

The selection of data for each of the programs above is determined purely by practical constraints. The scope of the data should be as large as possible, and the bounds are set by the time available for carrying out experiments and by the amount of s-expression storage available (smaller stores give longer execution times, and also set a maximum on the size of computation possible). The data selected is given in Appendix E, with the tabulated performance data, and is outlined in the next section.

6.1.6 Experimental results.

The six programs given above have been executed on the LM. The data chosen covers wide range of loads on the LM:

(i) Lists of lengths between 1 and 300 were processed by naive reverse.

(ii) Lists of lengths between 1 and 1000 were reversed by reverse with accumulating parameter.

(iii) "Worst case" lists of lengths between 1 and 1023 were sorted by quicksort.

(iv) Series of numbers from 1 to various points between 1 and 1000 were summed iteratively.

(v) Various powers of 2 from 1 to 15 were computed.

(vi) Series of numbers from 1 to various points between 1 and 1000 were summed by the higher order program.

For the experiments the capacity of the s-expression storage was kept to 50000 cells to be allocated to constructed nodes and numbers (which can be created during a computation), and 2000 cells for symbols (which cannot be created during a computation). Garbage collection covers the cons and number storage only. The load on the storage management varies considerably between executions. In particular reverse with accumulating parameter and iterative summing use little storage, whereas naive reverse and powering are very greedy.

Appendix E contains a summary of the experimental measurements obtained. Each table, one for each of the six test programs, shows the variation of statistics LS1-5 with the size of problem tackled. The variation of each individual statistic is seen by scanning down the appropriate column.

To obtain a <u>performance assessment,</u> of a particular software component, from the raw data contained in the tables, two columns must be selected (from the same table) which correspond to the abstract executions at the upper and lower interfaces of the software component. The values in the lower interface's column must then be related to the values in the upper interface's column - the latter variable is the independent variable, and the former is the dependent variable in any graphical or algebraic representation of the relationship.

The following pages show, graphically and algebraically, the four interesting comparisons described earlier. Each comparison covers the six test programs, and hence there are six graphs to be examined for each comparison.

Some comments on the presentation of the graphs are necessary. Each graph is titled by the name of its table in Appendix E, and the axes are labelled with the names of the columns from which the statistics are taken. The correspondence between tables and Lispkit programs is as follows:

Table 1: Naive reverse, Table 2: Reverse with accumulating parameter, Table 3: Quicksort, Table 4: Summing, Table 5: Higher order summing, Table 6: Powering. No scales are marked on the axes, to avoid unnecessary detail; instead each statistic has been scaled independently so that the maximum experimental value is represented by exactly 20 units on the graphs; the graphs are intended only to give a visual indication of trends, and the precise details are retained in the tables of Appendix E. The experimental points have been joined by straight line segments, again to indicate the trends; there is no implication that these represent fitted curves, or that any intermediate points would lie on the segments.

6.1.6.1 Inherent performance of the test program algorithms.

Figure 1 shows, for each of the six Lispkit programs, the number of function applications executed (LS1, which includes <u>where</u> and <u>whererec</u> expressions) as a function of the data supplied to the program. Where the data is a list the length of the list is taken as a representative parameter.

The normally accepted performance of each algorithm is quite obvious, and the empirical points fit the following equations precisely (list length is l):

Table 1: LS1 = l*l/2+3*l/2+2

Table 2: LS1 = l+4

Table 3: LS1 = 2*l*log(l+1)+2*log(l+1)-2*l+1

(logs to base 2)

Table 4: LS1 = n+1

Table 5: LS1 = 5*n

Table 5: LS1 = 2**k+2

Figure 1

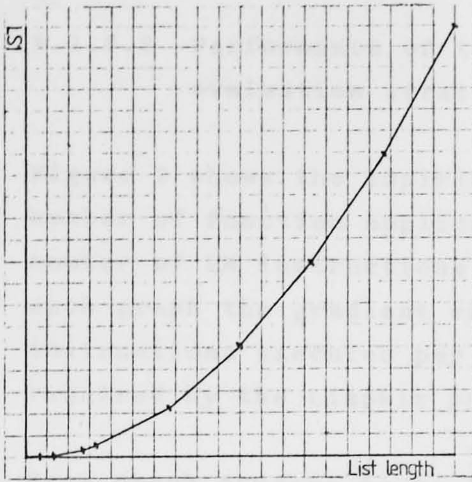


Table 1

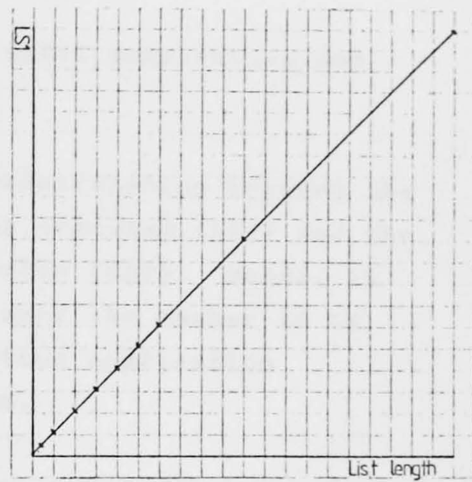


Table 2

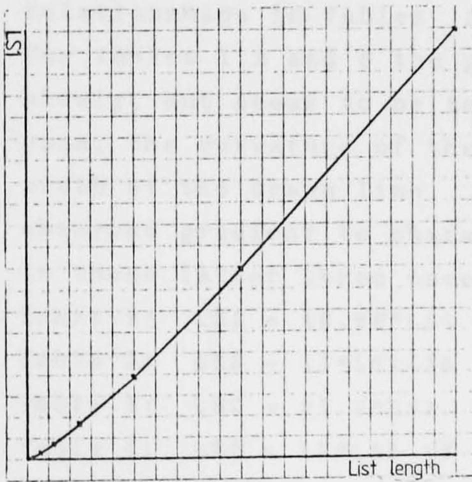


Table 3

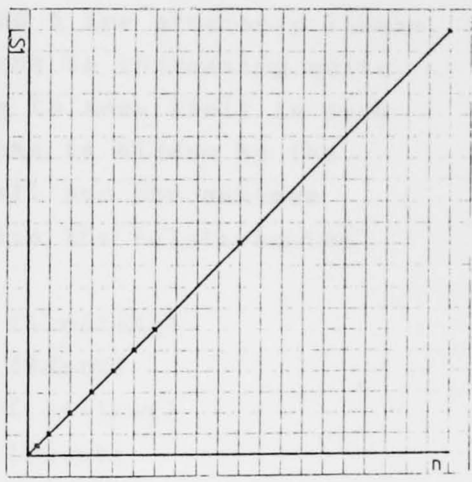


Table 4

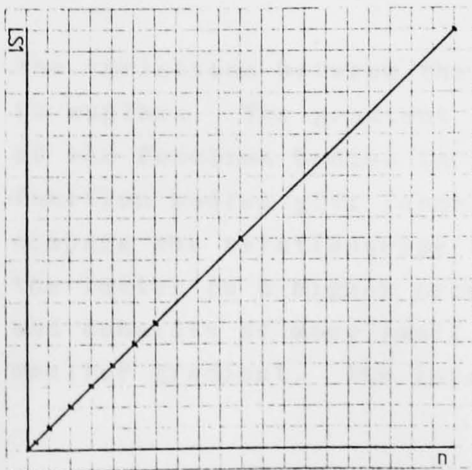


Table 5

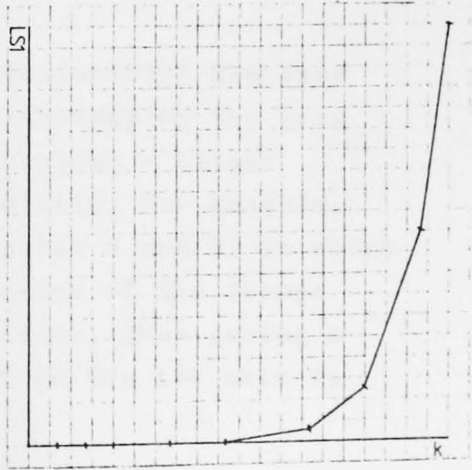


Table 6

6.1.6.2 Performance of the Lispkit compilation and evaluation strategy.

Figure 2 shows the empirical relationships between the number of function applications executed (LS1) and the number of LM instructions executed (LS2). Hence, in each graph the gradient represents the number of LM instructions executed per function application required by the Lispkit program.

Each graph appears to be linear, but a close examination of the tables in Appendix E shows that only the relationships in Tables 2,4 and 5 are precisely linear. For Tables 1,3 and 6 the gradient is increasing quite slowly, but seems to be tending to some limit in each case; the curvature of the graphs is hidden by the width of the drawn line. I shall use the maximum observed gradient to characterise the relationships in these latter three cases:

```
Table 1:  LS2 = 16.98*LS1      (Limiting)
Table 2:  LS2 = 17*LS1-36      (Exact)
Table 3:  LS2 = 21.2*LS1       (Limiting)
Table 4:  LS2 = 17*LS1-17      (Exact)
Table 5:  LS2 = 9.8*LS1        (Exact)
Table 6:  LS2 = 22.5*LS1       (Limiting)
```

The variations between these relationships are easy to explain. The gradient is determined by the size of the function bodies in each program - larger function bodies give larger gradients; for example, compare the relationships for Tables 4 and 5, in which the latter is a higher order version of the former and consists of many small functions, thus giving a smaller gradient. The intercept on the LS2 axis is

not obtainable for Tables 1,3 and 6; however the intercept is not a particularly useful observation as it is determined by the code to be executed before recursion starts, and by the amount of code to be executed in the base cases of the recursive functions.

The difference between the linear relationships of Tables 2,4 and 5, and the "tending to linear" character of Tables 1,3 and 6 is due to the different dynamic structure of the programs. In the programs for Tables 2,4 and 5 the recursion is caused by only <u>one</u> function, whereas for Tables 1 and 6 the recursion is shared between two functions (three in the case of Table 3) but in a way which depends on the data. For example, naive reverse contains a reverse function and an append function, and with larger lists to be reversed the proportion of append applications increases and dominates the relationship between LS1 and LS2; there are 17 LM instructions to be executed for a non-base case call of append, and this is thus the limiting gradient; contributions from the small base case of append, and the non-base case of reverse give an actual gradient of less than 17.

The conclusion here is that the results strongly suggest a linear performance characteristic for the compilation and evaluation strategy. In other words the number of LM instructions to be executed is some linear function of the number of expression evaluations to be expected from a model interpretation of a Lispkit program. A particular statistic which can be extracted from the results is the constant of proportionality in the linear performance; this appears to be about 20 for Lispkit

programs without large function bodies, though this obviously varies between programs and could be quite large (for example, for a compiler).

Figure 2

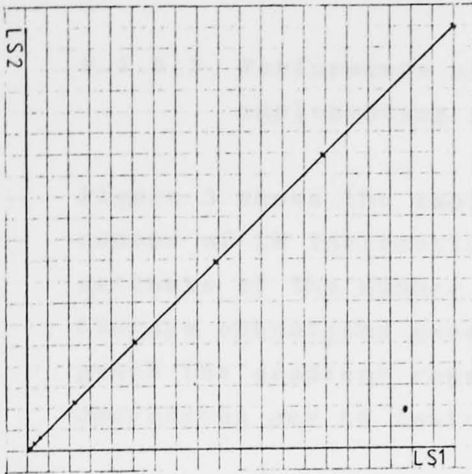


Table 1

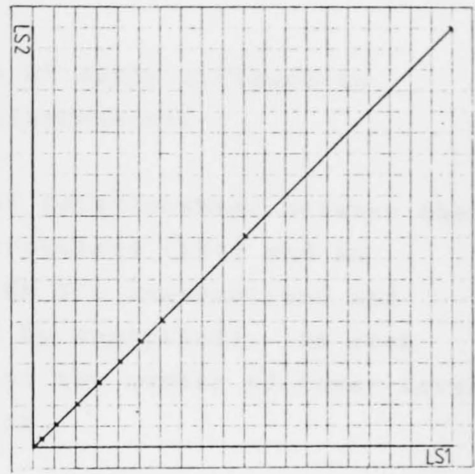


Table 2

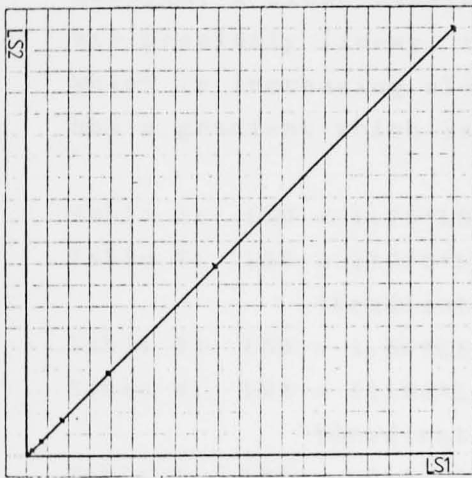


Table 3

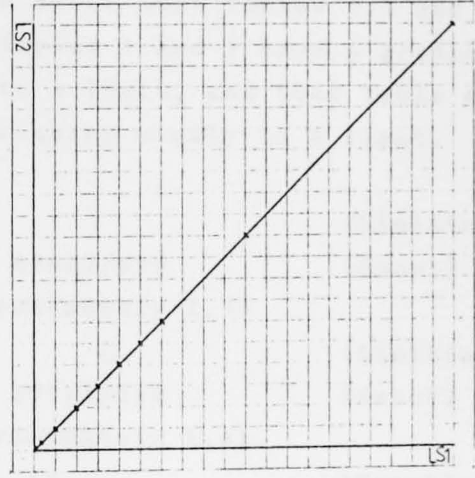


Table 4

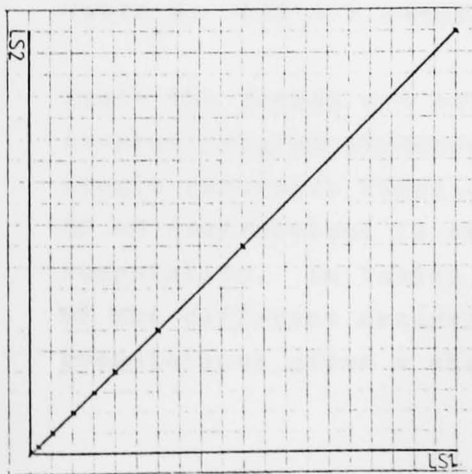


Table 5

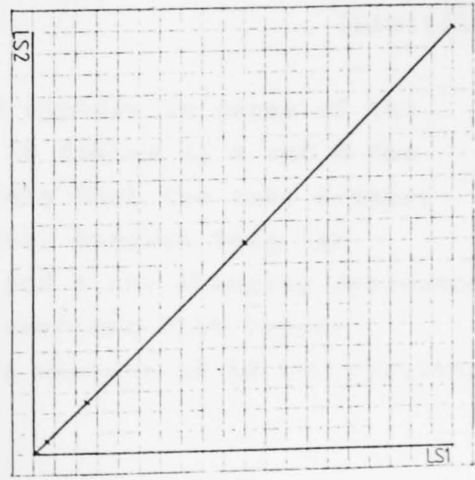


Table 6

6.1.6.3 Performance of the LM Apply software in implementing LM instructions.

Figure 3 shows the empirical relationships between the number of LM instructions processed (LS2) and an estimate of the number of IBM 370 instructions and Storage operations executed by Apply(LS3). In each graph the gradient represents the number of lower level operations per LM instruction.

Each graph appears to be linear, but, as in the previous section, a close examination shows that Tables 2,4 and 5 are precisely linear, that Tables 1 and 3 have a gradient which is increasing slowly to a limit, and that Table 6 has a gradient which is decreasing slowly to a limit.

Table 1:	LS3 = 1.76*LS2	(Limiting)
Table 2:	LS3 = (1073*LS2 - 6660)/629	(Exact)
	(Gradient approximately 1.7)	
Table 3:	LS3 = 1.86*LS2	(Limiting)
Table 4:	LS3 = (1184*LS2 - 3774)/629	(Exact)
	(Gradient approximately 1.9)	
Table 5:	LS3 = 101*LS2/49 - 12	(Exact)
	(Gradient approximately 2.1)	
Table 6:	LS3 = 1.44*LS2	(Limiting)

Again the trends are easy to explain in terms of the Lispkit program structure. In Tables 2, 4 and 5 the single recursive function means that the same mixture of LM instructions is executed, however long the computation. In Tables 1,3 and 6 the changing dominance of the different recursive functions with longer computations gives a changing mixture of LM instructions

which becomes dominated by a particular proportion of each instruction for each program; since each instruction is implemented by a different amount of Apply code, this gives the observed changing gradient.

These results suggest that the Apply software has a linear performance in implementing LM instructions.

Figure 3

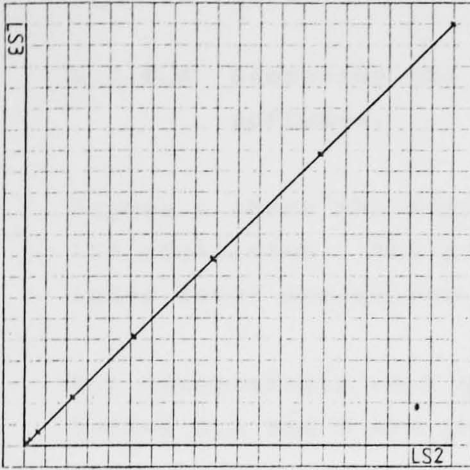


Table 1

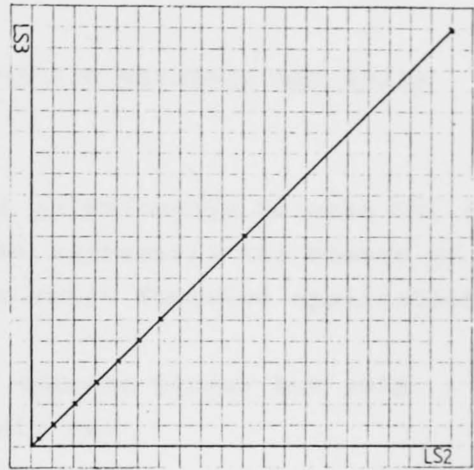


Table 2

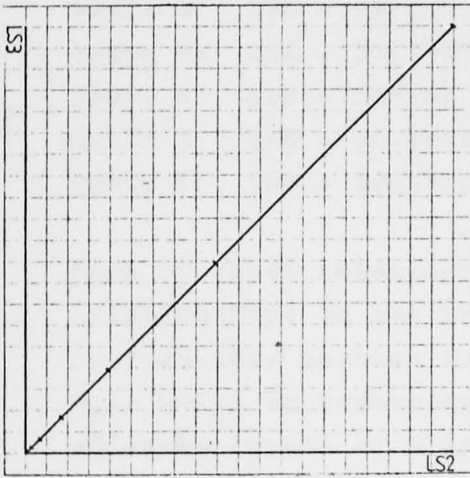


Table 3

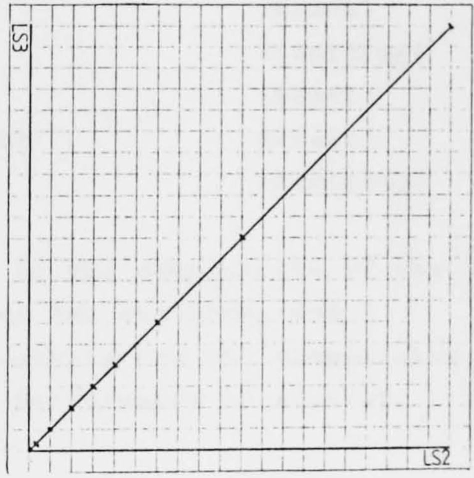


Table 4

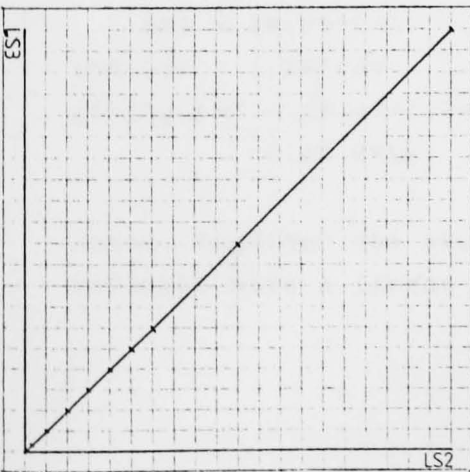


Table 5

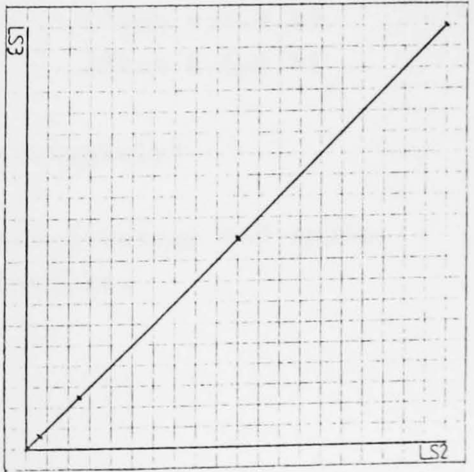


Table 6

6.1.6.4 Composing the evaluation strategy and Apply software.

Figure 4 shows the empirical relationship between LS1 and LS3 statistics. The gradient represents the number of lower level operations per Lispkit function application.

Not surprisingly each graph appears linear but only Tables 2,4 and 5 are precisely so:

Table 1:	LS3 = 29.9*LS1	(Limiting)
Table 2:	LS3 = 29*LS1 - 72	(Exact)
Table 3:	LS3 = 39.5*LS1	(Limiting)
Table 4:	LS3 = 32*LS1 - 38	(Exact)
Table 5:	LS3 = 20.2*LS1 - 12	(Exact)
Table 6:	LS3 = 32.5*LS1	(Limiting)

Each of these relationships is the composition of the relationships in the previous two sections, and I include this section to illustrate how the composition rule helps in combining the performance of several adjacent levels of a system.

For example, in the case of Table 1:

LS2 = 16.98*LS1 (from 6.1.6.2)

and LS3 = 1.76*LS2 (from 6.1.6.3)

giving LS3 = 16.98*1.76*LS1

= 29.9*LS1 (approximately)

Hence, together the evaluation strategy and Apply software have a linear performance.

Figure 4

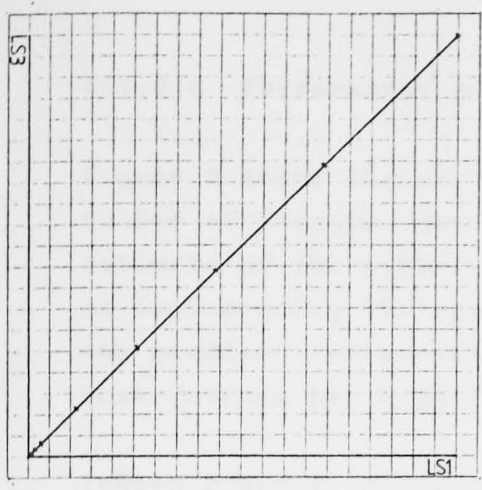


Table 1

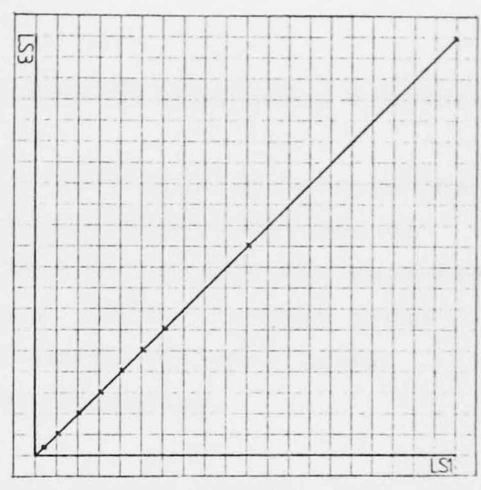


Table 2

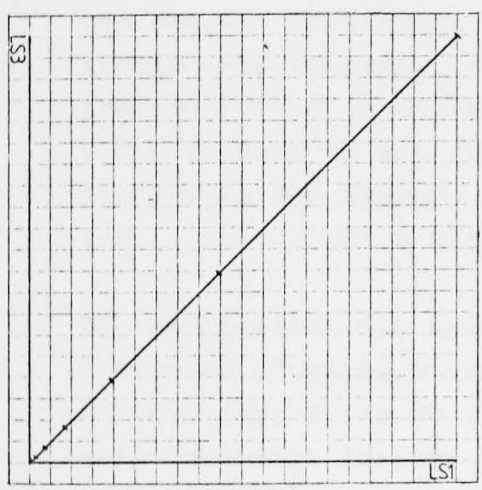


Table 3

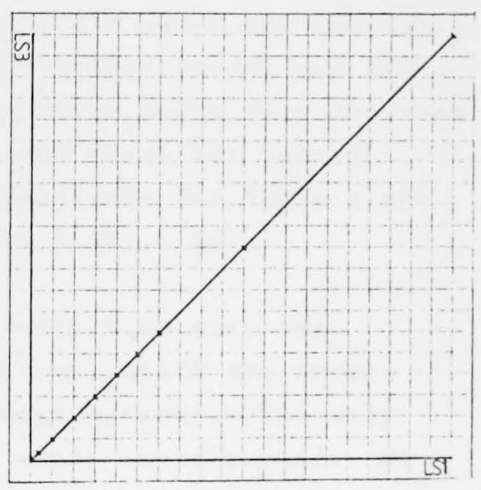


Table 4

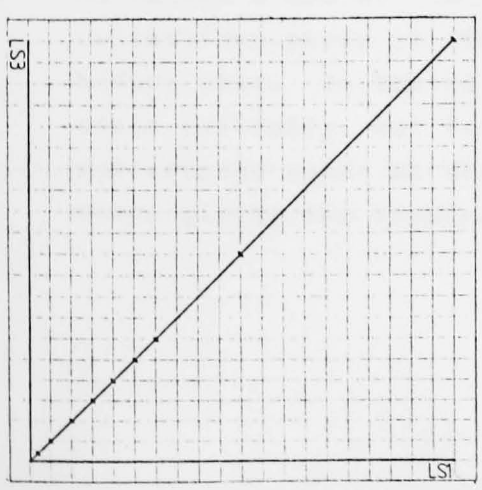


Table 5

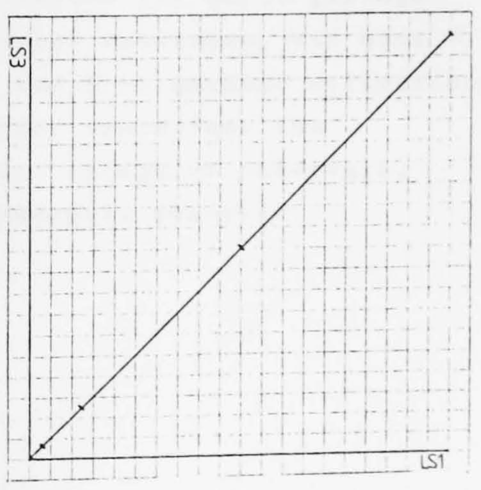


Table 6

6.1.6.5 Performance of the Storage management component.

Figure 5 shows the empirical relationships between the number of Storage operations processed (LS4) and an estimate of the number of IBM 370 instructions executed in implementing the operations (LS5). In each graph the gradient represents the number of lower level instructions per higher level operation.

Tables 2 and 4 give precisely linear graphs, but Table 5 gives a graph which curves sharply upwards for the longest execution. Tables 1,3 and 6 are initially linear relationships, but for longer executions the gradients show an overall increase which does not seem to be tending to any limit, and which has erratic decreases; for example the gradients for Table 1 are 1.0,...,1.0,1.079,1.074,1.072,1.071,1.082.

The explanation for these trends lies with the frequency of garbage collections, which are never invoked explicitly as a Storage operation but only occur when the heap of list cells is exhausted. No garbage collections occur in the executions recorded in Tables 2 and 4. In Table 5 one (or more) garbage collections occur in the longest execution, but none before that. In Tables 1,3 and 6 no garbage collections occur initially, but for longer executions the collections occur at discrete, irregular intervals, hence giving the erratic gradients observed.

These results show that the Storage management component of the LM has an unusual, certainly non-linear, performance characteristic. However, there is certainly not adequate experimental evidence here to clarify any peculiarities of behaviour, or to form any conclusions on the performance of the Storage management. More evidence, of the kind shown in Figure 6, is required; this shows the same Lispkit execution as Table 1, but the heap size has been reduced to 4500 cells and the storage is consequently more heavily loaded. The gradient is initially irregular, but for longer executions it increases rapidly with no apparent limit.

6.1.6.6 Conclusions on the performance of the LM Lispkit implementation.

The experiments reported above suggest very strongly that this particular compilation and evaluation strategy, and this particular design for the Apply component of the LM each have a linear performance, and hence that Lispkit programs may be executed in such a way that the number of Storage operations and IBM 370 instructions required is a linear function of the number of function applications expected from a model interpretation of the program.

However, evidence for the performance characteristics of the Storage management component of the LM is inconclusive and more investigation is required.

Figure 5

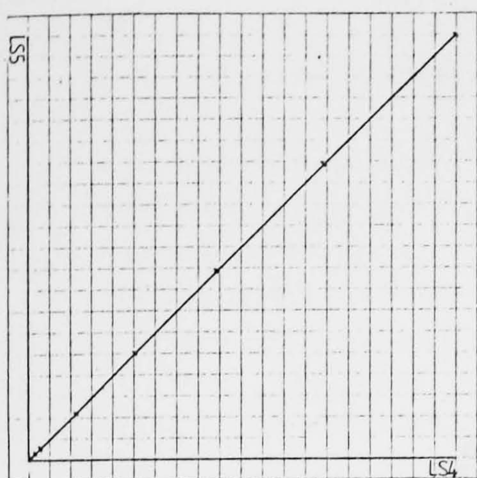


Table 1

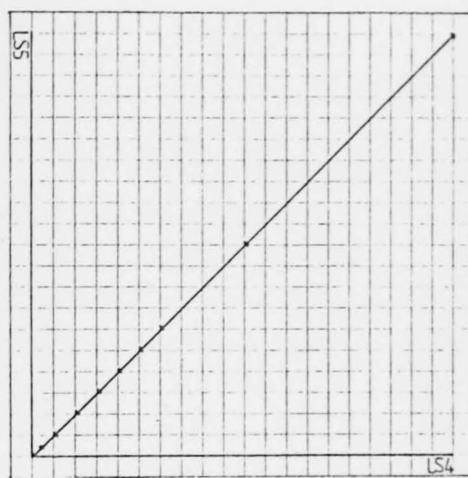


Table 2

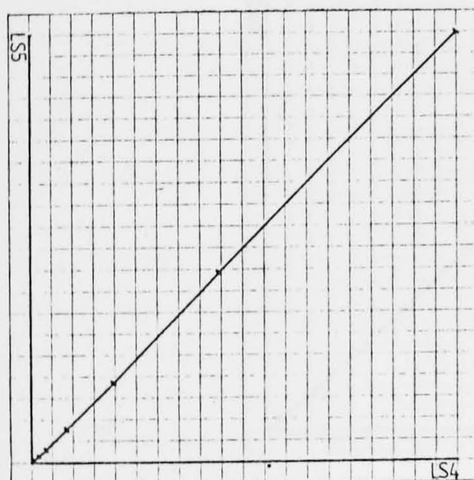


Table 3

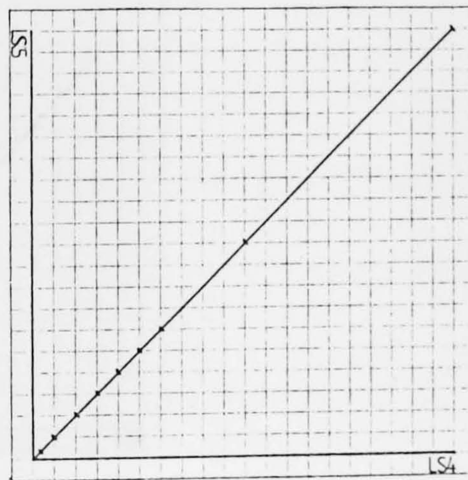


Table 4

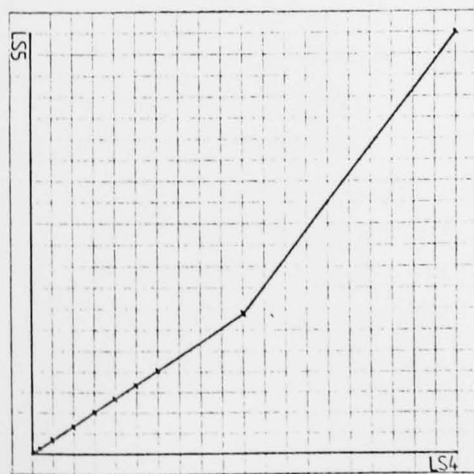


Table 5

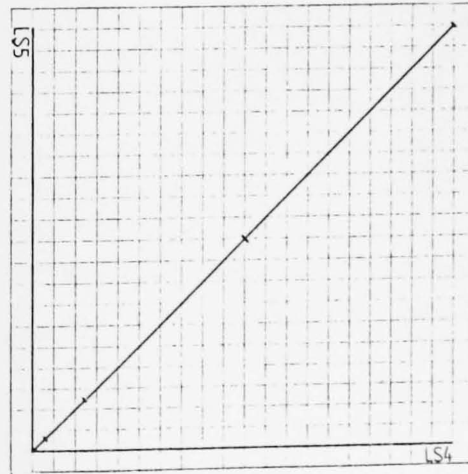


Table 6

Figure 6

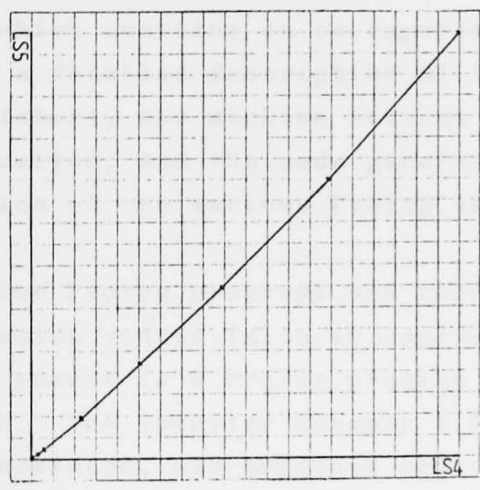


Table 1

6.2 A Prolog pseudo-machine.

The operational model for the execution of Prolog programs, given in Chapter 3, can be realised reasonably directly by a special purpose Prolog machine (PM). Prolog programs are precompiled to an intermediate machine code, which consists essentially of linear sequences of instructions for the PM. The PM has a conventional sequential machine architecture, but in which all programs, data and results are held in nine registers, DE, A, F, L, C, R, DU, B and N, each of which contains an su-expression. Appendix D contains a detailed description of the roles of the nine registers, the machine actions determined by each PM instruction, and the code generated by the compilation of the various Prolog language constructs.

In practice Prolog programs are represented in an su-expression syntax (also in Appendix D), and the compiler itself is a Prolog program which executes on the PM. The compiler accepts a Prolog program as an su-expression, and produces object code, which is also an su-expression, suitable for re-input to the PM as a compiled program.

Many of the characteristics of the structure and qualitative behaviour of the PM are very similar to those of the LM. This is also true of the reasoning necessary in setting up a practical analysis of the performance of the PM components. Consequently I shall present this discussion of the PM in an outline form, relying heavily on the material in the discussion of the LM.

Although the basic facilities provided by the PM are rather more sophisticated than those of the LM, I believe that it is nevertheless still interesting for its relative simplicity and economy.

6.2.1 The software components.

The PM is implemented as a medium sized AlgolW program (several hundred lines, about twice the size of the LM), which is compiled to execute on an IBM 370/168 under the supervision of MTS.

The PM software can be divided into five component parts:
su-expression storage management, su-expression input, su-expression output, the main Prolog evaluator ("apply"), and a group of routines which provide the evaluator with high level support (unification, backtracking and checking data structures before and after negated conditions).

The su-expression storage management is used by all parts of the PM. Storage is provided in a collection of arrays. Storage cells are allocated to constructed nodes, numbers, symbols, and unknowns (in addition cells allocated to unknowns can be modified by unification to represent indirect pointers to su-expressions, and backtracking can reverse this transformation).
The storage is managed as a heap with a mark and scan garbage collector (marking from the nine registers and a temporary register W). A storage initialisation procedure is provided, and also a "forcing" function which will follow a chain of indirections to yield

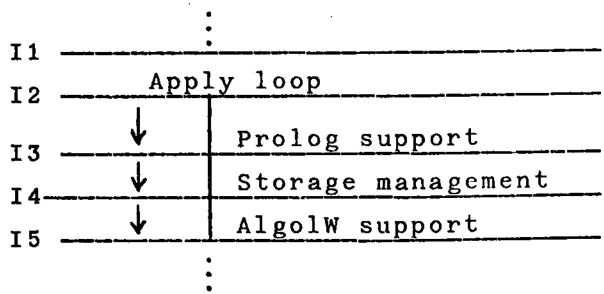
the referenced su-expression. Each access to a register, each access to a cell, each cell allocation request and each forcing is considered to be an operation provided by the storage management service.

The apply routine simply initialises the nine machine registers and then loops iteratively through the PM machine code program. It includes a large, but simple, routine implementing the primitive predicates.

The overall pattern of activity in the PM is a simple sequence: su-expression storage is initialised, the program and argument su-expressions are input, the program is applied to the arguments until the first solution (if any) is generated, the solution su-expressions are output, then the computation backtracks and a second solution is sought. The actions of application, output and backtracking are repeated until the search space of the program is exhausted.

Performance analysis will be concerned with the apply phase of PM execution, and with the storage management and Prolog support routines during this phase.

Initially treat the PM as a multi-level interpreter with the structure:



Above I1 will be a compiled Prolog program and its data. Below I5 will be a (virtual) machine to execute IBM 370 machine instructions.

6.2.2 General behavioural considerations.

At I1 the PM machine instructions of a compiled Prolog program are scanned and executed.

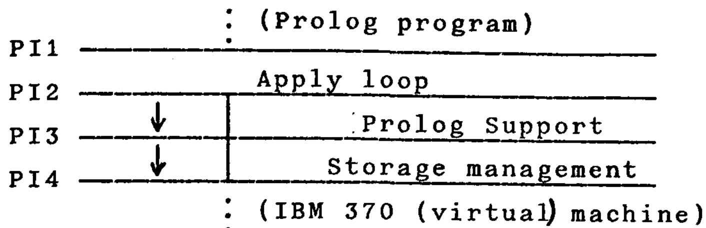
At I2 the actions of the apply loop are presented as IBM 370 instructions, calls on the storage management and AlgolW support (these three categories are passed on to I3), and as calls on the Prolog support routines.

At I3 the Prolog support realises its own actions as IBM 370 instructions, calls on AlgolW support (passed on to I4), and as calls on the storage management. These are mixed with the operations passed on from I2.

At I4 the storage management actions are realised as IBM 370 instructions (passed on to I5), and as calls on the AlgolW support.

At I5 all operations are IBM 370 instructions. These include the execution of the AlgolW support routines.

The inaccessibility of the AlgolW support software prompts a simplified multi-level structure in which the AlgolW support has been absorbed into the other levels:



The terms PI1, PI2, PI3, PI4, Apply, Support and Storage will be used throughout this description.

6.2.3 Performance assessments to be made.

There are five interesting comparisons to be made between the abstract executions at interfaces PI1-5:

(i) Comparing a model interpretation of the Prolog program with the trace of PM instructions executed at PI1 by Apply will show how effectively the evaluation scheme implements the Prolog language.

(ii) Comparing the model interpretation of the Prolog program with the sequence of IBM 370 instructions and Support and Storage operations executed by Apply at PI2 will show how effectively the detailed code of Apply implements the Prolog language.

(iii) Comparing the trace of PM instructions executed at PI1 with the sequence of instructions and operations at PI2 will show how effectively the detailed code of Apply implements the evaluation strategy which it imposes on the Prolog program,

(iv) Comparing the trace of Support operations at PI2 with the IBM 370 instructions and Storage operations presented by Support at PI3 will show how effectively the detailed code of Support implements its own higher level operations.

(v) Comparing the trace of Storage operations at PI3 with the IBM 370 instructions presented by Storage at PI4 will show how effectively the detailed code of Storage implements its own higher level operations.

6.2.4 Monitoring the behaviour.

Seven bodies of statistics are required in order to make the above comparisons.

The statistic PS1 is a count of the number of predicate cases which are tried during the course of a computation. This is more representative of the work performed by a Prolog program than a count of the number of predicates called, as most try several cases before finding a solution. The statistic has been obtained by manual analysis, and later by extracting the number of executions of the UNIFY instruction from PS2 statistics (see below) after noting a direct correspondence.

The Apply software of the PM is easily modified to count the PM machine instructions which are executed. PS2 is the total number of instructions.

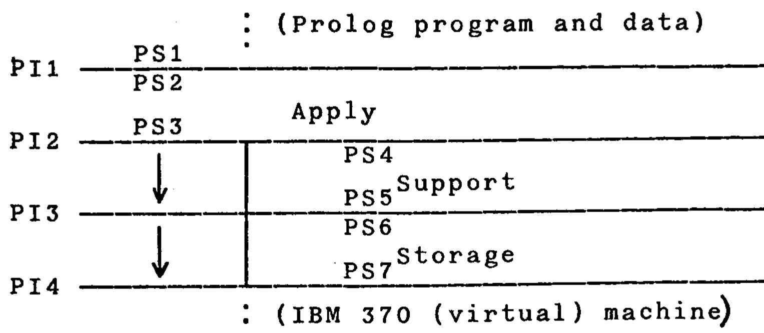
PS3 is proportional to the total number of IBM 370 instructions, Support operations and Storage operations executed by Apply at PI2. It is obtained by counting the loop body executions within Apply; these are the main interpretation loop (already counted for PS2), and small loops for the construction of local environments, and for looking up predicates and variables.

The number of Support operations (<u>PS4</u>) is easily monitored by modifying Apply, and similarly the number of Storage operations (<u>PS6</u>) can be found by modifying Apply and Support.

<u>PS5</u> is proportional to the number of IBM 370 instructions and Storage operations executed by Support; it is found by counting the internal loop body executions of the Support routines (unification, backtracking, data structure checking) and adding to this PS4 which is already counting the number of entries to Support operations.

<u>PS7</u> is proportional to the number of IBM 370 instructions executed by Storage; it is found by counting the loop body executions within Storage operations (garbage collection and forcing loops) and adding PS6, which includes the contribution from in-line accessing and register operations.

As a visual reminder here are the statistics marked on the simplified multi-level structure diagram:



6.2.5 Planning the test executions.

To assess the performance of the Apply, Support and Storage components, the statistics PS1-7 must be collected from a series of executions of the PM.

Five of the programs used to test the LM have been recoded in Prolog, as they cause a similar range of styles of computation. The higher order summing example has not been recoded, as Prolog does not have a higher order predicate capability. Using the same programs will also enable an attempt at absolute comparison between corresponding components of the LM and PM.

(i) Naive reverse:

```
query (l1,l2)←reverse(l1,l2)
reverse (NIL,NIL)←
reverse ((x.l1),l2)←reverse(l1,l3),
                    append(l3,(x),l2)
append (NIL,l,l)←
append ((x.l1),l2,(x.l3))←append(l1,l2,l3)
```

(ii) Reverse with accumulating parameter:

```
query(l1,l2)←revacc(l1,NIL,l2)
revacc (NIL,l,l)←
revacc ((x.l1),l2,l3)←revacc(l1,(x.l2),l3)
```

(iii) Quicksort(with an accumulating parameter):

```
query(l1,l2)←quicksort(l1,NIL,l2)
quicksort (NIL,rl,rl)←
quicksort ((x.l),rl,l1)←partition(x,l,leql,grl),
                        quicksort(leql,(x.rl1),l1),
                        quicksort(grl,rl,rl1)
partition(x,NIL,NIL,NIL)←
partition(x,(y.l),(y.leql),grl)←leq(y,x),
                     partition (x,l,leql,grl)
partition(x,(y.l),leql,(y.grl))←¬leq(y,x),
                     partition(x,l,leql,grl)
```

(iv) Iterative summing: (Assuming $m \leqslant n$)

```
query(m,n,result) ← sum(m,n,result)
sum(m,m,m) ←
sum(m,n,s) ← ¬eq(m,n),add(m,1,m1),
             sum(m1,n,s1),add(m,s1,s)
```

(v) Powering. Compute n**k in (n**k)+2 predicate invocations:

```
query(n,k,pow) ← main(n,k,pow)
main(1,k,pow) ← count(1,pow)
main(n,k,pow) ← ¬ eq(n,1),power(n,k,nk),sub(nk,k,nk1),
                count(nk1,pow1),add(pow1,k,pow)
power(n,1,n) ←
power(n,k,product) ← ¬eq(k,1),sub(k,1,k1),
                     power(n,k1,product1),mul(n,product1,
                                                product)
count(1,1) ←
count(2,x) ← count(1,x1),add(1,x1,x)
count(n,x) ← ¬leq(n,2),sub(n,1,nsub1),div(nsub1,2,
                                            ndiv2)
             count(ndiv2,x1), rem(nsub1,2,r1),
        add(ndiv2,r1,ndiv2p1),count(ndiv2p1,x2),
        add(x1,x2,x3),add(x3,1,x)
```

The data selected for each program is given in Appendix F, with the tabulated performance data, and is outlined in the next section.

6.2.6 Experimental results.

The five programs given above have been executed on the PM. The data chosen covers a range of loads on the PM:

(i) Lists of lengths between 1 and 66 were processed by naive reverse.

(ii) Lists of lengths between 1 and 1000 were processed by reverse with accumulating parameter.

(iii) "Worst case" lists of lengths between 1 and 127 were sorted by quicksort.

(iv) Series of numbers from 1 to points between 1 and 1000 were summed iteratively.

(v) Various powers of 2 from 1 to 10 were computed.

Note that each of the programs has only one result, and following the production of this result the PM will continue to scan, fruitlessly, the remaining search space. For the experimental executions the statistics are gathered only up to the production of this first, and only, result.

For the experiments the capacity of the su-expression storage was kept at 50000 cells to be allocated to constructed nodes, numbers and unknowns (which can all be created during a computation), and 2000 cells for symbols (which cannot be created). Garbage collection covers the cons, number and unknown cells only.

Appendix F contains a summary of the experimental measurements obtained. Each table, one for each program, shows the variation of the statistics PS1-7.

The following pages show, graphically and algebraically, the five interesting comparisons discussed earlier. Each comparison covers the five test programs, and hence there are five graphs to be examined for each comparison.

The graphs are again presented with simple labelling from the tables in Appendix F, and without scales on the axes. The tables correspond to Prolog programs as follows: Table 1: Naive reverse, Table 2: Reverse with accumulating parameter, Table 3: Quicksort, Table 4: Summing, Table 5: Powering. Again the points are joined by straight line segments to indicate trends.

6.2.6.1 Inherent performance of the test program algorithms.

Figure 7 shows, for each of the five Prolog programs, the number of predicate cases executed (PS1) as a function of the data supplied to the program. Where the data is a list, the length of the list is taken as a representative parameter.

The normally accepted performance of each algorithm is quite obvious, and the empirical points fit the following equations (list length is l):

Table 1: PS1 = l*l+2*l+2

Table 2: PS1 = 2*l+2

Table 3: PS1 = 5/2*l*log(l+1)+5/2*log(l+1)-l+2
(logs to base 2)

Table 4: PS1 = 2*n

Table 5: PS1 = 2*2**k+1

Figure 7

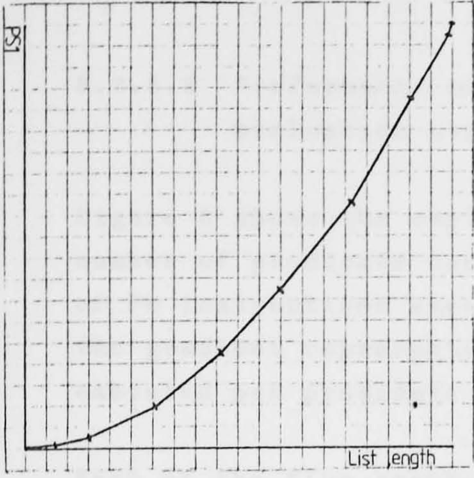


Table 1

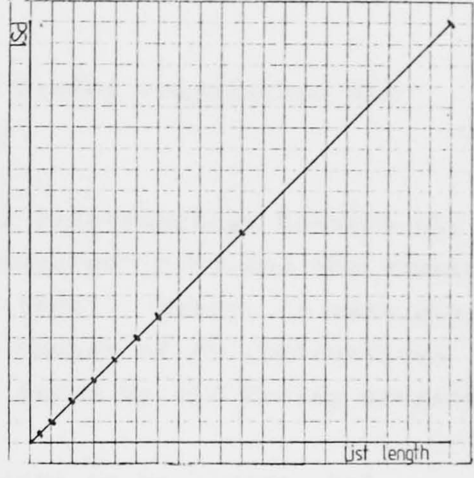


Table 2

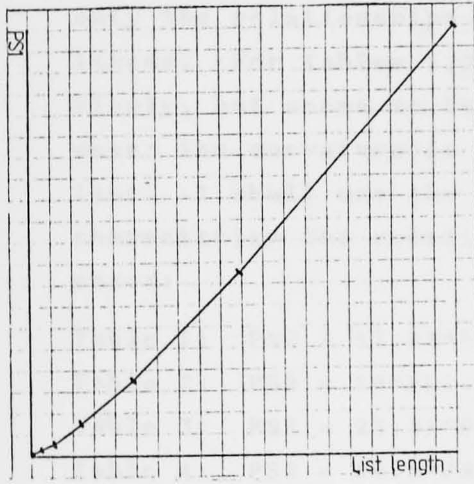


Table 3

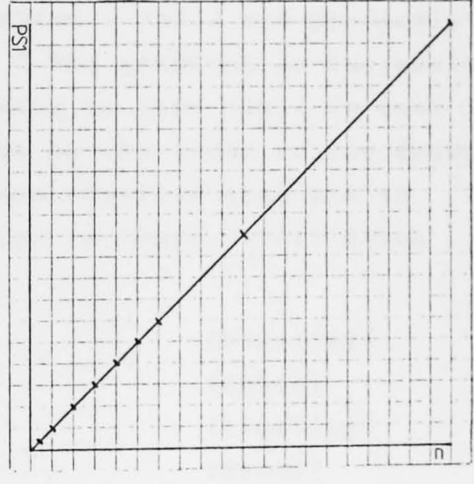


Table 4

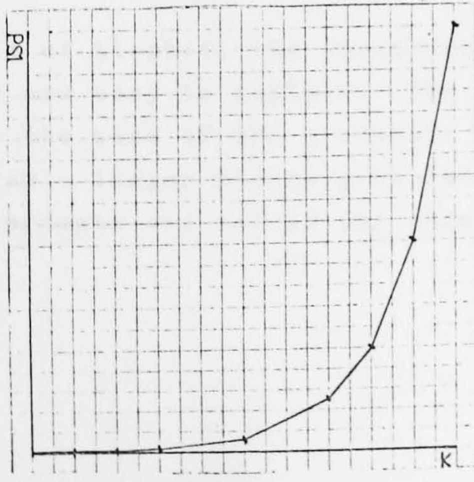


Table 5

6.2.6.2 Performance of the Prolog compilation and evaluation strategy.

Figure 8 shows the empirical relationships between the number of predicate cases executed (PS1) and the number of PM instructions executed (PS2). Hence, in each graph the gradient represents the number of PM instructions executed per predicate case tried by the Prolog program.

Each of the five graphs appears to be linear, but examination of the tables in Appendix F reveals that only the relationships in Tables 2 and 4 are precisely linear. For Tables 1,3 and 5 the gradient is increasing slowly, but seems to be tending to some limit in each case; the curvature is hidden by the width of the drawn line. I shall use the maximum observed gradient to characterise the relationships in these latter three cases:

Table 1:	PS2 = 17.98*PS1	(Limiting)
Table 2:	PS2 = 18*PS1-3	(Exact)
Table 3:	PS2 = 23.89*PS1	(Limiting)
Table 4:	PS2 = 29.5*PS1	(Exact)
Table 5:	PS2 = 26.73*PS1	(Limiting)

As in the case of Lispkit, the variations between these relationships are easy to explain. The gradient is determined by the size of the predicate case bodies in each program - larger bodies give larger gradients. Again the intercepts are not of any interest.

The linear relationships shown by Tables 2 and 4 are due to the dynamic structure of the respective programs which consist of only one recursive predicate. The "tending to linear" relationships shown by Tables 1, 3 and 5 are caused by programs in which two recursive predicates share the computation, but in a proportion which alters with the data; for longer computations one of the two predicates dominates the computation, and the size of its cases determine the limiting gradient.

The conclusion here is that the results strongly suggest a linear performance characteristic for the compilation and evaluation strategy. In other words the number of PM instructions to be executed is some linear function of the number of predicate case (or condition) executions which would be expected from a model interpretation of a Prolog program. For Prolog programs with small bodied predicate cases the constant of proportionality in the relationship appears to be about 20.

Figure 8

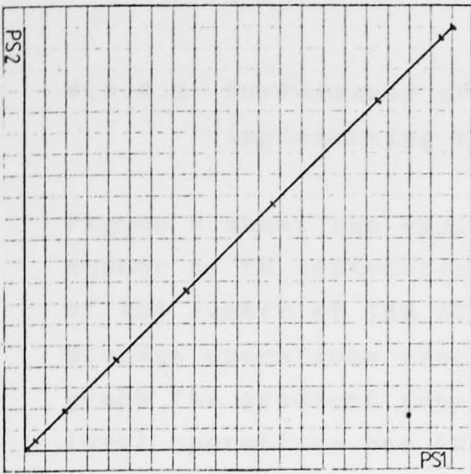


Table 1

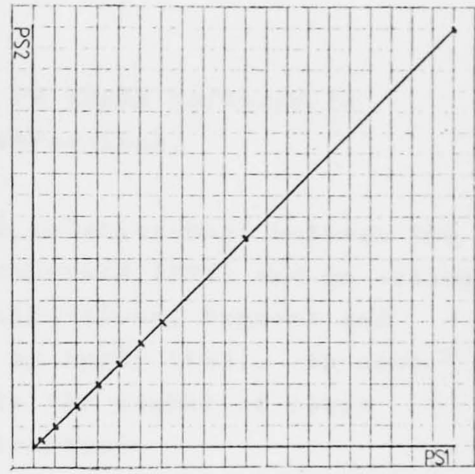


Table 2

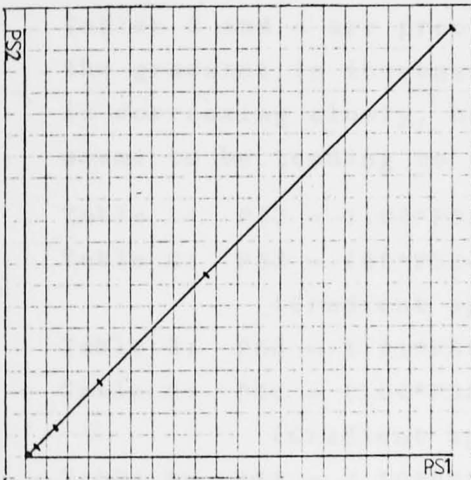


Table 3

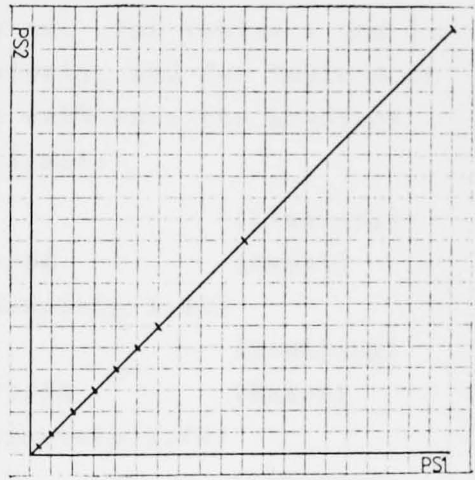


Table 4

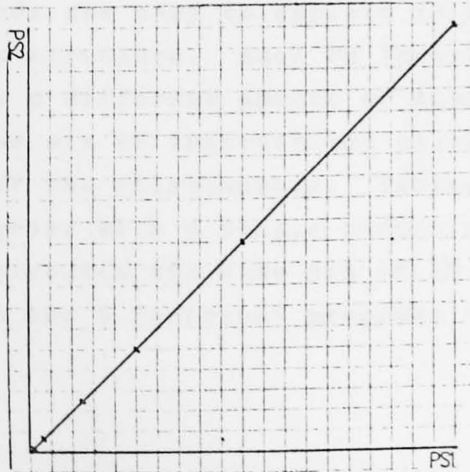


Table 5

6.2.6.3 Performance of the PM Apply software in implementing PM instructions.

Figure 9 shows the empirical relationships between the number of PM instructions processed (PS2) and an estimate of the number of IBM 370 instructions, Support and Storage operations executed by Apply (PS3). In each graph the gradient represents the number of lower level operations per PM instruction.

Each graph appears linear, but again only those for Tables 2 and 4 are precisely so. For Tables 1 and 5 the gradient is increasing slowly, and for Table 3 it is decreasing slowly, but in each case the gradient seems to be tending to some limit:

Table 1: PS3 = 1.83*PS2 (Limiting)

Table 2: PS3 = (65*PS2-453)/36 (Exact)
(Gradient approximately 1.81)

Table 3: PS3 = 1.91*PS2 (Limiting)

Table 4: PS3 = (104*PS2-277)/59 (Exact)
(Gradient approximately 1.76)

Table 5: PS3 = 2.10*PS2 (Limiting)

Again the trends are easy to explain in terms of the Prolog program structure. Each PM instruction is implemented by a different amount of Apply code, and the precise mix of instructions executed determines the gradient of the relationship. Tables 2 and 4 represent programs with a single recursive predicate which always executes the same mix of PM instructions, but Tables 1,3 and 5 represent programs in which one

of two recursive predicates (and hence one of two particular mixes of PM instructions) grows in dominance with longer computations.

These results suggest that the Apply software has a linear performance in implementing PM instructions.

Figure 9

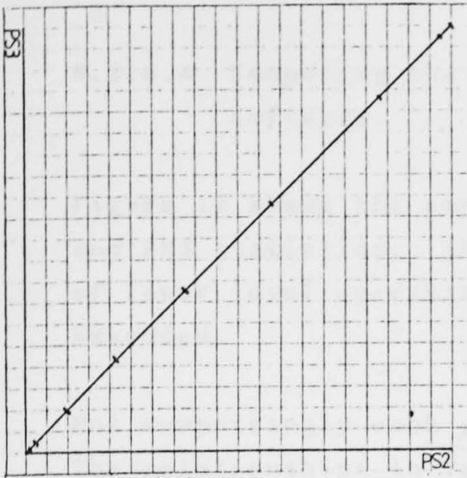


Table 1

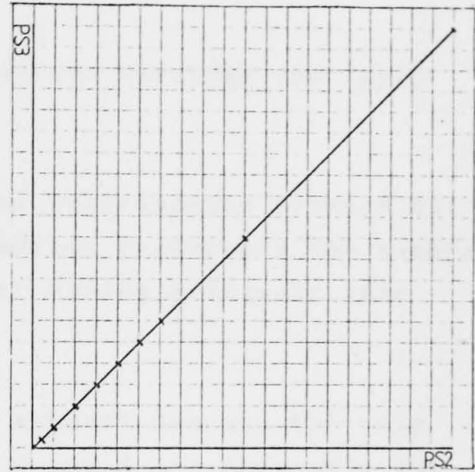


Table 2

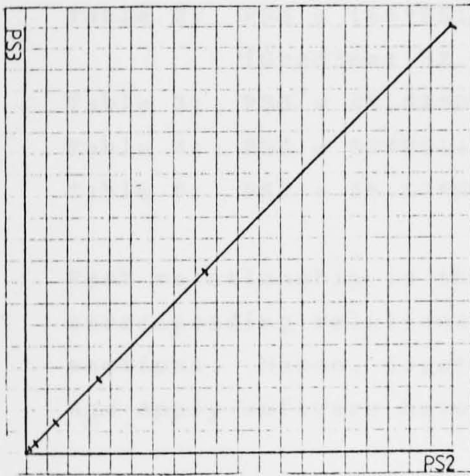


Table 3

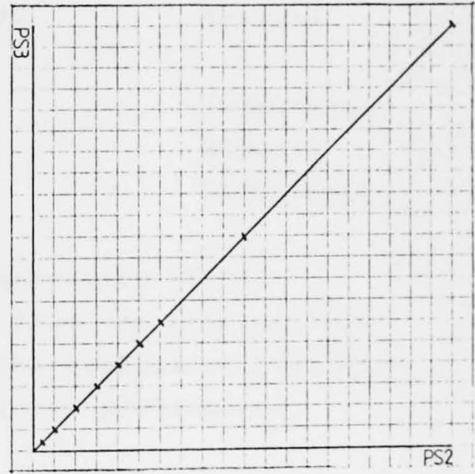


Table 4

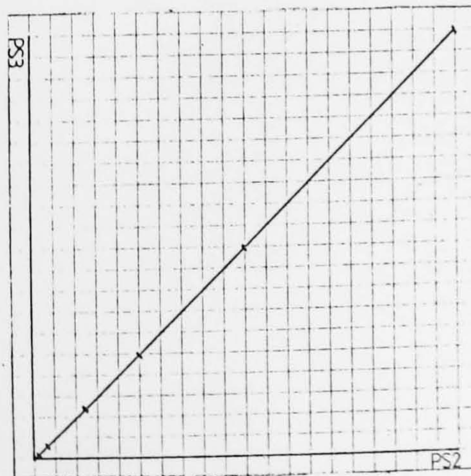


Table 5

6.2.6.4 Composing the evaluation strategy and Apply software.

Figure 10 shows the empirical relationship between PS1 and PS3 statistics. The gradient represents the number of lower level operations per Prolog predicate case executed.

Not surprisingly each graph appears linear, but only the relationships in Tables 2 and 4 are precisely so:

Table 1:	PS3 = 32.88*PS1	(Limiting)
Table 2:	PS3 = (65*PS1-36)/2	(Exact)
	(Gradient 32.5)	
Table 3:	PS3 = 45.58*PS1	(Limiting)
Table 4:	PS3 = 52*PS1+3	(Exact)
Table 5:	PS3 = 56.17*PS1	(Limiting)

Each relationship is the composition of the corresponding relationships from the previous two sections. Hence, together the evaluation strategy and Apply software have a linear performance.

Figure 10

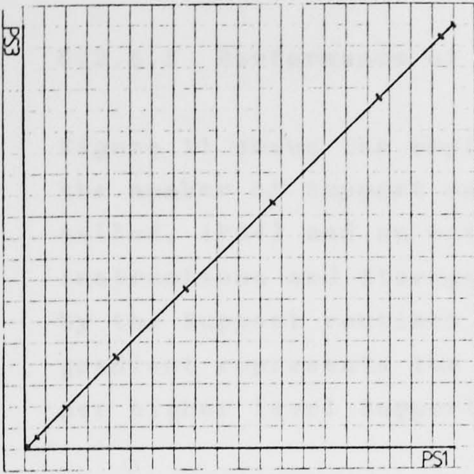


Table 1

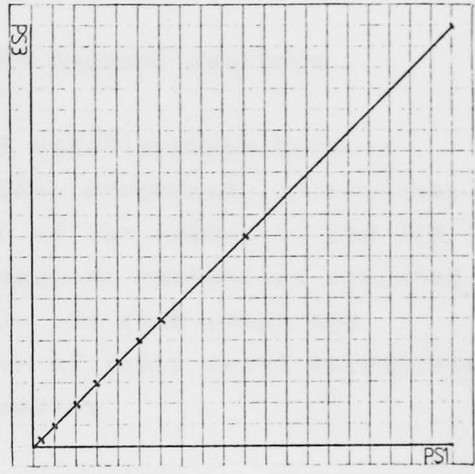


Table 2

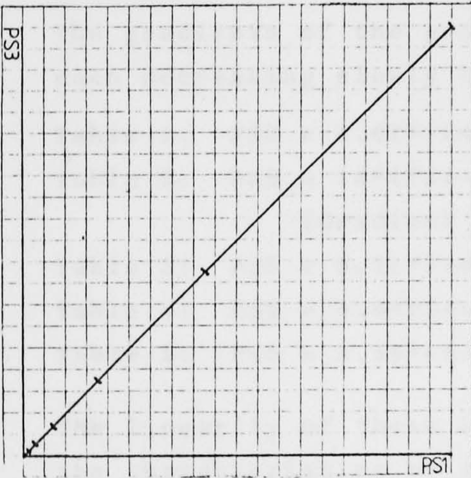


Table 3

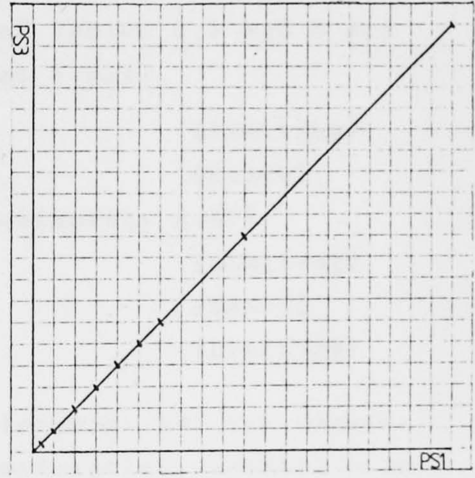


Table 4

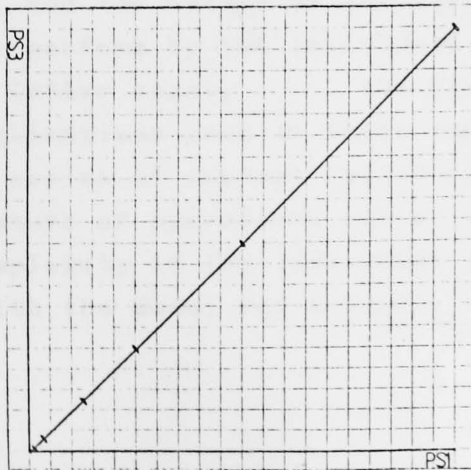


Table 5

6.2.6.5 Performance of the PM Support routines.

Figure 11 shows the empirical relationships between the number of Support operations processed, or routines called, (PS4) and an estimate of the number of IBM 370 instructions and Storage management operations executed by the Support routines (PS5). In each graph the gradient represents the number of lower level operations per higher level Support operation.

Again each graph appears to be linear, but only the relationships in Tables 2 and 4 are precisely so. The gradients of the graphs of Tables 1,3 and 5 are each decreasing slowly to some limit:

Table 1: PS5 = 4.67*PS4 (Limiting)
Table 2: PS5 = 14*(PS4+1)/3 (Exact)
(Gradient approximately 4.67)
Table 3: PS5 = 6.31*PS4 (Limiting)
Table 4: PS5 = 4.4*PS4+7.2 (Exact)
Table 5: PS5 = 3.38*PS4 (Limiting)

The linearity of these relationships has two causes, the characteristics of the unification, backtracking and data structure checking (for negation) routines themselves, and the nature of the tasks they are required to perform by the individual computations. Taking the latter cause, the five Prolog program applications used in these experiments each requests a series of Support operations in which the precise mixture of operations varies with the data, but the complexity of the individual operations does <u>not</u> vary with the data; for example, in the naive

reverse program the unification which occurs at entry to the non-base case of the append predicate forms a larger proportion of all unifications as the list to be reversed grows longer, but the complexity of the unification is the same in every instance of execution of that case. Given these facts concerning the demands made of the Support routines by the particular applications, it is clear that the experimental results strongly suggest that the routines have a linear performance in implementing the required operations.

However, this simple linearity will not be observed in the case of a Prolog program in which, for example, the complexity of a unification operation depends on the data. The following program has this property:

```
query(l1,l2) ← check(l1,l2)
check(l,l) ←
```

The lists supplied as data to the query (one unification for receiving the data) are passed to the check predicate where the second (and final) unification must scan the <u>entirety</u> of the lists to ensure equality. Hence, for a contribution of exactly 1 to the PS4 statistic, the check unification makes a contribution to the PS5 statistic which is dependent on the data; PS4 will always be 2, but PS5 can be varied at will by changing the lengths of l1 and l2.

Conclusions on the performance characteristics of the Support software must be stated carefully. Although for a number of Prolog applications (probably very many) the Support implements a linear relationship between PS4 and PS5, in general this is not true. It appears that

the Support implements a linear relationship between the complexity of the operations (unification, backtracking and structure checking) requested and PS5; perhaps some representation of this complexity would have been a better choice for the PS4 statistic.

Figure 11

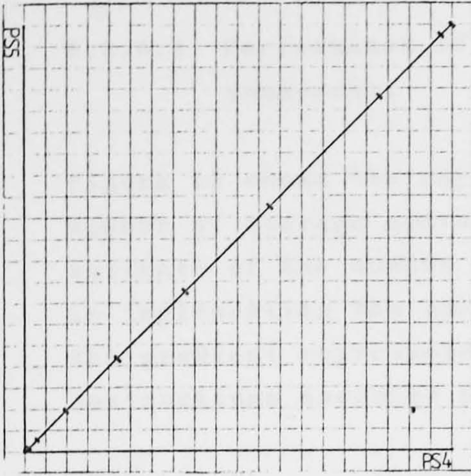


Table 1

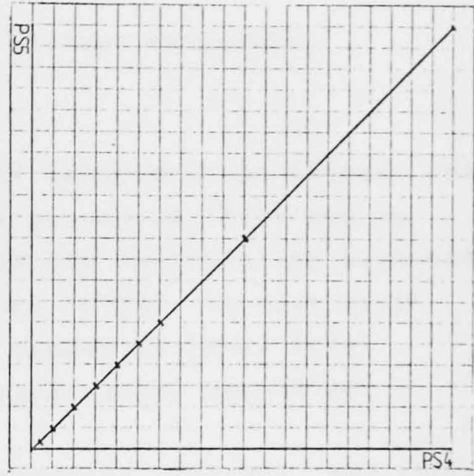


Table 2

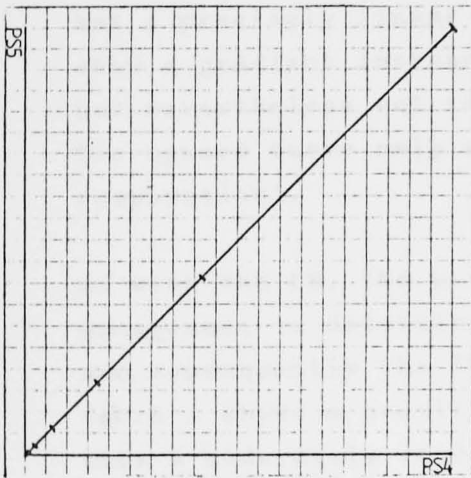


Table 3

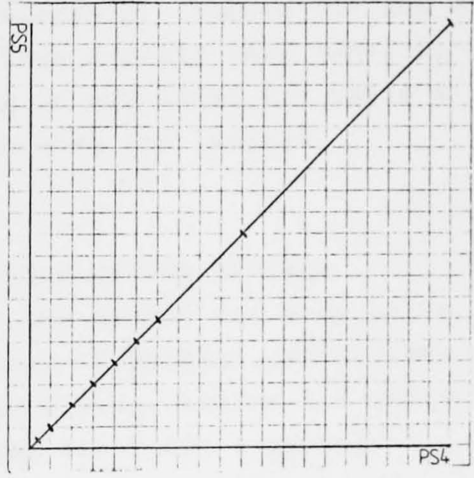


Table 4

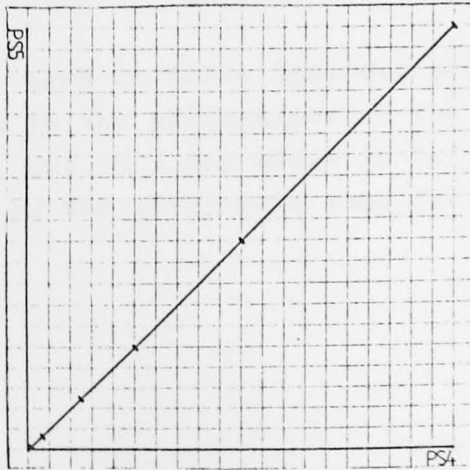


Table 5

6.2.6.6 Performance of the Storage management component.

Figure 12 shows the empirical relationships between the number of Storage operations processed (PS6) and an estimate of the number of IBM 370 instructions executed in implementing the operations (PS7). In each graph the gradient represents the number of lower level instructions executed per higher level operation.

Table 1 shows a steadily increasing gradient, and curves dramatically upwards at the longest executions. Table 2 has a precisely linear relationship. Tables 3,4 and 5 show a gradient increasing less rapidly than Table 1, but nevertheless not tending to any limit (in Table 4 the upward curve only starts with the longest computation).

As with the LM, the performance of the Storage management is determined by the demand for heap cells and consequently the frequency of garbage collections. Table 1 shows a heavily loaded heap, Tables 3,4 and 5 show a moderately loaded heap (in Table 4 garbage collection occurs only in the largest computation), and Table 2 shows no garbage collections at all.

The routine used to force indirect pointers in the heap contributes to the performance of the Storage management, but in a way which is not, in these cases, determined by the query data supplied to the programs.

These results show that the Storage management component of the PM has an unusual, certainly non-linear, performance characteristic, but there is clearly inadequate information to form a complete analysis. More careful experimentation would be necessary to enable such an analysis.

Figure 12

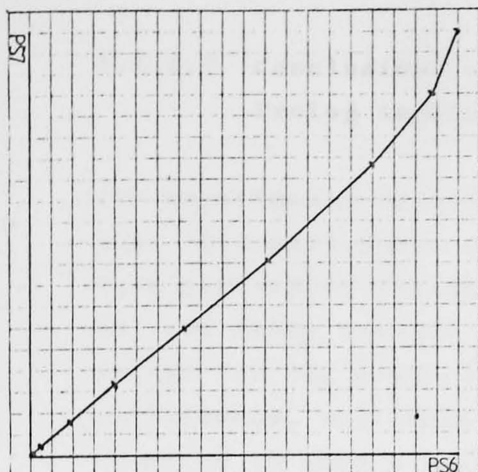


Table 1

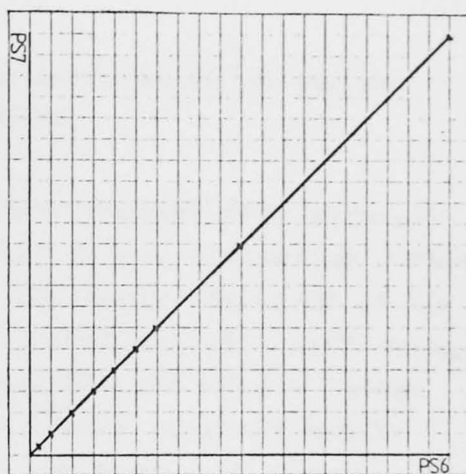


Table 2

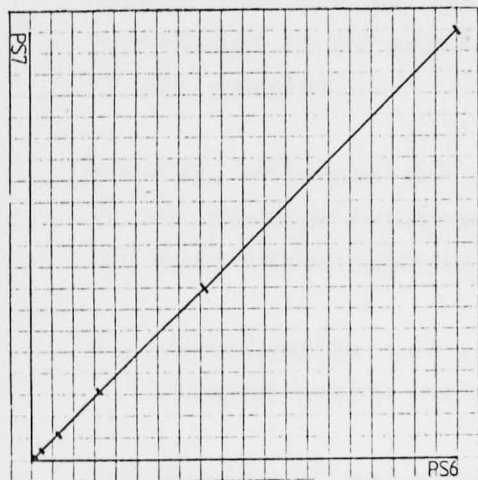


Table 3

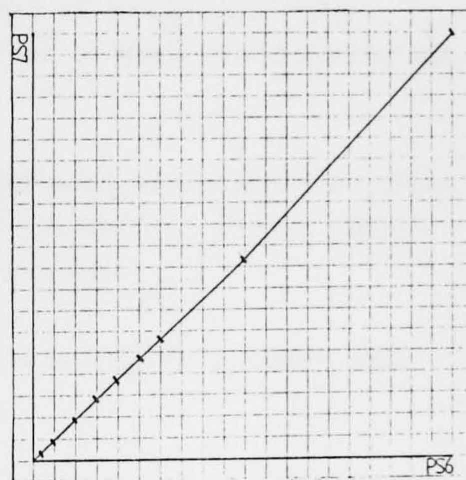


Table 4

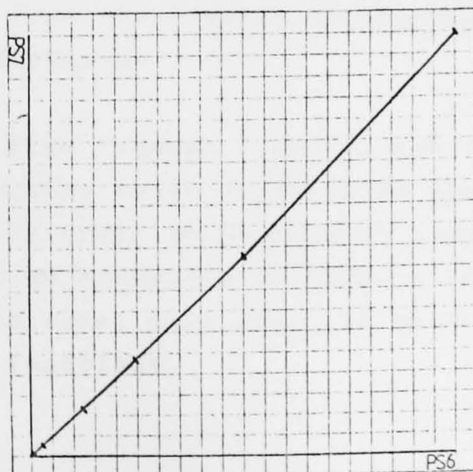


Table 5

6.2.6.7 Conclusions on the performance of the PM Prolog implementation.

The experiments reported above suggest very strongly that this particular compilation and evaluation strategy, this particular design for the Apply component of the PM, and (with certain qualifications) this particular design for the Support component each have a linear performance, and hence that Prolog programs may be executed in such a way that the number of Storage operations and IBM 370 instructions required is a linear function of the number of predicate case executions to be expected from a model interpretation of the program.

However, evidence for the performance characteristics of the Storage management component is inconclusive, and more investigation is required.

6.3 Conclusions and comparison of the Lispkit and Prolog pseudo-machines.

The results of experiments reported in this chapter enable a comparison of the relative performance to be expected from the Lispkit and Prolog implementations under consideration.

The results from Figures 1 and 7 show that about twice as many predicate cases as function applications must be executed in order to accomplish simple computations; quicksort is an exception to this as the Prolog version uses one predicate, partition, to perform the work of two Lispkit functions.

The results from Figures 2 and 8 then show that approximately the same number of pseudo-machine instructions (about 20) must be executed per function application as per predicate case.

Examination of the AlgolW coding for the pseudo-machines in Appendices B and D reveals that the average amount of code to be executed for PM instructions is a little more than for LM instructions.

The similarity of "programming technology" used in both Apply components means that a direct comparison will yield a useful result. From these simple observations it seems that the PM Apply component will do a little more than twice the work of the LM Apply component for a similar computation, although of course the flexibility of Prolog may enable a more subtle program (as for quicksort, mentioned above) and the PM Apply component may do less work than the LM Apply component.

However, this comparison only remains true while both Apply components make similar use of similar underlying virtual machines to execute lower level operations. The PM contains supporting software for unification, backtracking and data structure checking which is entirely absent in the LM; this provides a considerable processing overhead. Both the LM and PM execute with heap storage management, but the PM places a greater load on its heap than does the LM; this also provides a greater processing overhead for the PM.

Thus overall it seems that the doubling of workload (and hence halving of speed on real hardware) when moving from a Lispkit program to a Prolog program is a very best case, and in general a much worse degradation of performance should be expected. However, if we ignore the contribution of storage management to the overall performance of the pseudo-machines, then the experimental results have shown that Prolog programs can be expected to execute no worse than a linear factor slower than Lispkit programs. Hence if future research in the field of novel machine architectures can provide a computer in which heap management is a hardware function which occurs concurrently with program executions, then there is a good probability that not only will Lispkit and Prolog be able to execute much more efficiently than at present, but that the powerful logic programming style will be no worse than a simple linear factor slower than functional programming.

CHAPTER 7 HIGHER LEVEL INTERPRETATION OF LISPKIT AND PROLOG.

Higher level interpretation of Lispkit and Prolog.

In this context I am using the phrase "higher level interpretation" to imply that Lispkit and Prolog programs (in source code form) are being executed by some interpreter which is itself written in a very high level language; the interpreter is then executing on some special machine or virtual machine. Thus the overall system structure in which I am interested is

Lispkit or Prolog program and data
Interpreter in very high language X
(Virtual)machine to execute language X

7.1 Interpreting Lispkit.

For the specific cases which I shall treat in this chapter, the interpreter itself will be a Lispkit program executing (in compiled form) on the Lispkit pseudo-machine. The specific system structure will be

I1	Lispkit program and data
I2	Interpreter
	Lispkit pseudo-machine(LM)
	IBM 370

in which the internal structure of the pseudo-machine has been ignored, as it is of no interest here. Of course the Lispkit interpreter could equally well be executing on a virtual machine constructed from several levels of source code interpreters, and this should not affect our assessment of the performance of the one interpreter of interest.

When analysing the performance of systems with the above configurations, the methodology of Chapter 5 is particularly relevant. The Lispkit program algorithm, the interpreter, and the LM will all have their own performance characteristics, and simple minded monitoring of the IBM 370 (by timing, for example) will not necessarily enable the component of performance due to the interpreter alone to be distinguished.

The characteristic of interest in a performance assessment will be the relationship between the number of expressions (or, typically, function applications) which the Lispkit program expects to be evaluated at I1, and the number of expressions which the interpreter executes at I2. The former statistic must be derived from a model interpretation of the program, in order that it is independent of the interpreter, and that any bad behaviour within the scheme of evaluation embodied in the interpreter is not incorrectly attributed to the program. The way that this is achieved is made explicit in later sections.

Two Lispkit interpreters will be covered, as representatives of an open ended family of such interpreters. The interpreters are presented in the order in which I examined them - the undesirable performance characteristics of the first, as uncovered by experimental assessment, led directly to modifications yielding the second, more efficient interpreter.

7.1.1 A preliminary note on the Lispkit program syntax required by the interpreters.

I have mentioned elsewhere that in practice Lispkit programs are presented to the computer with an

s-expression syntax, rather than with the more palatable notation of Chapter 2. This is consistent with the requirement that the data (this includes program text) supplied to an interpreter written in Lispkit must be in the form of s-expressions. Hence the operations within the interpreters which extract the syntactic components of Lispkit expressions will be compositions of car and cdr selectors, and will be determined by the precise syntax of the programs. The s-expression syntax of Lispkit is given in Appendix B.

Throughout the discussion of the interpreters I shall avoid the use of car and cdr as sub-expression selectors wherever possible. Instead I shall use the following mnemonic selector functions, defined here in terms of car and cdr. An expression is atomic only if it is a variable, and no selectors are required for this case. All other expressions have the form of a list in which the first element indicates the type of expression, and successive elements are operands:

rator(e)	= car(e)	Select operator
rand1(e)	= car(cdr(e))	Select first operand
rand2(e)	= car(cdr(cdr(e)))	Select second operand
rand3(e)	= car(cdr(cdr(cdr(e))))	Select third operand (conditional expressions only)

with several special purpose selectors:

argsandbody(e)	= cdr(e)	Of a lambda expression
arglist (e)	= cdr(e)	Of a function application
qualified(e)	= car(cdr(e))	Of a <u>where</u> or <u>whererec</u> expression
definitions(e)	= cdr(cdr(e))	Of a <u>where</u> or <u>whererec</u> expression

```
defvar(d)   =  car(d)     Select variable from a definition
defexp(d)   =  cdr(d)     Select defining expression from
                                  a definition.
```

7.1.2 First Lispkit interpreter for Lispkit, LISPINT1.

7.1.2.1 The interpreter program.

At its outermost level LISPINT1 is a function of two arguments, a Lispkit program text and a list of arguments respectively. The program text must be an expression whose value is a function, and hence the main structure of LISPINT1 is

λ (fn,args)"evaluate fn and apply it to args"
<u>whererec</u> "auxiliary functions required for evaluation".

The most important auxiliary function is eval, which accepts a Lispkit expression and an environment of variable names with associated values, and returns the value of the expression in the given environment. Eval is simply a case analysis of the possible expression types, and it calls itself recursively as necessary for the evaluation of subexpressions; in particular, recursive calls within <u>where</u> expressions, <u>whererec</u> expressions and function applications are supplied environments which have been extended with new definitions.

The general evaluation strategy can be illustrated by a few selected cases from within eval (given in its entirety later). If e is the expression to be evaluated and n and v contain the current environment then:

Fetching the value of a variable from the environment (assoc to be defined later):

<u>if</u> atom(e) <u>then</u> assoc(e,n,v) <u>else</u> ...

A typical binary operator:

```
    if eq(rator(e),ADD) then  eval(rand1(e),n,v)+eval(rand2(e),
                                                          n,v)
    else  ...
```

Conditional expressions:

```
    if eq(rator(e),IF) then
          if eval(rand1(e),n,v) then eval(rand2(e),n,v)
                                else eval(rand3(e),n,v)
       else ...
```

Building a closure to represent a function value:

```
  if eq(rator(e),LAMBDA) then cons(argsandbody(e),cons(n,v))
  else  ...
```

Before proceeding to describe the expressions which extend the environment, it is necessary to give the structure of the environment itself. From the example cases of eval, above, it is apparent that the environment consists of two parts, named n and v. The s-expression n records the names of the variables whose values have been entered in the s-expression v. The fact that the environment is extended by lists of simultaneous definitions is reflected by the structure of n; n is a list of lists of variable names, for example

```
    ((X Y Z) (A B C) (X Y) ...)
```

Each sublist corresponds to one group of definitions, and sublists nearest to the head of n correspond to inner scopes.

The values entered in v follow exactly the same pattern as the corresponding names in n. However there is a very important difference; the true values of the

variables are not recorded, but instead each sublist of values is represented by a closure (function) which, when applied to an empty parameter list, returns the actual values of the variables in the sublist. The evaluation of each group of defining expressions has been *delayed*, and must be *forced* when access is required. So, for the name list above the value list will be represented by

(λ()xyz   λ()abc   λ()xy ...)

where the body xyz evaluates to give a 3-list of values for X,Y and Z, and similarly for abc and xy.

From these descriptions of n and v the assoc function for looking up a variable's value can be defined:

```
assoc(x,n,v) = if member(x,head(n)) then locate(x,head(n),head(v)())
                                    else assoc(x,tail(n),tail(v)
member(x,l) = if eq(l,NIL) then F else
              if eq(x,head(l)) then T else member(x,tail(l))
locate(x,n,v) = if eq(x,head(n)) then head(v)
                                 else locate(x,tail(n),tail(v))
```

In the first line of assoc the expression "head(v)()" forces the delayed sublist of definitions by applying the closure to an empty parameter list.

This treatment of the environment may seem strange, but it has good justification, which will become apparent in the description of the evaluation of *whererec* expressions below.

With this structure for an environment the evaluation of a where expression is quite straightforward, and function applications are only slightly more involved. To evaluate a where expression (keyword LET) the qualified expression is simply evaluated in an environment extended by the qualifying definitions; the names in the definitions are added to n, and the evaluation of the list of defining expressions (in the current environment by evlis) is delayed and the closure is added to v:

```
    if eq(rator(e),LET) then
          (eval(qualified(e),cons(newnames,n),cons(newdefns,v))
           where newnames = vars(definitions(e))
                 newdefns = λ()evlis(exprs(definitions(e)),n,v))
   else ...
```

where the three auxiliary functions are defined as

```
   vars(deflist) = if eq(deflist,NIL) then NIL
                   else cons(defvar(head(deflist)),
                             vars(tail(deflist)))
   exprs(deflist) = if eq(deflist,NIL) then NIL
                    else cons(defexp (head(deflist)),
                              exprs(tail(deflist)))
   evlis(explist,n,v) = if eq( explist,NIL) then  NIL
                        else cons(eval(head(explist),n,v),
                                  evlis(tail(explist)n,v))
```

To evaluate a function application the rator field of the expression must be evaluated to obtain a closure, which contains a qualified expression to be evaluated and a list of variable names which are to be associated with the actual argument values. This is the default case for the eval case analysis, and so e is known to be a function application:

```
eval(body(fn), cons(formalargs(fn),oldn(fn)),
               cons(actualargs,oldv(fn)))
where fn = eval(rator(e),n,v)
      actualargs = λ()evlis(arglist(e),n,v)
```

where the extra selector functions may be defined as

```
formalargs(clos) = car(car(clos))
body(clos) = car(cdr(car(clos)))
oldn(clos) = car(cdr(clos))
oldv(clos) = cdr(cdr(clos))
```

Now to tackle the evaluation of a <u>whererec</u> expression (keyword LETREC). This is closely related to the evaluation of a <u>where</u> expression, but there is a new problem because the new definitions must be evaluated <u>not in the current environment</u>, but in the <u>extended environment</u> (which is only available for access when the definitions have been completed). In other words, what we would like to write is

```
newdefns = λ()evlis(exprs(definitions(e)),
              cons(newnames,n), cons(newdefns,v))
```

in which newdefns is defined in terms of itself. In fact this has precisely the desired effect due to the technique of <u>delaying</u> the evaluation of the defining expressions until they are accessed; evlis does not attempt to access newdefns until newdefns has certainly been associated with the delayed environment level, and then, provided that the restriction on <u>whererec</u> defining expressions (see Chapter 2) has been obeyed, evlis will not invoke a nonterminating recursive forcing of newdefns (recursion is allowed only in defining expressions which are themselves lambda expressions, which naturally delay the evaluation of their bodies

until required). With a slight modification to the above expression, the complete evaluation of a wehererec expression is:

```
    if eq(rator(e),LETREC) then
       (eval(qualified(e), newn,cons(newdefns,v))
        whererec newn = cons(vars(definitions(e)),n)
                 newdefns=  λ()evlis(exprs(definitions(e)),
                                 newn,cons(newdefns,v)))
   else ...
```

All that remains to complete the interpreter is to give eval in its entirety, and to show how the evaluation is initiated from the main arguments fn and args.

To initiate the evaluation fn must be evaluated in an empty environment to obtain a closure:

```
    clos = eval(fn,NIL,NIL)
```

and this must be applied to args in exactly the same way as a normal function application (note that args themselves do not need to be evaluated but they do need delaying!):

```
   λ(fn,args)(eval(body(clos), cons(formalargs(clos),
                                       oldn(clos)),
                                   cons( λ()args,oldv(clos)))
              where clos = eval(fn,NIL,NIL))
```

and the complete definition of eval is

```
eval(e,n,v) = if atom(e)                then assoc(e,n,v) else
              if eq(rator(e),QUOTE)     then rand1(e) else
              if eq(rator(e),CAR)       then car(eval(rand1(e),n,v)) else
              if eq(rator(e),CDR)       then cdr(eval(rand1(e),n,v)) else
              if eq(rator(e),CONS)      then cons(eval(rand1(e),n,v),eval(rand2(e),n,v)) else
              if eq(rator(e),ATOM)      then atom(eval(rand1(e),n,v)) else
              if eq(rator(e),EQ)        then eq(eval(rand1(e),n,v),eval(rand2(e),n,v)) else
              if eq(rator(e),LEQ)       then eval(rand1(e),n,v) ≤ eval(rand2(e),n,v) else
              if eq(rator(e),IF)        then if eval(rand1(e),n,v) then eval(rand2(e),n,v)
                                                                   else eval(rand3(e),n,v) else
              if eq(rator(e),LAMBDA)    then cons(argsandbody(e),cons(n,v)) else
              if isarithop(rator(e))    then arith(e,n,v) else
              if eq(rator(e),LET)       then ( eval(qualified(e),cons(newnames,n),cons(newdefns,v))
                                               where newnames = vars(definitions(e))
                                                 and newdefns = λ()evlis(exprs(definitions(e)),n,v)  ) else
              if eq(rator(e),LETREC)    then ( eval(qualified(e),newn,cons(newdefns,v))
                                               whererec     newn = cons(vars(definitions(e)),n)
                                                    and newdefns = λ()evlis(exprs(definitions(e)),
                                                                     newn,cons(newdefns,v)) )
              else ( eval(body(fn),cons(formalargs(fn),oldn(fn)),cons(actualargs,oldv(fn)))
                     where          fn = eval(rator(e),n,v)
                       and actualargs = λ()evlis(arglist(e),n,v) )


isarithop(op) = if eq(op,ADD) then T else if eq(op,SUB) then T else
                if eq(op,MUL) then T else if eq(op,DIV) then T else
                if eq(op,REM) then T else F


arith(e,n,v) = if eq(op,ADD) then   (a1+a2)   else if eq(op,SUB) then   (a1-a2)   else
               if eq(op,MUL) then   (a1*a2)   else if eq(op,DIV) then (a1 div a2) else
               if eq(op,REM) then (a1 rem a2) else ERROR
               where op = rator(e)
                 and a1 = rand1(e)
                 and a2 = rand2(e)
```

Two final comments on the LISPINT1 program are appropriate. Firstly, a more conventional approach to evaluating wherērec expressions is to extend the purely functional Lispkit language with an extra primitive operator "rplaca" ("replace car") which appears to be an identity function, but which actually has the side effect of modifying an s-expression (in this case to tie a self-referential loop in the environment). Whererec is then evaluated by essentially the same controlled trick as is implemented by the DUM and RAP instructions of the LM(Appendix B, Henderson (1980)). I have chosen to avoid this, and to look at the expression and performance of interpreters in a purely functional language.

Secondly, the interpreter is not so revealing about the semantics of Lispkit as, perhaps, it could be; this is due to the somewhat circular definition in which each Lispkit expression is evaluated by calling on an expression of exactly the same type in the interpreter. Nevertheless, the interpreter does seem quite interesting, and it is certainly illustrative of performance assessment problems, as covered in the next section.

7.1.2.2 The performance of LISPINT1.

The important question to ask about LISPINT1 (in addition to "Does it work correctly?") is "How efficiently does it interpret Lispkit programs?" This question can be phrased slightly more precisely as "How does the number of expression evaluations required by LISPINT1 depend on the number of expression evaluations required by the program which LISPINT1 is interpreting?" However,

the characteristics of interest must be specified even more tightly before experimental evidence can be gathered. The methodology of Chapter 5 will again be the guideline.

Experimental assessment of the performance of LISPINT1 will be made in the following system configuration:

	Data
I1	Lispkit (source) program
I2	LISPINT1 (compiled)
	LM
	IBM 370/168

in which I have ignored the contribution of MTS, and internal detail of the LM has been suppressed, as statistics will not be required from interfaces below I2.

The component of interest is LISPINT1, and hence it is the interactions at I1 and I2 which must be monitored. At I2 the LM is interpreting the machine instructions of LISPINT1, and at I1 the eval function is scanning and interpreting the constructs of the Lispkit program.

At I2 the valuable statistic is the amount of work (expression evaluations) generated by LISPINT1, as determined by a model interpretation of LISPINT1. This statistic, call it LIS2, will be independent of the virtual machine below I2.

At I1 the valuable statistic, call it LIS1, is the amount of work (expression evaluations) generated by the Lispkit program, as determined by a model interpretation of the program.

Hence the performance assessment question can be rephrased as "How does the number of expression evaluations predicted by a model interpretation of LISPINT1 depend on the number of expression evaluations predicted by a model interpretation of the Lispkit program?"

Recalling the results of Chapter 6, a good representation of the LIS2 statistic can be obtained by monitoring the number of LM instructions executed during the computation. These figures are easily obtained - and more efficiently than by reprogramming LISPINT1 to monitor its own activity.

Similarly, LIS1 statistics can be obtained quite effectively by executing the Lispkit program in question (plus data) directly on the LM and monitoring the total number of LM instructions executed. This technique is justified by the fact that the model interpretation is determined solely by the programming language semantics and not by particular interpreters.

Note that the actual interpreter code used in the assessment is not exactly as the eval function has appeared here; the selector functions are all expanded "in line", and there are other minor syntactic variations, none of which alter the evaluation strategy implemented by the interpreter.

7.1.2.3 Experimental results.

In an ideal world LISPINT1 should have its behaviour monitored for a wide range of program applications to

obtain an accurate assessment of its performance; the selection of programs used to assess the LM would be a good choice. However, the comparative inefficiency of LISPINT1 (when loaded onto the LM and IBM 370) restricts the size of computation which it can execute in a reasonable time; this renders impractical the more complex programs, such as quicksort and powering,and the longer executions of the simpler programs. This restriction is not too serious, as the general trends of each program were seen to be very similar in the assessment of the LM, and a good impression of the characteristics of LISPINT1 should be possible by examining just a few applications.

Test executions of LISPINT1 have been performed, interpreting the naive reverse program for lists of length 0,1,2,3, and interpreting the reverse program with accumulating parameter for lists of length 0,1,2, 3,4,5,6. Naive reverse for a list of length 4 was interrupted after 200 seconds of CPU time without having found a solution, so no further data points are practically possible for this algorithm. Reverse with accumulating parameter for list length 6 used approximately 60 CPU seconds; I expected list length 7 to use several hundred CPU seconds, so data collection was stopped at this point.

The collected LIS1 and LIS2 statistics are tabulated in Appendix E (Tables 7,8), and their relationships with respect to the performance of LISPINT1 are analysed in the following graphs. As in Chapter 6 the graphs

are presented with labelled but ungraduated axes, and with the empirical data points joined by straight line segments. The purpose of the graphs is to give a visual guide to the trends discussed in the comments; the precise data is to be found in Appendix E.

Figure 1 shows the results of executing the naive reverse program on LISPINT1. The upper graph shows the inherent performance of the reverse algorithm as the relationship between the list length and the number of LM instructions that the program would execute on the pseudo-machine (LIS1). The lower graph shows the performance of the interpreter as the relationship between LIS1 and the number of instructions that the interpreter executes on the LM(LIS2).

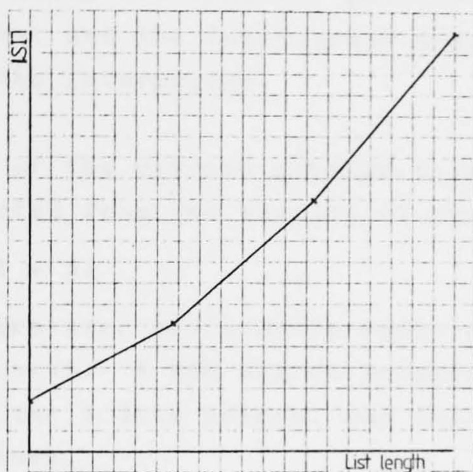
The inherent performance is, as expected, quadratic in form. The data points fit the following equation precisely:

$$LIS1 = 8.5*l*l + 19.5*l + 19.$$

On the other hand the relationship between LIS1 and LIS2 shows a very dramatic upward curvature. Since only 4 data points are available, little can be deduced about the true nature of the relationship. However, the curve is certainly increasing more rapidly than a quadratic function as a cubic function is required to fit the points (but a cubic will fit any 4 points, and hence the latter is not by itself a useful observation).

Bearing in mind that the lower graph contains no contribution from the performance of either the reverse program or the LM, the performance of LISPINT1 clearly is not linear.

Figure 1 (Table 7)



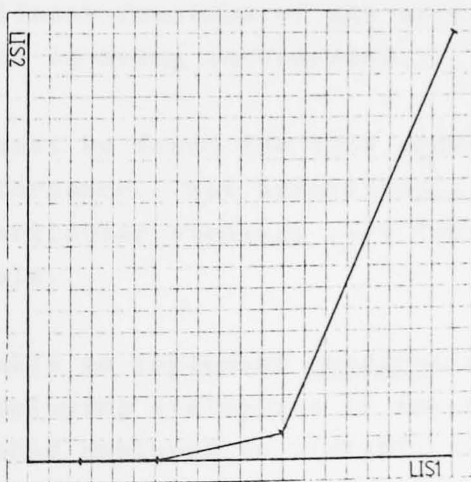


Figure 2 shows the results of executing the reverse with accumulating parameter program on LISPINT1. The upper graph shows the inherent performance of the reverse algorithm, and the lower graph shows the performance of the interpreter.

The performance of the reverse program is precisely linear, as expected from the results of Chapter 6:

LIS1 = 17*l + 32

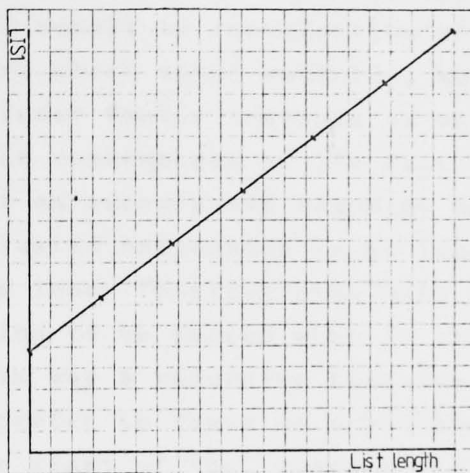
However the interpreter performance again shows a rapidly rising curve. A 6th degree polynomial is required to fit the 7 data points, but that is not conclusive evidence.

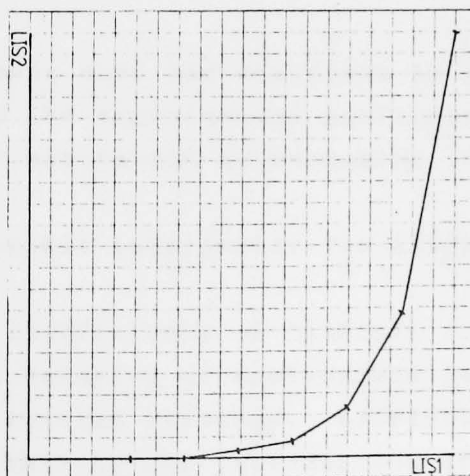
These results from executing the two list reversing programs on LISPINT1 point out very clearly that the interpreter has an extremely bad performance characteristic; the work performed by the interpreter increases much more rapidly than a linear function of the work demanded by the program which is being interpreted.

An explanation must be found for this undesirably inefficient performance. Not surprisingly the answer lies with the particular strategy used for delaying environment levels.

In order to be able to implement <u>whererec</u> successfully some method of delaying evaluation is appropriate, but although the method employed in LISPINT1 seemed natural and correct, it turns out to be rather less than desirable.

Figure 2 (Table 8)





The method employed effectively implements a "call by name" evaluation strategy in which the arguments in a function application are passed in unevaluated form to the function body; within that body each access to an argument will result in reevaluation of the argument's value, which seems wasteful, but it leads to at worst a linear factor increase in workload. A much more serious consequence of the call by name strategy is that an unevaluated argument may be passed within an unevaluated expression to an inner function application; the inner function body will cause both levels of delaying to be forced when it accessess the argument; in this way a recursive function will often cause the interpreter to trace an arbitrary distance back towards the start of the computation each time an argument is accessed. The sublist delaying strategy aggravates this problem, as a whole tree of unnecessary auxiliary values may be computed each time an argument is accessed.

I seem to have made some bad decisions in the design of LISPINT1, and the experimental performance analysis has provided the motivation to reexamine the design.

7.1.3 Second Lispkit interpreter for Lispkit, LISPINT2.

The undesirable inefficiency of LISPINT1 is caused by the way in which the delaying of environment levels led to a call by name evaluation strategy. LISPINT2 is an attempt to improve upon this by the introduction of a "call by value" evaluation strategy.

7.1.3.1 The Interpreter program.

LISPINT2 is essentially the same interpreter as LISPINT1, the only difference being in the treatment of delayed environment levels. The environment, as given by the n and v arguments of eval, will have exactly the same structure and properties as for LISPINT1; each level in v will require forcing before the values can be accessed, and the assoc function will remain as before. However, in order to avoid the call by name mechanism the definitions which comprise each level are evaluated <u>before</u> being grouped together and delayed, to yield a call by value mechanism; this is a fairly obvious change of strategy, involving only minor changes to the interpreter code, which is quite clearly correct in the case of function applications and <u>where</u> expressions, but requires a little more thought in the case of <u>whererec</u> before operational safety is apparent.

Tackling <u>where</u> expressions first, here is the interpretation in LISPINT1 for comparison:

```
if eq(rator(e),LET) then
   (eval(qualified(e),cons(newnames,n),cons(newdefns,v))
    where newnames = vars(definitions(e))
          newdefns = λ()evlis(exprs(definitions(e)),n,v))
else ...
```

In LISPINT2 this is modified simply by moving the "λ()":

```
if eq(rator(e),LET) then
    (eval(qualified(e),cons(newnames,n),cons( λ()newdefns,v))
    where newnames = vars(definitions(e))
          newdefns = evlis(exprs(definitions(e)),n,v))
else ...
```

Appealing to the semantic background of Lispkit these two interpreter fragments are clearly equivalent. Operationally, the LISPINT2 fragment has introduced no new recursions, and will be safe to execute provided that the definitions to be evaluated are safe (they do not contain non-terminating recursions).

The interpretation of function applications is changed in the same way:

```
    eval(body(fn),cons(formalargs(fn),oldn(fn)),
                  cons(λ() actualargs,oldv (fn)))

   where fn = eval(rator(e),n,v)
         actualargs = evlis(arglist(e),n,v)
```

The call by value parameter mechanism is explicit here.

An extra change must be made in the whererec case in order not to violate the variable usage rules:

```
    if eq(rator(e),LETREC) then
       ((eval(qualified(e),newn,cons(λ()newdefns,v))
         whererec newdefns = evlis(exprs(definitions(e)),
                                   newn,cons(λ()newdefns,v)))

       where newn = cons(vars(definitions(e)),n))

       else ...
```

As in the previous two cases the "λ()" has been moved in such a way as to preserve the meaning of the interpreter fragment, though it has now appeared in two places - at both occurrences of newdefns. The definition of newn has been moved to an enclosing scope, to satisfy the restriction on whererec definitions (as it is used in the evaluation of the defining expression for newdefns), and as a slight economy the cons(...,n)

has been taken into the definition. The validity and safety of the definition of newdefns must be considered carefully; the defining expression itself mentions newdefns, and both requirements will be satisfied if the expression evaluations invoked by evlis never attempt to access newdefns (which after all, is at the head of the environment in which the evaluations occur) - of course evaluating the qualified expression may attempt access to newdefns, but that is operationally safe. The first point to note is that the reference to newdefns is delayed:

```
evlis(exprs(definitions(e)),newn,cons(λ()newdefns,v))
```

and so attempted access to newdefns will only occur if any of the evaluations of exprs (definitions(e)) call assoc to look up a variable whose value is contained in newdefns (call on assoc to look up variables whose values are in v are perfectly safe). Recalling that we are considering the interpretation of a <u>whererec</u> expression, such a call of assoc will occur only if the evaluation of one of the defining expressions requires the values of one of the locally defined variables. This occurrence is precisely what the restriction on defining expressions in <u>whererecs</u> disallows. Hence the interpreter fragment is valid and safe if the exprs(definitions(e)) consist only of expressions which delay the variable references that they contain (for example, lambda expressions defining recursive functions), and other expressions which refer only to variables in outer scopes.

This discussion has shown, in a somewhat paradoxical fashion, how the restriction on variable usage in *whererec* expressions arises. The circularity appears in interpreting *whererec* expressions by using *whererec*, but hopefully the resolution of the problem will have served to reinforce the understanding of the limitations of *whererec*. The restriction is appropriate in the case of programs compiled to execute directly on the LM for analogous reasons.

7.1.3.2 The performance of LISPINT2.

The same reasoning applies here as in the assessment of LISPINT1.

Experimental assessment of the performance of LISPINT2 will be made in the system configuration:

	Data
I3	Lispkit (source) program
I4	LISPINT2 (compiled)
	LM
	IBM 370/168

where again the internal structure of the LM has been suppressed, and the contribution of MTS has been ignored.

The enquiry about the performance of LISPINT2 is phrased as "How does the number of expression evaluations predicted by a model interpretation of LISPINT2 depend on the number of expression evaluations predicted by a model interpretation of the Lispkit program?"

The statistics collected at interfaces I3 and I4 will be counts of LM instructions executed, LIS3 and LIS4, analogous to LIS1 and LIS2 respectively.

7.1.3.3 Experimental results.

Not surprisingly LISPINT2 is much more efficient than LISPINT1, and test executions have been performed for the same two programs, naive reverse and reverse with accumulating parameter, up to lists of length 50.

Tabulated results for the statistics LIS3 and LIS4 are given in Appendix E (Tables 9,10), and their relationships with respect to the performance of LISPINT2 are analysed in the following graphs. Again the graphs are presented with labelled but ungraduated axes, and joined by straight line segments to show the trends discussed in the accompanying comments.

Figure 3 shows the results of executing the naive reverse program on LISPINT2. The upper graph shows the inherent performance of the reverse algorithm, and the lower graph shows the performance of the interpreter as the relationship between LIS3 and LIS4 statistics.

The inherent performance of the reverse algorithm is precisely quadratic

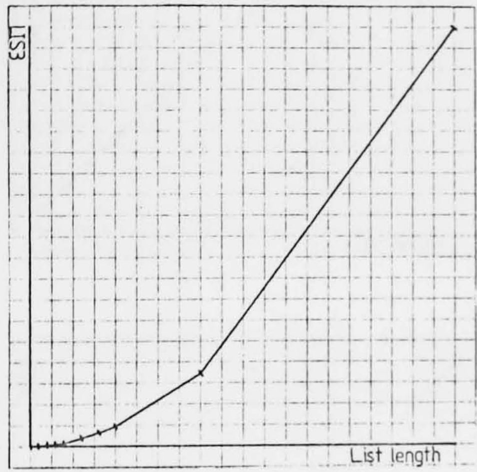
LIS3 = 8.5*l*l+19.5*l+19

The lower graph shows a relationship which has a gradient increasing slowly, apparently to some limit; approximately

LIS4 = 55.8*LIS3

It seems that LISPINT2 has a linear performance.

Figure 3 (Table 9)



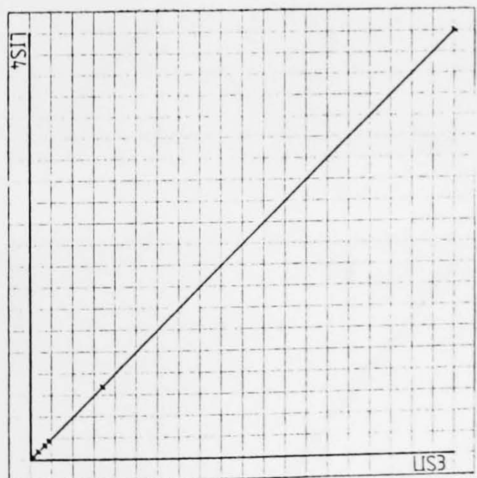
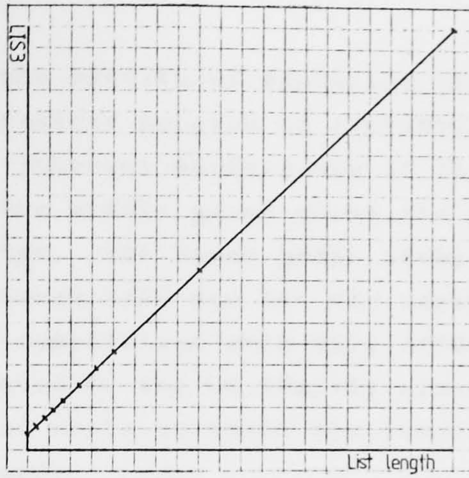
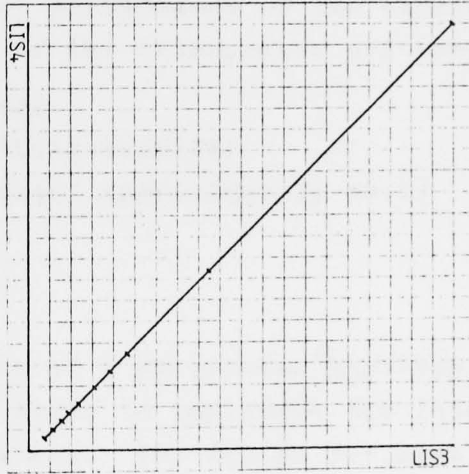


Figure 4 (Table 10)





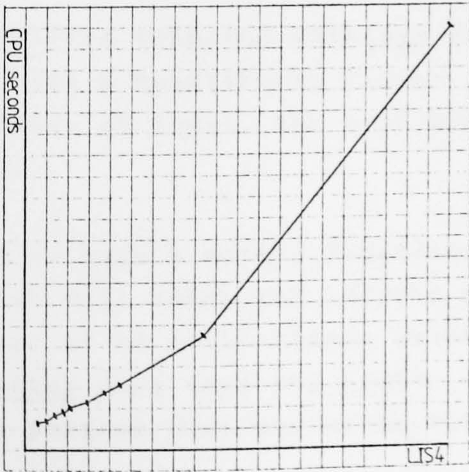


Figure 4 shows the results of executing the reverse with accumulating parameter program on LISPINT2. The upper graph shows the inherent performance of the reverse algorithm, and the middle graph shows the performance of the interpreter. The lower graph shows the relationship between LIS4 and the CPU seconds required to complete the computation when only 5400 heap cells have been allocated to the LM (fairly close to the minimum number of cells in which the longest computation could be performed).

Both the inherent performance of the reverse program and the performance of the interpreter are precisely linear:

LIS3 = 17*l+32

LIS4 = (913*LIS3-4600)/17

(Gradient approximately 53.7)

7.1.4 Comments.

LISPINT1 made use of a particular strategy for delaying environments in order to implement the interpretation of <u>whererec</u> expressions. The design decision led to an interpreter with an unacceptable performance characteristic.

In LISPINT2 a modified delaying strategy is used, and the interpreter exhibits a beautifully linear performance characteristic (for, at least, the test programs used in the experiments).

Thus in LISPINT2 we have a purely functional interpreter for Lispkit, which will interpret Lispkit programs by executing a number of LM instructions which is no worse than a <u>linear factor</u> more than the number of instructions performed if the program were executing directly on the LM.

The linear factor appears to be about 55.

"Rplaca" is usually introduced in order to bring the interpreter performance under control. Assuming that it enables a linear performance interpreter to be produced, then the linear factor would probably be smaller than the 55 observed for LISPINT2. However it would only be a small linear factor better than LISPINT2, and this must be weighed against the introduction of the semantically untidy "rplaca" operation.

To illustrate the problems which might be encountered in a naive attempt to assess the performance of an interpreter consider the lowest graph in Figure 4. The graph increases irregularly, and the overall trend is clearly much worse than linear; this is due to the heavy loading of the heap store of the LM. An experiment which attempted to assess the performance of LISPINT2 by relating, for example, function applications in the interpreted program to the CPU time required for execution, would be unable to isolate the behaviour of the interpreter from that of the LM. The non-linear performance might be incorrectly attributed to LISPINT2 which, as we have seen, is quite innocent.

7.2 Interpreting Prolog.

It would be attractive to embark on an exploration of Prolog interpretation mechanisms, as started for Lispkit in a small way in the first part of this chapter.

However, opportunity permits me to show only one, straightforward, example of a Prolog interpreter, written in Prolog, which has a linear performance characteristic.

The interpreter, PROLOGINT, handles only a restricted form of Prolog - negated conditions and primitive predicates have been omitted both for simplicity and to enable sizeable computations to be performed in a reasonable time. Like the Lispkit interpreters, PROLOGINT makes use of the facilities of Prolog to implement the facilities of Prolog; in particular unknowns in the interpreted computation are represented by unknowns in the interpreter, and unification is implemented implicitly by a generalised equality predicate - more details appear below.

PROLOGINT will be executed on the PM in the following configuration:

Data
Prolog (source) program
PROLOGINT (compiled)
PM
IBM 370/168

In a performance assessment the behaviour will be isolated carefully from that of the PM with its internal storage management. The concern of a performance assessment will be the relationship between the work performed by PROLOGINT and the work required by the Prolog program.

7.2.1 The syntax of interpreted programs.

To be acceptable as data for PROLOGINT, a Prolog program must have a syntax within the framework of su-expressions. Appendix D gives one such syntax, and PROLOGINT accepts programs in this form.

The interpreter will not require the definition of any special selector functions (or predicates), as all parsing is performed by pattern matching and unification, and the significance of a construct will be apparent from the structure of the pattern and the variable mnemonics used.

7.2.2 The interpreter, PROLOGINT.

7.2.2.1 The interpreter program.

PROLOGINT is a query which requires two arguments, a Prolog program (no negated conditions or primitive predicates) and a parenthesised list of the arguments which the program would expect if it were executed directly on the PM. The arguments are checked, to determine whether they satisfy the query represented by the program, by the interpreter predicate satisfy, which has a central role corresponding to eval in the Lispkit interpreters; as a side effect of the checking, the arguments may become elaborated, and the results of the interpretation, if any, are valid elaborations of the unknowns in the original arguments.

The predicate satisfy accepts one argument representing a condition to be checked, and the list of predicate definitions which may be called on by the program:

```
    satisfy((predname.args),deflist)<- ...
```

The query of PROLOGINT initiates the interpretation by explicitly constructing a call on the predicate named QUERY in the program being interpreted, and inserting QUERY into the definitions list:

```
  query(((query.argsandconds)where.deflist),arguments) <-
          satisfy((QUERY.arguments),((QUERY argsandconds)
                                           .deflist))
```

The satisfy predicate follows the actions described in the operational model of Prolog execution in Chapter 3; to interpret a condition the predicate it names must be found, one of the predicate's cases selected, the formal argument list constructed and unified with the actual arguments of the condition, and any extra conditions associated with the case must be executed:

```
  satisfy((predname.actualargs),deflist)<-
    finddef(predname,deflist,cases),selectcase(cases,
                                          (args.conds)),
    localvars((args.conds),locals),argbuild(args,locals,
                                     formalargs),
    unify(actualargs,formalargs),
    satisfyconds(conds,locals,deflist)
```

The auxiliary predicates finddef and selectcase are easily programmed:

```
    finddef(predname,((predname.cases).deflist),cases)<-
    finddef(predname,(other.deflist),cases)<-
                      finddef(predname,deflist,cases)
    selectcase((case.cases),case)<-
    selectcase((other.cases),case)<- selectcase(cases,case)
```

finddef simply looks up the appropriate entry in the list of predicate definitions, and selectcase returns each case in a list as backtracking requires ; selectcase implements backtracking for the Prolog program by backtracking in the interpreter.

satisfyconds is also straightforward; it checks that each condition in a list of conditions is satisfied when the actual arguments are built from a given list of local variable values:

```
  satisfyconds((if.conds),locals,deflist) <-
                              satisfyeach (conds,locals,deflist)
  satisfyconds(NIL,locals,deflist) <-
```

(The second clause expresses the fact that a case with no antecedent conditions is satisfied with no further checking)

```
  satisfyeach(((predname.args).conds),locals,deflist) <-
               argbuild(args,locals,actualargs),
               satisfy((predname.actualargs),deflist),
               satisfyeach(conds,locals,deflist)
  satisfyeach(NIL,locals,deflist) <-
```

In a Prolog program the variables mentioned in each case are purely local to that case; they have their values built into the formal arguments of the consequent of the case and the actual arguments of any conditions present. When a case is entered, by satisfy, the local variables are identified and collected into a local environment list by the predicate localvars. The environment list is a list of pairs (x.v) where x is a variable name and v is its value. When localvars creates an environment list the values are set to new

unknowns, and they are subsequently elaborated by unification. localvars makes use of several additional predicates:

```
    localvars((args if.conds),vars)<-inargs(args,NIL,vars1),
                                     inconds(conds,vars1,vars)
    localvars((args),vars)<- inargs(args,NIL,vars)
    inargs(NIL,vars,vars)<-
    inargs(x,oldvars,newvars)<- atom(x),¬eq(x,NIL),
                                addvar (x,oldvars,newvars)
    inargs(('.x),vars,vars) <-
    inargs((x.y),oldvars,newvars)<- ¬eq(x,'),
                                   inargs(x,oldvars,newvars1),
                                   inargs(y,newvars1,newvars)
    inconds(((predname.args).conds),oldvars,newvars)<-
                  inargs(args,oldvars,newvars1),
                  inconds(conds,newvars1,newvars)
    inconds(NIL,vars,vars)<-
    addvar(x,oldvars,((x.newvar).oldvars))<-
                               ¬member(x,oldvars)
    addvar(x,vars,vars)<- member(x,vars)
    member(x,((x.varx).vars))<-
    member(x,(other.vars))<- member(x,vars)
```

Several comments are appropriate here. The predicates localvars, inargs, and inconds each use an accumulating parameter technique to avoid appending lists of variables. The apostrophe mentioned in inargs has crept in because of the necessity to indicate explicitly the constant parts of su-expression patterns (see Appendix D); also note that the atom NIL is treated specially as a constant and not as a variable. Finally, the variable newvar which appears once in addvar introduces a new unknown into the environment list; the unknown acts as a

placeholder for the value which will eventually be associated with the variable x.

The auxiliary predicate argbuild is used by satisfy to construct formal arguments from formal argument patterns, and by satisfyeach to construct actual arguments from actual argument patterns; variables in the patterns are replaced by their values from the local environment list:

```
argbuild(NIL,locals,NIL)←
argbuild(x,locals;varx) ← atom(x), ¬ eq(x,NIL),
                          findvar(x,locals,varx)
argbuild(('.x),locals,x)←
argbuild((x.y),locals,(buildx.buildy))← ¬eq(x,'),
         argbuild(x,locals,buildx),argbuild(y,locals,buildy)
findvar(x,((x.varx).vars),varx)←
findvar(x,(other.vars),varx)←findvar(x,vars,varx)
```

Finally, unification is defined very simply by a predicate which asserts equality between actual and formal arguments:

```
unify(args,args)←
```

7.2.2.2 The performance of PROLOGINT.

Experimental assessment of PROLOGINT will be made in the following system configuration:

```
              Data
       ______________________
I1     Prolog (source) program
       ______________________
I2     PROLOGINT (compiled)
       ______________________
                PM
       ______________________
            IBM 370/168
```

in which, again, the contributions of MTS and the internal structure of the PM have been ignored.

To assess PROLOGINT the interactions at I1 and I2 must be monitored and compared.

The performance question to be addressed is "How much work does PROLOGINT perform in interpreting the requirements of the Prolog program?" Again adopting the view of a model interpretation, which predicts the amount of work required by a program independently of the virtual machine on which it is executing, the question is rephrased as "How does the work predicted by a model interpretation of PROLOGINT depend on the work predicted by a model interpretation of the Prolog program?"

Recalling the observation, in Chapter 6, that the number of PM instructions executed during a Prolog computation is a good representation of the work predicted by a model interpretation, the interactions at I1 and I2 can be monitored in exactly the same way as for the Lispkit interpreters.

The statistics of I2 are obtained by counting the total number of PM instructions executed during the interpretation; call these statistics PIS2. At I1, the statistics PIS1 are obtained by executing the Prolog program directly on the PM and again counting the total number of PM instructions executed.

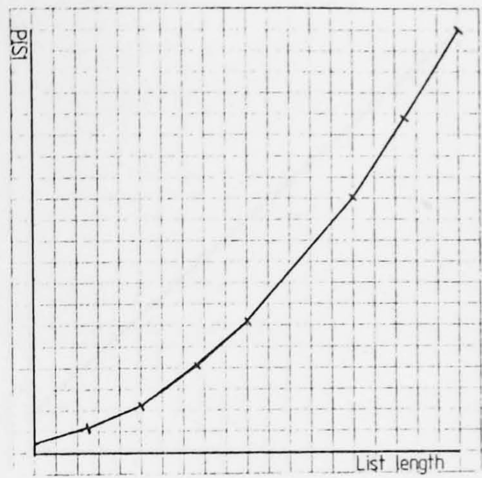
7.2.2.3 Experimental results.

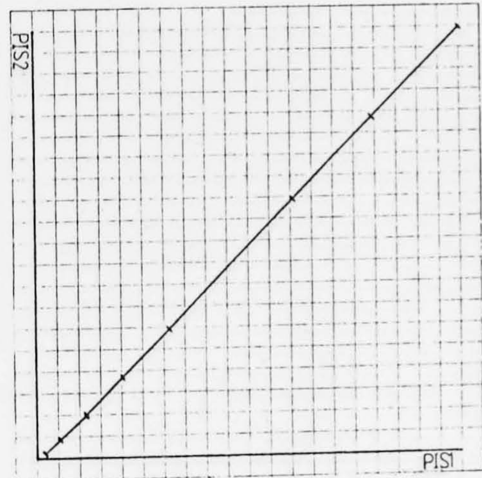
Test executions of PROLOGINT have been performed for the familiar programs naive reverse and reverse with accumulating parameter, over a small range of list lengths between 0 and 10. Statistics PIS1 and PIS2 are collected up to the production of the single result of the interpretation.

The results are tabulated in Appendix F(Tables 6,7), and their relationships with respect to the performance of PROLOGINT are analysed in the following graphs. The graphs are presented with labelled but ungraduated axes, and the empirical points are joined by straight line segments to shows the trends discussed in the accompanying comments.

Figure 5 shows the results of executing the naive reverse program on PROLOGINT. The upper graph shows the inherent performance of the reverse algorithm as the relationship between the list length and the number of PM instructions that the program would execute on the pseudo-machine (PIS1). The middle graph shows the performance of the interpreter as the relationship between PIS1 and the number of instructions that the interpreter executes on the PM (PIS2). The lower graph shows the relationship between PIS2 and the CPU seconds required to complete the computation when 70000 heap cells are allocated to the PM.

Figure 5 (Table 6)





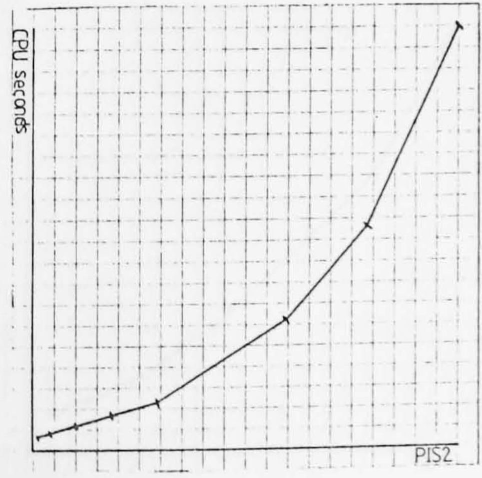
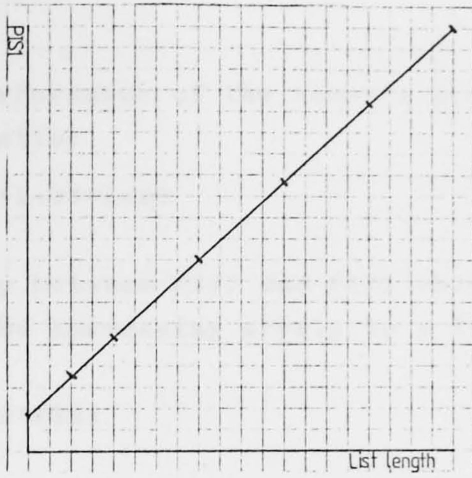
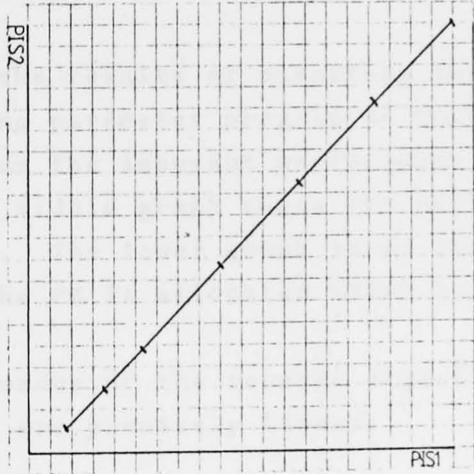
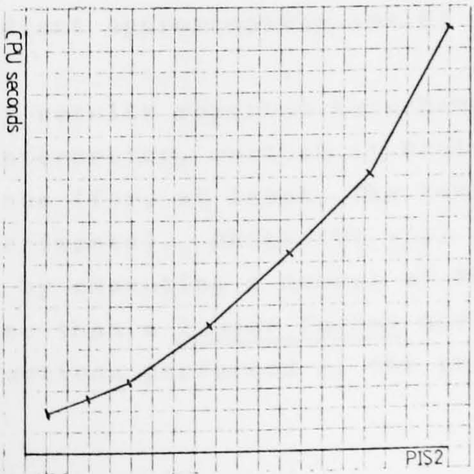


Figure 6 (Table 7)







The inherent performance of the reverse program is precisely quadratic:

PIS1 = 18*l*l+33*l+29

The relationship between PIS1 and PIS2 shows a gradient which seems to be increasing slowly to a limit, approximately

PIS2 = 148.6*PIS1

Hence the performance of PROLOGINT would appear to be linear.

Figure 6 shows the results of executing the reverse with accumulating parameter program on PROLOGINT. The upper graph shows the inherent performance of the reverse algorithm. The middle graph shows the performance of the interpreter. The lower graph shows the CPU seconds required when the PM is allocated 23000 heap cells.

Both the performance of the reverse algorithm and of the interpreter are precisely linear:

PIS1 = 36*l+33

PIS2 = (5567*PIS1-70851)/36

(Gradient approximately 154.6)

The experimental results reported here have shown that the PROLOGINT interpreter, written in Prolog, has a linear performance (for, at least, the test programs used in the experiments). PROLOGINT will interpret Prolog programs by executing a number of PM instructions which is no worse than a linear factor more than the number of instructions performed if the program were

executing directly on the PM. The linear factor appears to be about 150.

The lowest graphs in Figures 5 and 6 illustrate again that naive experiments which fail to isolate the behaviour of PROLOGINT from that of the PM could give misleading results.

7.3 Conclusions and comments on the Lispkit and Prolog interpreters.

The design and assessment exercises reported in this chapter have revealed two particular results. Firstly that there exists a purely functional interpreter for Lispkit which has a linear performance characteristic. Secondly that there also exists a Prolog interpreter for (restricted) Prolog which has a linear performance characteristic. These results are important when looking ahead to the possible availability of special purpose hardware, with linear performance characteristics, to replace the compound virtual machines LM/IBM 370 and PM/IBM 370. On such machines Lispkit and Prolog programs could be interpreted (without compilation) in a _time_ proportional to the computational demands of the program.

The interpreters themselves do not contain any exciting, innovative techniques, but the experiments reported illustrate how the performance assessment methodology can lead to a carefully controlled empirical exploration of interpretation mechanisms - an exploration which has barely begun in this chapter.

CHAPTER 8 - SUMMARY AND CONCLUSIONS.

SUMMARY AND CONCLUSIONS.

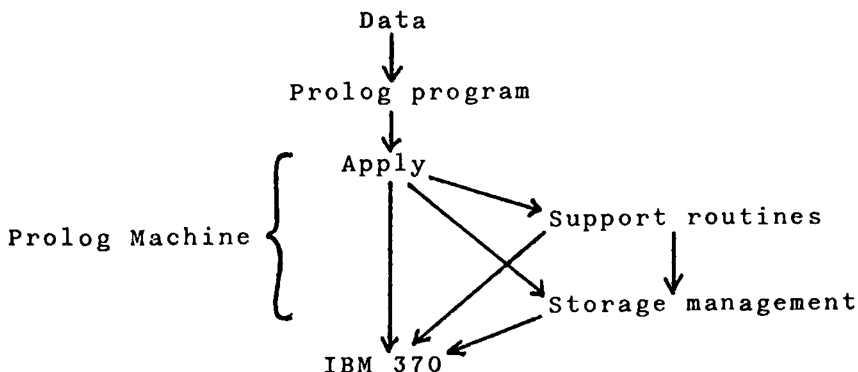
In this thesis I have made a theoretical and practical exploration of the problems of assessing the performance of individual components of interpreter based computer systems.

Of particular interest are interpreter systems for the execution of (very) high level language programs. Chapters 2 and 3 establish the high level language context by introducing the functional and logic styles of Lispkit and Prolog, and by hinting at software pseudo-machine implementations which are described in more detail in Appendices B and D. Although the treatment of Lispkit is quite conventional, the treatment of Prolog is (as far as I am aware) innovative in the use of su-expressions to enable a bottom-up description of the language following the pattern of the Lispkit description; su-expressions and the elaboration of unknowns certainly aid in understanding the operation of the Prolog Machine, but their merit in the understanding of Prolog programming is not certain.

Given that the primary task of performance assessment in multi-level interpreter based systems is to assess the performance of individual components (discussed in Chapter 1 and in the introduction to Chapter 5), Chapters 4 and 5 examine the structure and behaviour of such multi-level systems and their components. A simple form of diagram is adopted for distinguishing and naming individual components in a system (both hardware and software); interfaces separate machine from program, interpreter from program, and program from data in a

linear stacking scheme. The Interpretation model then describes the semantic attributes of systems constructed from innately active hardware components and passive program and data texts. However, the Interpretation model does not enable an active characteristic to be attributed to software components in isolation from the remainder of the system, and so the Abstract Execution model is introduced in which each program or interpreter component is represented by a function which maps the interaction Trace at the higher level interface of the component to the Trace at the lower level interface. The Abstract Execution model can be related to the Interpretation model, but also serves to enable the definition of a performance characteristic as a mapping between statistics of interest extracted from the upper and lower interface Traces of a component.

Although the system structure diagrams are adequate for simple systems, in more complex systems (such as those incorporating the Prolog Machine with its two interpreter extension levels) the meaning is not so clear. In retrospect it might have been better to employ simple directed graphs to make the role played by interpreter extensions more explicit; a directed arc would represent the use of a service provided by a lower level component. For example, the execution of a Prolog program on the PM would be represented by:



This representation, being more explicit, would also be more amenable to re-working of the Interpretation and Abstract Execution models, and the definition of performance, to include interpreter extensions; they are entirely absent at present, though in Chapter 6 the performances of several interpreter extensions are assessed by methods exactly similar to those used for normal interpreter levels.

Several other aspects of the Abstract Execution model and performance assessment methodology remain slightly hazy and would benefit from a closer examination and tighter specification. I am thinking particularly of the concepts of interactions, Traces and model interpretations which seem clear enough to be applied in Chapters 6 and 7, but which might not be so clear in other cases.

Nevertheless, even with considerable refinement the behaviour and performance models can never be more than guidelines for empirical performance assessment, as there must remain flexibility in the choice of interfaces and precise statistics of interest; the models provide a formal framework within which the significance of empirical results is clear.

The guidelines were sufficiently clear to give structure to practical experiments assessing pseudo-machine and high level interpreter implementations of Lispkit and Prolog. These experiments are reported in Chapters 6 and 7. The great attention to detail that was required in planning and carrying out the experiments, and in forming conclusions from the results, is evidence of the

genuine difficulty of making accurate performance assessments. However, I believe that following the assessment methodology has yielded specific observations and results which are unarguably good representations of the performance characteristics of particular software components.

The implementations and experiments were chosen not only for their illustrative properties, but also because the results obtained would touch on some aspects of current research questions, such as "How do functional and logic language compare?" and "What overheads must we suffer in interpreting rather than compiling high level language programs?"

Useful results obtained can be summarised as follows:

1. The Lispkit and Prolog Machines (LM and PM respectively) can each be divided into two distinct software components concerned with pseudo-machine instruction execution and storage management. Both the instruction execution components have a linear performance characteristic, but both the storage management components have non-linear performance characteristics (though they are similar, for similar reasons).
2. For similar programming examples, the PM must execute about twice as many pseudo-machine instructions as the LM.
3. The LM and PM were deliberately constructed using similar programming technology, and from inspection of the AlgolW code the average work per PM pseudo-machine instruction is a small factor (2 to 5, maybe) greater than that for the LM.

4. Hence for similar programs, and ignoring the influence of the storage management components, the PM can be expected to perform about, maybe, 10 times as much work as the LM, but note that this is only a linear factor and applies whatever the size of the computation. This observation is interesting as it may be conjectured that future computing hardware might provide assistance with heap storage management and thus potentially remove (or drastically reduce) the overhead of garbage collection. In this circumstance both the LM and PM would become linear performance pseudo-machines, rather than being progressively less efficient for larger and larger computations.
5. The interpreters LISPINT2 and PROLOGINT each have a linear performance characteristic; a program running on one of the interpreters suffers only a linear factor overhead in the number of pseudo-machine instructions executed (although the factors are rather large, about 50 and 150 respectively). Again this suggests that if the garbage collector overheads could be absorbed into hardware then the entire interpreter virtual machines could have a linear performance.
6. The previous observation also supports the conjecture that direct source code interpretation is not necessarily worse than compiling for a pseudo-machine. A source code interpreter written to execute directly on current hardware would be the composition of a linear performance evaluation mechanism and a non-linear storage management, just as the pseudo-machines are.

The performance assessment methodology has been valuable in the analysis of a small set of high level language interpreters, and there are many other similar cases to which it could be applied. However I hope that the approach I have outlined to the behaviour and performance of multi-level systems is sufficiently clear to guide the investigation of other systems which can be structured in a similar way. Where the approach is inadequate I hope that the notions I have discussed are sufficiently sound and clear to be refined to give a more satisfyingly complete theory of performance assessment.

Appendix A.

A brief history of Lispkit.

This appendix is intended to be no more than a sketch of the origins of Lispkit (Henderson (1980)), and of the influences on its development, with pointers to the appropriate literature. Henderson gives a much broader coverage of theliterature, and of the properties of Lispkit itself.

Lispkit is a significant variant of the language Lisp described by McCarthy (1960). Lisp was important for the introduction of s-expressions as both a data type and program syntax, and for showing the power of recursion in the processing of the tree structured s-expressions.

Landin (1964) discusses the modelling of computational expressions by the applicative structure of Church's lambda notation(Church (1941)(1952)), and an abstract machine for the evaluation of such expressions. The applicative structure defines clearly the scope of local variables qualifying an expression, and shows the properties of function valued expressions. The abstract machine is the four register SECD machine, to which the Lispkit machine in Appendix B is very closely related. The SECD machine is described by an applicative program.

Landin (1966) introduces ISWIM, a family of expression oriented programming languages which take applicative (or "purely functional",or"nonimperative") structure as fundamental, with "procedural notions grafted on in such a way as not to disturb many desirable properties" (paraphrased).

Hence Henderson's Lispkit language and machine have arisen from these ideas - a purely functional language operating on s-expression data structures, with programs represented for computational purposes as s-expressions, and executed on an abstract machine of the same family as Landin's SECD machine. Extensions to the language and machine are made and used with care - such as the introduction of delayed evaluation, and nondeterministic constructs discussed by Henderson.

APPENDIX B - A PSEUDO-MACHINE IMPLEMENTATION OF LISPKIT.

Appendix B

A pseudo-machine implementation of Lispkit.

One possible approach to the implementation of Lispkit is that of designing a special purpose high level "virtual machine". The Lispkit Machine (LM) described in this appendix accepts programs in the form of a special purpose machine language, the primitive operations of which are at a high level of complexity compared to the primitive operations of a conventional computer. Lispkit programs are compiled into the language of the LM. The semantic capabilities of the implemented language are those embodied in the language description and operational model of Lispkit given in Chapter 2, though that is certainly not a formal specification. The operational model is a reasonably accurate reflection of the behaviour of the LM.

Architecturally the LM is a fairly conventional sequential machine in which each sequential step involves a change of state. The state of the machine is held entirely in 4 registers - this includes program and data structures.

This appendix outlines an s-expression notation for Lispkit programs, the high level machine code obtained by compiling such programs, the machine state representation, and the state transitions invoked by each machine instruction. The appendix is not a complete description of a software realisation for the LM, though a listing of the machine used in experiments is included for completeness. Some more details of the software implementation appear in Chapter 6, and a much more complete "kit" is available in Henderson (1980).

B.1 An s-expression syntax for Lispkit.

For practical purposes Lispkit is processed on the computer in the syntactic form of s- expressions. Below I give the correspondence between Lispkit expressions, in the notation of Chapter 2, and one possible s-expression syntax. This particular syntax is that in which the higher level interpreters of Chapter 7 expect to receive Lispkit programs.

In the s-expression forms, an asterisk following a subexpression denotes that the subexpression must also be represented in s-expression form.

A variable is represented by the single symbolic atom obtained by converting the letters in the variable into upper case only, e.g. x1 is represented by X1.

A constant s-expression c is represented by (QUOTE c).

Other expressions:

car(x,y) is represented by (CAR x* y*),
cdr(x,y) is represented by (CDR x* y*),
cons(x,y)is represented by (CONS x* y*),
x+y is represented by (ADD x* y*),
x-y is represented by (SUB x* y*),
x*y is represented by (MUL x* y*),
x <u>div</u> y is represented by (DIV x* y*),
x <u>rem</u> y is represented by (REM x* y*),
eq(x,y) is represented by (EQ x* y*),
x≤y is represented by (LEQ x* y*),
atom(x) is represented by (ATOM x*),

if c *then* x *else* y is represented by (IF c* x* y*),
λ(x1, ...)e is represented by (LAMBDA(x1* ...) e*),
e *where* x1=e1 ... is represented by
(LET e* (x1*.e1*)...),
e *whererec* x1=e1 ... is represented by
(LETREC e* (x1*.e*)...),
f(e1, ...) is represented by (f* e1* ...).

For example, the naive reverse program:

```
reverse
whererec reverse (l) = if eq(l,NIL) then NIL
                       else append(reverse(cdr(l)),cons(car(l),NIL))
and append (l1,l2) = if eq(l1, NIL) then l2
                 else cons(car(l1), append(cdr(l1),l2))
```

is represented by

```
(LETREC  REVERSE
  (REVERSE LAMBDA (L)
     (IF (EQ L (QUOTE NIL))   (QUOTE NIL)
       (APPEND (REVERSE (CDR L))
               (CONS(CAR L)(QUOTE NIL)))))
  (APPEND LAMBDA (L1 L2)
     (IF (EQ L1(QUOTE NIL))  L2
         (CONS(CAR L1) (APPEND(CDR L1) L2)))))
```

B.2 Compiling Lispkit programs for the LM.

The compilation process must take a Lispkit program, which represents a function, and produce an s-expression machine code program for the LM which evaluates the function, applies it to some arguments and then halts.

I shall give here, in an informal fashion, the code skeletons generated for each type of expression in Lispkit. The compiled form of subexpressions will be denoted by the use of a postfixed asterisk *. A

vertical bar | indicates list concatenation.

The compiler maintains a correctly structured environment of variable names, and the indices required when compiling a variable reference are obtained by examining the environment. Treatment of the environment will be completely implicit in the code skeletons, for simplicity (Henderson gives a full treatment).

A program p is compiled and the instructions AP and STOP are added to give the resultant object code:

```
        p* | (AP STOP)
```

in which p* evaluates the function closure, and AP applies it to the input argument list.

Individual expression types:

```
 x*  =  (LD (m.n))
        where x is a variable
        and m,n are found by inspecting the environment
        and locating x,
(QUOTE e)* = (LDC e),
(CAR e)*   = e* | (CAR),
(CDR e)*   = e* | (CDR),
(CONS e1 e2)* = e2* | e1* | (CONS),
(ADD e1 e2)* = e1* | e2* | (ADD),
(SUB e1 e2)* = e1* | e2* | (SUB),
(MUL e1 e2)* = e1* | e2* | (MUL),
(DIV e1 e2)* = e1* | e2* | (DIV),
(REM e1 e2)* = e1* | e2* | (REM),
(EQ e1 e2)*  = e1* | e2* | (EQ),
(LEQ e1 e2)* = e1* | e2* | (LEQ),
(ATOM e)*  =  e* | (ATOM),
```

```
(IF e1 e2 e3)* = e1* | (SEL e2* | (JOIN) e3* | (JOIN)),
(LAMBDA (x1 ...) e)* = (LDF e* | (RTN))
                       and the environment for compiling
                       e is extended with the variable
                       names (... x1),
(f e1 ...)* = (LDC NIL) | e1* | (CONS) | ... | f* | (AP)
(LET e (x1.e1)...)* =
       (LDC NIL) | e1* | (CONS) | ... | (LDF e* | (RTN) AP)
       and the environment for compiling e
       is extended with the variable names (... x1),
(LETREC e (x1.e1) ...)* =
        (DUM LDC NIL) | e1* | (CONS) | ... | (LDF e* | (RTN)RAP)
        and the environment for compiling e,e1,... is
        extended with the variable names (... x1).
```

For example, the naive reverse program of the previous section compiles to give the code:

```
  (DUM LDC NIL
       LDF (LD (0.0) LDC NIL EQ
            SEL (LDC NIL JOIN)
                (LDC NIL
                 LDC NIL LD (0.0) CDR CONS LD (1.1) AP
                 CONS
                 LDC NIL LD (0.0) CAR CONS
                 CONS LD (1.0) AP JOIN)
            RTN)
       CONS
       LDF (LD (0.1) LDC NIL EQ
            SEL (LD (0.0) JOIN)
                (LDC NIL LD (0.1) CDR CONS
                         LD (0.0) CONS
                         LD (1.0) AP
                 LD (0.1) CAR CONS JOIN)
            RTN).
      CONS
      LDF (LD (0.1) RTN)
RAP
AP STOP)
```

B.3 The Lispkit Machine.

B.3.1 The registers.

At each step of the computation the entire machine state is held in 4 registers, S,E,C and D. These registers are the memory of the LM and contain the program code, results, and working storage.

The value held by each register is an s-expression, and hence all operations of the machine are performed by constructing new s-expressions, and by selecting subexpressions. In the machine transitions described below the register values will be represented as s-expressions, but with variables naming whole expressions or subexpressions in order to be able to identify identical values before and after transitions.

The general roles of the 4 registers can be described as follows:

The S register ("Stack") is used for temporary working storage in the construction of data structures required by the machine. S is a list of values, and the head of the list corresponds to the head of the stack.

The E register ("Environment") contains a list of lists corresponding to the environment of defined values in which Lispkit expressions are evaluated. Each member of the inner lists is a defined value, and each inner list contains all the values defined as a group (in a where expression, whererec expression or function application). Inner lists nearer the head of the

environment correspond to inner levels of definitions. E grows and shrinks during a computation in such a way that the "current local variables" are always at the head of E. Variable values in E are accessed by using an ordered pair of indices generated when the program is compiled. The ordered pair (x.y) locates the value which is the (y+1)th member of the (x+1)th inner list of E.

The C register ("Control") contains the sequence of instructions currently being executed. The instruction at the head of C is the next to be executed. New instruction sequences are loaded into C primarily at function application and conditional expressions. (Note, from the compiler code skeletons in this appendix, that <u>where</u> and <u>whererec</u> expressions are executed like function applications).

The D register ("Dump") maintains a record of instruction sequences suspended at function applications and of the corresponding S and E values which must be restored when the current function returns its result. D is also used during conditional expression evaluation.

B.3.2 The state transition rules.

There are three phases of execution to be tackled here. Firstly the LM must be set up with the code of the program to be executed, and with its data. Secondly the transitions which accomplish the computation must be described. Finally the result of the computation must be extracted from the final state of the machine.

Each state of the LM will be given as an ordered quadruple (S,E,C,D) in which the items correspond to the appropriately named registers above.

Initially the input to the machine consists of an s-expression, fn, which is a machine language program, and a list of s-expression values, args, forming the input arguments to the program. The program fn contains code to evaluate a closure, to apply the closure to arguments at the head of S, and then to halt. Hence the initial state of the LM is

((args), NIL, fn, NIL)

i.e. S =cons(args, NIL) E = D = NIL C=fn

The computation proceeds by executing LM instructions from the head of C. The LM has a repertoire of 21 instructions represented by the mnemonics LD,LDC,CONS, CAR,CDR,ADD,SUB,MUL,DIV,REM,EQ,LEQ,ATOM,SEL,JOIN,LDF, AP,DUM,RAP,RTN,STOP. Each instruction determines exactly one of the following transition rules, in which the previous and next states are shown on the left and right sides of a right pointing arrow, $\longrightarrow$.

To load the value of a variable or constant onto S:

(s,e,(LD (m.n).c),d) $\longrightarrow$ ((x.s),e,c,d)

where x is obtained from e:

$$e = (\underbrace{\ldots}_{\substack{m \\ \text{items}}} (\underbrace{\ldots}_{\substack{n \\ \text{items}}} x \ldots) \ldots)$$

(s,e (LDC x.c),d) $\longrightarrow$ ((x.s),e,c,d)

List construction and dissection on S:

```
((a b.s),e,(CONS.c),d)→(((a.b).s),e,c,d)
(((a.b).s),e,(CAR.c),d)→((a.s),e,c,d)
(((a.b).s),e,(CDR.c),d)→((b.s),e,c,d)
```

Arithmetic on S. Note that the *second* operand is at the head of S:

```
((x y.s),e,(ADD.c),d)→((z.s),e,c,d) where z=y+x
((x y.s),e,(SUB.c),d)→((z.s),e,c,d) where z=y-x
((x y.s),e,(MUL.c),d)→((z.s),e,c,d) where z=y*x
((x y.s),e,(DIV.c),d)→((z.s),e,c,d) where z=y div x
((x y.s),e,(REM.c),d)→((z.s),e,c,d) where z=y rem x
```

Predicates testing the head members of S:

```
((x y.s),e,(EQ.c),d)→((b.s),e,c,d)
                    where b=T if x, y are equal atoms
                          =F otherwise

((x y.s),e,(LEQ.c),d)→((b.s),e,c,d)
                    where, assuming x and y are numbers,
                        b=T if y  x
                          =F otherwise

((x.s),e,(ATOM.c),d)→((b.s),e,c,d)
                    where b=T if x is an atom
                          = F otherwise
```

Conditional expression branching and rejoining:

```
((b.s),e,(SEL c1 c2.c),d)→(s,e,csel,(c.d))
                        where csel=c1 if b=T
                                  =c2 otherwise

(s,e,(JOIN),(c.d))→(s,e,c,d)
```

Constructing a function closure:

```
(s,e,(LDF body.c),d)→(((body.e).s),e,c,d)
```

Applying a function, or evaluating an expression qualified by <u>where</u>:

```
    (((body.eclos) args.s),e,(AP.c),d)
                 -->(NIL,(args.eclos),body,(s e c.d))
```

Constructing a recursive environment, and evaluating an expression qualified by <u>whererec</u>:

```
  (s,e,(DUM.c),d) --> (s,(x.e),c,d)
                          where x is any value (NIL would be
                                                  appropriate)

  (((body.eclos) args.s), eclos,(RAP.c),d)
              -->(NIL,eclos', body,(s e c.d))
                 where  e=cdr (eclos)
                   and eclos' is obtained from eclos by
                              modifying the car field of
                              eclos to be args.
                              Informally:
                              eclos=(x.e)=(args.e)=eclos'
```

Returning the result from function applications:

```
  ((x.s1), e1,(RTN),(s2 e2 c.d))-->((x.s2), e2,c,d)
```

Terminating the computation and returning the result:

```
  ((x.s),e,(STOP),d) --> No successor state
```

When the computation has terminated in the state

```
  ((x.s),e,(STOP),d)
```

the result which is extracted from the machine is the value of x from the head of the stack S.

B.4 An AlgolW software realisation of the LM.

The following pages contain a listing of the AlgolW implementation of the LM which was used for the experimental work reported in this thesis.

```
ALGOL W : LISP.MACHINE   DEFINITION OF LIST STORAGE AREA   1981 JUNE

0000 1-       BEGIN
0001 --
0001 --       INTEGER PAGES,NO_OF_ELEMS,NO_OF_SYMBS,
0001 --               FIRST_ELEM,LAST_ELEM,
0001 --               FIRST_SYMB,LAST_SYMB;
0002 --       WRITE("NO. OF PAGES, NO. OF CONSES/NUMBS ?"); IOCONTROL(2);
0004 --       READ( PAGES,NO_OF_ELEMS );
0005 --       IF (340*(PAGES-6))<NO_OF_ELEMS THEN
0005 2-       BEGIN NO_OF_ELEMS:=340*(PAGES-6);
0007 --          WRITE("NO. OF CONSES/NUMBS TOO GREAT. REDUCED TO ",NO_OF_ELEMS);
0008 --          IOCONTROL(2);
0009 -2       END;
0010 --       NO_OF_SYMBS:=2000; %#P OF SYMBS%
0011 --       FIRST_ELEM:=1; LAST_ELEM:=FIRST_ELEM+NO_OF_ELEMS-1;
0013 --       FIRST_SYMB:=LAST_ELEM+1; LAST_SYMB:=FIRST_SYMB+NO_OF_SYMBS-1;
0015 --
0015 2-       BEGIN
0016 --
0016 --          INTEGER NIL,NEXT_ELEM,NEXT_SYMB;
0017 --
0017 --          INTEGER ARRAY FLAGS,CAR,CDR(FIRST_ELEM::LAST_ELEM);
0018 --          STRING(12) ARRAY SVAL(FIRST_SYMB::LAST_SYMB);
0019 --

0019 --          PROCEDURE INIT_LIST_STORAGE;
0020 3-          BEGIN NEXT_ELEM:=FIRST_ELEM;
0022 --             NEXT_SYMB:=FIRST_SYMB;  NIL:=SYMB("NIL");
0024 --             FOR I:=FIRST_ELEM UNTIL LAST_ELEM DO CAR(I):=I+1;
0025 -3          END;
0026 --

0026 --          INTEGER PROCEDURE SIMPLECONS(INTEGER VALUE NEWCAR,NEWCDR);
0027 3-          BEGIN INTEGER TEMP;       %ONLY USED IN GETEXP%
0029 --             IF NEXT_ELEM>LAST_ELEM THEN
0029 4-                BEGIN WRITE("CONS/NUMB SPACE OVERFLOW DURING S-EXP INPUT");
0031 --                   ASSERT FALSE;
0032 -4                END;
0033 --             TEMP:=NEXT_ELEM;  NEXT_ELEM:=CAR(NEXT_ELEM);
0035 --             CAR(TEMP):=NEWCAR;  CDR(TEMP):=NEWCDR;  FLAGS(TEMP):=1;
0038 --             TEMP
0038 -3          END;
0039 --

0039 --          INTEGER PROCEDURE CONS(INTEGER VALUE NEWCAR,NEWCDR);
0040 3-          BEGIN INTEGER TEMP;
0042 --             IF NEXT_ELEM>LAST_ELEM THEN GARBAGE_COLLECT;
0043 --             TEMP:=NEXT_ELEM;  NEXT_ELEM:=CAR(NEXT_ELEM);
0045 --             CAR(TEMP):=NEWCAR;   CDR(TEMP):=NEWCDR;  FLAGS(TEMP):=1;
0048 --             TEMP
0048 -3          END;

0049 --          INTEGER PROCEDURE SIMPLENUMB(INTEGER VALUE NEWNUMB);
0050 3-          BEGIN INTEGER TEMP;  %ONLY USED IN GETEXP%
0052 --             IF NEXT_ELEM>LAST_ELEM THEN
0052 4-                BEGIN WRITE("CONS/NUMB SPACE OVERFLOW DURING S-EXP INPUT");
0054 --                   ASSERT FALSE;
0055 -4                END;
0056 --             TEMP:=NEXT_ELEM;  NEXT_ELEM:=CAR(NEXT_ELEM);
0058 --             CAR(TEMP):=NEWNUMB; FLAGS(TEMP):=2;
0060 --             TEMP
0060 -3          END;
0061 --
0061 --          INTEGER PROCEDURE NUMB(INTEGER VALUE NEWNUMB);
0062 3-          BEGIN INTEGER TEMP;
0064 --             IF NEXT_ELEM>LAST_ELEM THEN GARBAGE_COLLECT;
0065 --             TEMP:=NEXT_ELEM;  NEXT_ELEM:=CAR(NEXT_ELEM);
0067 --             CAR(TEMP):=NEWNUMB;  FLAGS(TEMP):=2;
0069 --             TEMP
0069 -3          END;
0070 --

0070 --          INTEGER PROCEDURE SYMB(STRING(12) VALUE NEWSYMB);
0071 3-          BEGIN INTEGER TEMP;
0073 --             IF NEXT_SYMB>LAST_SYMB THEN
0073 4-                BEGIN WRITE("SYMBOL SPACE OVERFLOW");
0075 --                   ASSERT FALSE;
0076 -4                END;
0077 --             TEMP:=NEXT_SYMB;  NEXT_SYMB:=NEXT_SYMB+1;
0079 --             SVAL(TEMP):=NEWSYMB;
0080 --             TEMP
0080 -3          END;
0081 --

0081 --          LOGICAL PROCEDURE ISCONS(INTEGER VALUE PTR);
0082 --          FIRST_ELEM<=PTR AND PTR<=LAST_ELEM AND ABS(FLAGS(PTR))=1;
0083 --

0083 --          LOGICAL PROCEDURE ISNUMB(INTEGER VALUE PTR);
0084 --          FIRST_ELEM<=PTR AND PTR<=LAST_ELEM AND ABS(FLAGS(PTR))=2;
0085 --

0085 --          LOGICAL PROCEDURE ISSYMB(INTEGER VALUE PTR);
0086 --          FIRST_SYMB<=PTR AND PTR<=LAST_SYMB;
0087 --

0087 --          LOGICAL PROCEDURE ISELEM(INTEGER VALUE PTR);
0088 --          FIRST_ELEM<=PTR AND PTR<=LAST_ELEM;
0089 --
```

```
ALGOL W : LISP.MACHINE   GARBAGE COLLECTION                 1961 JUNE

0089 --        INTEGER ARRAY STACK(1::1000);  INTEGER ST_PTR;
0091 --

0091 --        PROCEDURE GARBAGE_COLLECT;
0092 3-        BEGIN MARK_FROM(S); MARK_FROM(E); MARK_FROM(C);
0096 --           MARK_FROM(D); MARK_FROM(W);
0098 --
0098 --           NEXT_ELEM:=LAST_ELEM+1;
0099 --           FOR I:=FIRST_ELEM UNTIL LAST_ELEM DO
0099 --             IF FLAGS(I)<0 THEN FLAGS(I):=ABS(FLAGS(I))
0099 4-                           ELSE BEGIN CAR(I):=NEXT_ELEM;
0101 -4                                      NEXT_ELEM:=I;  END;
0103 --
0103 --           IF NEXT_ELEM>LAST_ELEM THEN
0103 4-             BEGIN WRITE("CONS/NUMB SPACE OVERFLOW");
0105 -4                ASSERT FALSE;  END;
0107 --
0107 -3        END GARBAGE_COLLECT;
0108 --

0108 --        PROCEDURE MARK_FROM(INTEGER VALUE PTR);
0109 3-        BEGIN ST_PTR:=2;   STACK(1):=PTR;
0112 --
0112 --           WHILE ST_PTR¬=1 DO
0112 4-           BEGIN ST_PTR:=ST_PTR-1;   PTR:=STACK(ST_PTR);
0115 --              WHILE ISCONS(PTR) AND FLAGS(PTR)>0 DO
0115 5-              BEGIN FLAGS(PTR):=-1;
0117 --                 IF ST_PTR>1000 THEN
0117 6-                    BEGIN WRITE("GARBAGE COLLECTOR STACK OVERFLOW");
0119 --                       ASSERT FALSE;
0120 -6                    END;
0121 --                 STACK(ST_PTR):=CDR(PTR); ST_PTR:=ST_PTR+1;
0123 --                 PTR:=CAR(PTR);
0124 -5              END;
0125 --              IF ISNUMB(PTR) THEN FLAGS(PTR):=-2;
0126 -4           END;
0127 --
0127 -3        END MARK_FROM;
```

```
ALGOL W : LISP.MACHINE   INPUT AND OUTPUT OF S-EXPRESSIONS 1981 JUNE

0128 --        STRING(12) TOKEN,TYPE;
0129 --

0129 --        PROCEDURE SCAN;
0130 33        BEGIN GETTOKEN(TOKEN,TYPE); IF TYPE="ENDFILE" THEN TOKEN:=")" END;
0133 --

0133 --        PROCEDURE GETEXP(INTEGER RESULT E);
0134 --        IF TOKEN="(" THEN
0134 33        BEGIN SCAN; GETEXPLIST(E); ASSERT TOKEN=")"; SCAN END ELSE
0136 --        IF TYPE="NUMERIC" THEN
0138 33              BEGIN E:=SIMPLENUMB(TOINTEGER(TOKEN)); SCAN END ELSE
0140 33        BEGIN E:=SYMB(TOKEN); SCAN END;
0143 --

0143 --        PROCEDURE GETEXPLIST(INTEGER RESULT E);
0144 3-        BEGIN E:=SIMPLECONS(0,0); GETEXP(CAR(E));
0147 44          IF TOKEN="." THEN BEGIN SCAN; GETEXP(CDR(E)) END ELSE
0149 -3          IF TOKEN=")" THEN CDR(E):=NIL ELSE GETEXPLIST(CDR(E)) END;
0150 --

0150 --        PROCEDURE PUTEXP(INTEGER VALUE E);
0151 --        IF ISSYMB(E) THEN PUTTOKEN(SVAL(E)) ELSE
0151 --        IF ISNUMB(E) THEN PUTTOKEN(TOSTRING(CAR(E))) ELSE
0151 3-        BEGIN PUTTOKEN("(");
0153 --        WHILE ISCONS(E) DO
0153 44          BEGIN PUTEXP(CAR(E)); E:=CDR(E) END;
0156 --              IF ISSYMB(E) AND SVAL(E)="NIL" THEN ELSE
0156 43              BEGIN PUTTOKEN("."); PUTEXP(E) END; PUTTOKEN(")") END;
0160 --

0160 --        INTEGER PROCEDURE TOINTEGER(STRING(12)VALUE T);
0161 3-        BEGIN STRING(1)D; INTEGER I,S; LOGICAL NEG; S:=0;
0166 44          IF T(0|1)="-" THEN BEGIN NEG:=TRUE; I:=1 END
0168 44          ELSE BEGIN NEG:=FALSE; I:=0 END; D:=T(I|1);
0172 --          WHILE D¬=" " DO
0172 4-          BEGIN S:=10*S+(DECODE(D)-DECODE("0")); I:=I+1;
0175 --                IF I>11 THEN D:=" " ELSE D:=T(I|1);
0176 -4          END;
0177 --          IF NEG THEN -S ELSE S
0177 -3        END;
0178 --

0178 --        STRING(12) PROCEDURE TOSTRING(INTEGER VALUE I);
0179 --        IF I=0 THEN "0" ELSE
0179 3-        BEGIN STRING(12)T; LOGICAL NEG; INTEGER L;
0183 --          NEG:=I<0; T:=" ";
0185 --          I:=ABS(I);L:=2+TRUNCATE(LOG(I+.1));
0187 --          FOR J:= L-1 STEP -1 UNTIL 1 DO
0187 44          BEGIN T(J|1):=CODE(DECODE("0")+I REM 10); I:=I DIV 10 END;
0190 --        IF NEG THEN T(0|1):="-"; T
0191 -3        END;
0192 --

0192 --        PROCEDURE INIT_SYNTAX; SCAN;
```

```
ALGOL W : LISP.MACHINE    INPUT/OUTPUT OF TOKENS                1981 JUNE

0194 --         STRING(1)ARRAY INBUFFER(1::256); INTEGER INBUFPTR,INBUFEND;
0196 --         LOGICAL EOF;
0197 --

0197 --         PROCEDURE GETCHAR;
0198 3-         BEGIN IF INBUFPTR>INBUFEND THEN
0199 4-               BEGIN GETLINE(INBUFFER,INBUFEND,EOF);
0201 -4                     INBUFPTR:=1 END;
0202 -3           CH:=INBUFFER(INBUFPTR); INBUFPTR:=INBUFPTR+1 END;
0204 --
0204 --         STRING(1)CH;
0205 --

0205 --         PROCEDURE GETTOKEN(STRING(12)RESULT TOKEN,TYPE);
0206 3-         BEGIN TOKEN:=" "; WHILE¬EOF AND CH=" " DO GETCHAR;
0209 --         IF EOF THEN TYPE:="ENDFILE" ELSE
0209 --         IF "0"<=CH AND CH<="9" OR CH="-" THEN
0209 4-         BEGIN INTEGER I; TYPE:="NUMERIC";
0212 --           TOKEN(0|1):=CH; I:=1; GETCHAR;
0215 --           WHILE "0"<=CH AND CH<="9" DO
0215 55           BEGIN ASSERT I<12; TOKEN(I|1):=CH; I:=I+1; GETCHAR END
0219 -4         END ELSE
0219 --         IF "A"<=CH AND CH<="Z" THEN
0219 4-         BEGIN INTEGER I; TYPE:="ALPHANUMERIC";
0222 --           TOKEN(0|1):=CH; I:=1; GETCHAR;
0225 --           WHILE "A"<=CH AND CH <="9" DO
0225 5-           BEGIN
0226 --            ASSERT I<12; TOKEN(I|1):=CH; I:=I+1; GETCHAR
0229 -5           END
0229 -4         END ELSE
0229 4-         BEGIN TYPE:="DELIMITER";
0231 53           BEGIN TOKEN(0|1):=CH; GETCHAR END END END;
0234 --
0234 --         STRING(1)ARRAY OUTBUFFER(1::80); INTEGER OUTBUFPTR;
0236 --

0236 --         PROCEDURE PUTCHAR(STRING(1)VALUE C);
0237 3-         BEGIN IF OUTBUFPTR=80 THEN FORCELINEOUT;
0239 -3           OUTBUFPTR:=OUTBUFPTR+1; OUTBUFFER(OUTBUFPTR):=C END;
0241 --

0241 --         PROCEDURE FORCELINEOUT;
0242 33         BEGIN PUTLINE(OUTBUFFER,OUTBUFPTR); OUTBUFPTR:=0 END;
0245 --

0245 --         PROCEDURE PUTTOKEN(STRING(12)VALUE T);
0246 3-         BEGIN INTEGER LEN; LEN:=12;
0249 --           WHILE LEN>0 AND T(LEN-1|1)=" " DO LEN:=LEN-1;
0250 -3           FOR I:=0 UNTIL LEN-1 DO PUTCHAR(T(I|1)); PUTCHAR(" ") END;
0252 --

0252 --         PROCEDURE INIT_LEXICAL;
0253 3-         BEGIN GETLINE(INBUFFER,INBUFEND,EOF); INBUFPTR:=1; GETCHAR;
0257 -3           OUTBUFPTR:=0 END;
```

```
ALGOL W : LISP.MACHINE    INPUT/OUTPUT OF LINES                 1981 JUNE

0258 --         PROCEDURE GET(STRING(1)ARRAY LINE(*);INTEGER RESULT LEN;
0260 --                                      INTEGER VALUE MOD,LNUM,DEV);
0261 --         FORTRAN "READ";
0262 --

0262 --         PROCEDURE PUT(STRING(1)ARRAY LINE(*); INTEGER VALUE LEN,MOD,
0263 --                                    LNUM,DEV);
0264 --         FORTRAN "WRITE";
0265 --

0265 --         PROCEDURE GETLINE(STRING(1)ARRAY LINE(*);INTEGER RESULT LEN;
0267 --                                             LOGICAL RESULT EOF);
0268 3-         BEGIN LINE(1):=" "; GET(LINE,LEN,C,0,0);
0271 --               EOF :=R_CODE>0;
0272 --               LEN:=NUMBER(BITSTRING(LEN) SHR 16);
0273 -3         END;
0274 --

0274 --         PROCEDURE PUTLINE(STRING(1)ARRAY LINE(*); INTEGER VALUE LEN);
0276 --         PUT(LINE,NUMBER(BITSTRING(LEN) SHL 16),0,0,1);
```

```
ALGOL W : LISP.MACHINE   EXECUTION CYCLE                     1981 JUNE

0277 --         INTEGER S,E,C,D,W;
0278 --

0278 --         INTEGER PROCEDURE APPLY(INTEGER VALUE FN,ARGS);
0279 3-         BEGIN INTEGER T,F;
0281 --             S:=CONS(ARGS,NIL); E:=NIL; C:=FN; D:=NIL; W:=NIL;
0286 --             T:=SYMB("T"); F:=SYMB("F");
0288 --
0288 --             CYCLE:CASE CAR(CAR(C)) OF
0288 4-             BEGIN
0289 5-             BEGIN LD   : W:=LOOKUP(CAR(CAR(CAR(CDR(C)))),CAR(CDR(CAR(CDR(C)))),E);
0291 -5                          S:=CONS(W,S); C:=CDR(CDR(C)) END;
0293 55            BEGIN LDC  : S:=CONS(CAR(CDR(C)),S); C:=CDR(CDR(C)) END;
0296 55            BEGIN LDF  : W:=CONS(CAR(CDR(C)),E); S:=CONS(W,S); C:=CDR(CDR(C)) END;
0300 5-            BEGIN AP   : D:=CONS(CDR(C),D); D:=CONS(E,D); D:=CONS(CDR(CDR(S)),D);
0304 --                         E:=CONS(CAR(CDR(S)),CDR(CAR(S))); C:=CAR(CAR(S));
0306 -5                         S:=NIL END;
0307 5-            BEGIN RTN  : S:=CONS(CAR(S),CAR(D));
0309 --                         E:=CAR(CDR(D)); C:=CAR(CDR(CDR(D)));
0311 -5                         D:=CDR(CDR(CDR(D))) END;
0312 55            BEGIN DUM  : E:=CONS(NIL,E); C:=CDR(C) END;
0315 5-            BEGIN RAP  : D:=CONS(CDR(C),D); D:=CONS(CDR(E),D);
0318 --                                                   D:=CONS(CDR(CDR(S)),D);
0319 --                         E:=CDR(CAR(S)); CAR(E):=CAR(CDR(S)); C:=CAR(CAR(S));
0322 -5                         S:=NIL END;
0323 5-            BEGIN SEL  : D:=CONS(CDR(CDR(CDR(C))),D);
0325 --                         IF SVAL(CAR(S))="T" THEN C:=CAR(CDR(C)) ELSE
0325 -5                         C:=CAR(CDR(CDR(C))); S:=CDR(S) END;
0327 55            BEGIN JOIN: C:=CAR(D); D:=CDR(D) END;
0330 55            BEGIN CAR_: S:=CONS(CAR(CAR(S)),CDR(S)); C:=CDR(C) END;
0333 55            BEGIN CDR_: S:=CONS(CDR(CAR(S)),CDR(S)); C:=CDR(C) END;
0336 5-            BEGIN ATOM: IF ISSYMB(CAR(S)) OR ISNUMB(CAR(S)) THEN
0337 -5                        S:=CONS(T,CDR(S)) ELSE S:=CONS(F,CDR(S)); C:=CDR(C) END;
0339 5-            BEGIN CONS_:W:=CONS(CAR(S),CAR(CDR(S))); S:=CONS(W,CDR(CDR(S)));
0342 -5                        C:=CDR(C) END;
0343 5-            BEGIN EQ_  : IF ISSYMB(CAR(S)) AND ISSYMB(CAR(CDR(S))) AND
0344 --                            SVAL(CAR(S))=SVAL(CAR(CDR(S))) OR
0344 --                            ISNUMB(CAR(S)) AND ISNUMB(CAR(CDR(S))) AND
0344 --                            CAR(CAR(S))=CAR(CAR(CDR(S))) THEN S:=CONS(T,CDR(CDR(S)))
0344 -5                        ELSE S:=CONS(F,CDR(CDR(S))); C:=CDR(C) END;
0346 --
0346 55            BEGIN ADD  : S:=ARITH(1,S);  C:=CDR(C) END;
0349 55            BEGIN SUB  : S:=ARITH(2,S);  C:=CDR(C) END;
0352 55            BEGIN MUL  : S:=ARITH(3,S);  C:=CDR(C) END;
0355 55            BEGIN DIV_ : S:=ARITH(4,S);  C:=CDR(C) END;
0358 55            BEGIN REM_ : S:=ARITH(5,S);  C:=CDR(C) END;
0361 --
0361 5-            BEGIN LEQ  : IF CAR(CAR(CDR(S)))<=CAR(CAR(S)) THEN
0362 --                         S:=CONS(T,CDR(CDR(S))) ELSE S:=CONS(F,CDR(CDR(S)));
0363 -5                         C:=CDR(C); END;
0365 55            BEGIN STOP: GO TO ENDCYCLE END;
0367 -4            END; GOTO CYCLE;
0369 --
0369 --            ENDCYCLE : CAR(S)
0369 -3         END;

0370 --         INTEGER PROCEDURE LOOKUP(INTEGER VALUE LEVELS,ITEMS,W);
0371 3-         BEGIN FOR I:=1 UNTIL LEVELS DO W:=CDR(W); W:=CAR(W);
0374 --               FOR I:=1 UNTIL ITEMS DO W:=CDR(W); W:=CAR(W);
0376 --               W
0376 -3         END LOOKUP;
0377 --

0377 --         INTEGER PROCEDURE ARITH(INTEGER VALUE OP,S);
0378 3-         BEGIN
0379 --            W:=CASE OP OF   %NOTE THAT AN IVAL IS SELECTED BY A CAR FIELD%
0379 --               (   NUMB(CAR(CAR(CDR(S)))+CAR(CAR(S))),
0379 --                   NUMB(CAR(CAR(CDR(S)))-CAR(CAR(S))),
0379 --                   NUMB(CAR(CAR(CDR(S)))*CAR(CAR(S))),
0379 --                   NUMB(CAR(CAR(CDR(S))) DIV CAR(CAR(S))),
0379 --                   NUMB(CAR(CAR(CDR(S))) REM CAR(CAR(S))) );
0380 --            CONS(W,CDR(CDR(S)))
0380 -3         END;
```

```
ALGOL W : LISP.MACHINE   CONTROL OPERATION                   1981 JUNE

0381 --         INTEGER FN,ARGS,RES;
0382 --         INIT_LEXICAL;  INIT_SYNTAX;  INIT_LIST_STORAGE;
0385 --         GETEXP(FN);  GETEXPLIST(ARGS);
0387 --         RES:=APPLY(FN,ARGS);
0388 --         PUTEXP(RES);  FORCELINEOUT;
0390 -2      END MAIN_BLOCK;
0391 -1      END.
```

Appendix C.

A brief history of Prolog.

This appendix is a brief sketch of the origins and development of my personal dialect of Prolog and its implementation, as described in Chapter 3 and Appendix D respectively. I will provide a few signposts into the literature of logic programming. Kowalski (1979) tackles the application of logic methods to problem solving in great depth, and also provides an extensive bibliography.

The Prolog language and its semantics arise from the notions of theorem proving as applied to a restricted form of sentences in the first order predicate calculus.

A problem is represented as sentences, or statements, in the quantifier free clausal form of logic (Nilsson (1971)). The solution of the problem is found by discovering an inconsistent set of instances of the statements. Robinson (1963) describes the "combinatorial explosion" which, in general, hampers the search for an inconsistent set of instances.

The use of unification with the resolution inference method are presented in Robinson (1965) and (1967), with the emphasis on a more directed search for inconsistent sets of instances.

Kowalski (1974) noted that the Horn clause subset of sentences has a useful interpretation as procedures describing relations or predicates over symbolic data structures, and that problems expressed in this form were programs which could be executed by an efficient, top down, theorem prover.

Horn clause logic thus adopted the status of a prototype for a programming language ("Prolog"), and specially designed interpreters and compilers appeared (for example Roussel (1975), Warren (1977), and Clark (1979)). However, each of these systems incorporates extra features to overcome fundamental limitations of programming in the Horn clause style. These include: evaluable predicates (to avoid the necessity of defining arithmetic and other basic relations explicitly), "cut" (to provide explicit control of the backtracking of the theorem prover), negated antecedents in clauses, side effects (allowing a program to change the clauses from which it is built), and control flow annotations (to guide the activity of the theorem prover).

The Prolog language of Chapter 3 is a restrained extension of Horn clause programming but compatible with enabling a reasonable set of interesting applications. Simple evaluable ("primitive") predicates are provided in a limited form. Negation of antecedents is important and has been included, but its implementation is problematical. Negation steps outside the Horn clause formalism, back towards general sentences, and the semantics of negation, in the context of the efficient Horn clause theorem prover, must be considered carefully if spurious, incorrect results are to be avoided (Clark (1978)). Warren's Prolog and Clark's IC-Prolog differ in their treatment of negation. I have chosen to follow IC-Prolog's safer scheme, by disallowing the continuation of a computation when theory shows that the outcome of the execution of a negation is inconclusive in determining success or failure. My dialect of Prolog is also constrained to manipulating symbolic expressions built using the s-expression dot constructor - but clearly other constructors can be simulated by appropriate s-expressions.

The Prolog implementation described in Appendix D has been a deliberate attempt to follow the abstract machine style of Landin (1964), as discussed in the context of Lispkit in Appendix A.

APPENDIX D. A PSEUDO-MACHINE IMPLEMENTATION OF PROLOG.

Appendix D.

A pseudo-machine implementation of Prolog.

The implementation of Prolog described here follows the same design principles as the Lispkit machine in Appendix B. The implementation is a high level Prolog Machine (PM) which requires that Prolog programs be compiled into a special purpose intermediate machine language before execution. The semantic capabilities of the implemented language are those outlined in the language description and operational model of Chapter 3, though that is not a formal specification. The operational model is a reasonably accurate reflection of the behaviour of the PM.

Architecturally the PM is a fairly conventional sequential machine in which each sequential step involves a change of the machine state. The state of the machine is held in 9 registers - this includes program code and data structures.

I have chosen to follow the Lispkit machine paradigm in the design and presentation of the PM; the sequential architecture is a straightforward style to handle. Although conceived and designed independently, the sequential machine style is very similar to Warren's Prolog implementation (Warren (1977)). Warren compiles Prolog programs into the machine language of the DEC10 computer, although the code skeletons are based on the microprograms for high level machine operations. I choose to maintain the integrity of a single high level machine, and to isolate its states and transitions explicitly. I shall use a style of notation for presenting the transitions of the PM very close to that

used for the LM, though it may not succeed in clarity as well as in Appendix B since the operations of the PM are essentially more powerful than those of the LM.

This appendix outlines one possible s-expression syntax for Prolog programs, the code produced when compiling such programs, the machine state representation, and the state transitions. The appendix is not a complete description of a software realisation of the PM, but the listing of an AlgolW realisation is included. Some more details of the AlgolW PM are given in Chapter 6.

D.1 An s-expression syntax for Prolog.

For practical purposes Prolog is processed on the computer in the form of s-expressions. Each Prolog construct, in the notation of Chapter 3, is given a corresponding s-expression representation. This particular syntax is that in which the higher level interpreter of Chapter 7 expects to receive Prolog programs. The syntax is similar to that employed by the MicroProlog system implemented at Imperial College and described in McCabe (1980).

One trivial extension to the s-expression notation is employed below. This is the distinguishable symbolic atom consisting of a single apostrophe '. It is used as the "keyword" introducing a constant s-expression in an su-expression pattern. ' thus corresponds to QUOTE in the s-expression syntax of LIspkit, but I hope that it is less obtrusive as it is to be used within patterns.

In the s-expression forms a subexpression postfixed by an asterisk * must also be represented in s-expression form.

Each variable and predicate name is represented by the single symbolic atom formed by converting letters from lower to upper case, e.g. x, p are represented as X, P.

Su-expression patterns used as formal or actual arguments:
Variables: Represented by upper case form.
Constant su-expression (not containing variables): c is represented by ('.c) with the exception of NIL, which is allowed to represent itself, for convenience.
Constructed expressions: (x.y) is represented as (x*.y*).

Conditions calling defined predicates:
p(a1, ...) is represented as (p* a1* ...).

Conditions calling primitive predicates:

```
add (x,y,z) is represented by (ADD x* y* z*),
sub (x,y,z) is represented by (SUB x* y* z*),
mul (x,y,z) is represented by (MUL x* y* z*),
div (x,y,z) is represented by (DIV x* y* z*),
rem (x,y,z) is represented by (REM x* y* z*),
atom(x)     is represented by (ATOM x*),
eq  (x,y)   is represented by (EQ x* y*),
leq (x,y)   is represented by (LEQ x* y*).
```

Negated conditions:
¬c is represented by (NOT.c*).

Predicate definitions: Collecting together all the cases of predicate p, and naming the cases, stripped of the name p, by ca1, ca2, etc:

```
p( ... ) ←... or p ca1 is represented by (p* ca1* ca2* ... )
p( ...)  ←...    p ca2
   ...              ...
```

where each of the cases is represented as follows:

```
(a1, ...)←  is represented by ((a1* ... )),
(a1, ...)← c1, ... is represented by ((a1* ... )IF c1* ...).
```

The overall program structure, where the cases for the defined predicates have been collected together as p1 p2, etc:

```
query (a1, ... )←c1, ...
p1
p2 ...
```

is represented as

```
((QUERY (a1* ... ) IF c1* ...)
 WHERE
 p1* p2* ... )
```

For example, the naive reverse program:

```
query (l1,l2)← rev(l1, l2)
rev (NIL,NIL)←
rev ((x.l1), l2)← rev(l1,l3), append (l3, (x), l2)
append (NIL,l,l)←
append ((x.l1), l2, (x.l3)) ←append (l1, l2, l3)
```

is represented in the s-expression syntax as:

```
((QUERY (L1 L2) IF (REV L1 L2)) WHERE
(REV   ((NIL NIL))
       ((X.L1) L2) IF (REV L1 L3) (APPEND L3 (X) L2)))
(APPEND (( NIL L L ))
       ((( X.L1) L2 (X.L3)) IF (APPEND L1 L2 L3))))
```

D.2 Compiling Prolog programs for the PM.

The compilation process must take a Prolog program, which is a list of predicate definitions with a distinguished query predicate, and produce an s-expression machine code program for the PM which contains the compiled definitions and instructions to invoke the query predicate.

I shall give here, in an informal notation, the code skeletons generated for Prolog constructs. The compiled

form of syntactic items will be denoted by the use of a postfixed asterisk *. A vertical bar | indicates list concatenation.

When compiling a predicate case the compiler maintains an ordered list of the local variables used in the case. The indices required for ALDV and FLDV instructions are computed from this list.

A program consisting of a query case q and a list of predicate definitions p1, ... (each of which is a group of one or more cases):

```
((QUERY.q) WHERE p1 ...)* =
        ((INVOKE 0 HALT) (QUERY q)* p1* ...).
```

Note that the query has been changed to a single case predicate (QUERY q)* to be compiled.

A predicate definition with cases c1, c2, ... :

```
(p c1 c2 ...)* = (TRYCASE c1* TRYCASE c2* ... ENDP).
```

Individual cases without and with conditions:

```
((a1 a2 ...))* = (DCL m) | (a1 a2 ...)* | (UNIFY ENDC)
((a1 a2 ...) IF c1 c2 ...)* =
             (DCL m) | (a1 a2 ...)* | (UNIFY)|c1*|...|(ENDC)
```

where m is the number of local variables in the case, and (a1 a2 ...)* must be compiled as a formal argument list.

Formal argument list:

```
(a1 a2 ...)* = a1* | a2* | ... | (FLDC NIL FCONS ... FTOP)
```

where the number of FCONS instructions is equal to the number of arguments a1, a2, ...

Individual formal argument patterns:

```
Variables:  x* = (FLDV n) where n is the index of x in
                          the local environment
                          (numbered from zero).
```

Constant expressions c* = (FLDC c).
Constructed expressions (x.y)* = x* | y* | (FCONS).
Conditions, not negated and negated:
(p a1 ...)* = (a1 ...)* | p*
(NOT p a1 ...)* = (a1 ...)* | (NSUCCTRAP) | p* | (NFAIL)
where (a1 ...)* is compiled as an actual argument list,
and p* as a predicate name.

The predicate name in a condition, either a defined or
primitive predicate:

```
  defined* = (INVOKE n) where n is the index of the
                        predicate in the global list of
                        definitions (numbered from zero).
prim*      = (PRIM prim)where prim specifies the primitive
                        action to be performed.
```

Actual argument list:
(a1 a2 ...)* = a1* | a2* | ... | (ALDC NIL ACONS ...ATOP)
where the number of ACONS instructions is equal to the
number of arguments a1, a2, ...

Individual actual argument patterns:

```
Variables:  x* = (ALDV n) where n is the index of x in
                          the local environment
                          (numbered from zero).
```

Constant expressions: c* = (ALDC c).
Constructed expressions: (x.y)* = x* | y* | (ACONS).

For example, the naive reverse program from the previous
section compiles to give:

```
((INVOKE 0 HALT)
 (TRYCASE (DCL 2 FLDV 0 FLDV 1 FLDC NIL FCONS  FCONS FTOP
          UNIFY ALDV 0 ALDV 1 ALDC NIL ACONS ACONS ATOP
          INVOKE 1 ENDC)
ENDP)
(TRYCASE (DCL 0 FLDC NIL FLDC NIL FLDC NIL FCONS FCONS  FTOP
         UNIFY ENDC)
TRYCASE  (DCL 4 FLDV 0 FLDV 1 FCONS FLDV 2 FLDC NIL
```

```
            FCONS FCONS FTOP UNIFY
            ALDV 1 ALDV 3 ALDC NIL ACONS ACONS ATOP INVOKE 1
            ALDV 3 ALDV 0 ALDC NIL ACONS ALDV 2 ALDC NIL
            ACONS ACONS ACONS ATOP INVOKE 2 ENDC)
ENDP)
(TRYCASE   (DCL 1 FLDC NIL FLDV 0 FLDV 0 FLDC NIL
            FCONS FCONS FCONS FTOP UNIFY ENDC)
TRYCASE    (DCL 4 FLDV 0 FLDV 1 FCONS FLDV 2 FLDV 0 FLDV 3
            FCONS FLDC NIL FCONS FCONS FCONS FTOP UNIFY
            ALDV 1 ALDV 2 ALDV 3 ALDC NIL
            ACONS ACONS ACONS ATOP INVOKE 2 ENDC)
ENDP))
```

D.3 The Prolog Machine.

D.3.1 The registers.

At each step of the computation the entire machine state is held in 9 registers De, A,F,L,C,Du,R,B,N. These registers are the memory of the PM and contain program code, results, working storage, and all the information necessary for complex operations such as backtracking.

The value held by each register is an su-expression, though the contents of Du, and C will always be simply s-expressions. All operations of the machine are performed by constructing new su-expressions and by selecting sub-expressions. In the machine transitions described below the register values will be represented as su-expressions (with unknowns in the asterisk notation of Chapter 3) but with variables naming whole expressions and subexpressions in order to identify values before and after transitions.

The general roles of the 9 registers can be described as follows:

The De register ("Definitions") contains the s-expressions representing the predicates of the Prolog program. De is a list of lists. Each inner list is the code for one predicate, and is a list of the cases of the predicate. The first predicate, the head of De, is the query predicate. Predicates are accessed by an index computed during compilation - they are numbered from zero.

The A register ("Actuals") is used for the construction and processing of actual parameter lists.

The F register ("Formals") is used for the construction and processing of formal parameter lists.

The L register ("Locals") contains the environment of variable values which are local to the predicate case currently executing. Note that only the local variables are accessible directly. L is a list of su-expression values, one per variable, and the members are accessed by indices which are computed during compilation (variables are numbered from zero).

The C register ("Control") contains the sequence of instructions currently being executed. The head of C is the next instruction to be executed. New instruction sequences are loaded into C primarily at the invocation of defined predicates.

The Du register ("Dump") maintains a record of instruction sequences which have been suspended at the invocation of a defined predicate and of the corresponding local environments which must be restored when the sequences are resumed.

The B register ("Backtrack") maintains a record of the decision points as they are passed during program execution. B is used as a stack in which the most recent backtrack point is nearest to the head of B. At each decision point all necessary parts of the machine state to enable resumption are recorded.

The R register ("Restore"or "Reset") keeps an ordered list of the su-expressions which have been modified from an unknown since the most recent backtrack point was passed. R is a part of the information which is stacked on B as each new decision point is reached.

The N register ("Negation") is used in conjunction with B to handle the execution of negated conditions. N and B must cooperate for negated conditions as the success/ failure actions of these conditions must be reversed and B must be modified specially for their execution. N is a repository for machine states, including B, and behaves as a stack to cater for nested negations.

The collection of registers and roles which I have described is certainly adequate for the execution of Prolog. However, the number of registers (9) does seem rather large and possibly a more careful analysis would uncover a better factorisation of the roles.

D.3.2 The state transition rules.

There are three phases of execution to be described. Firstly the PM must be set up with the machine code representation of the program to be executed, and with the input su-expressions. Secondly the transitions which

accomplish the computation must be given. Finally the resultant su-expressions must be extracted from the final machine state.

Each state of the PM consists of an su-expression value for each of the 9 registers De, A, F, L, C, Du, R, B, and N. The value of De remains fixed once the computation has started, and it will not appear explicitly in the transitions. The states will be given as ordered octuples (A, F, L, C, Du, R,B, N) in which the items correspond to the appropriately named registers - however, to save space, registers which do not contribute to particular transitions will be omitted (though the delimiting commas will be retained).

Initially the input to the machine consists of an s-expression, prog, which is the machine language program, and a list of su-expressions, args, which are the arguments of the query predicate. Any unknowns in args are formed into a local environment, vars = (*1 *2 ...), which is treated as the local environment in which the condition invoking the query predicate is executed. prog is a pair (p.de) in which p is a short sequence of instructions which simply invokes the query predicate and then halts, and de is the list of compiled predicate definitions. Hence the PM receives(p.de) and args, derives vars from args, and the initial state of the machine is:

(args, NIL, vars, p, NIL, NIL, NIL, NIL)

and the value of De throughout the computation is de. (In fact the value of F is arbitrary, and I have set it to NIL).

The computation proceeds by executing PM instructions from the head of C. The PM has a repertoire is 18 basic instructions, represented by the mnemonics DCL, INVOKE, ENDC, TRYCASE, ENDP, UNIFY, ALDC, ALDV, ACONS. ATOP, FLDV, FLDC, FCONS, FTOP, NSUCCTRAP, NSUCC, NFAIL, HALT, and 8 variants of PRIM which implement the primitive predicates.

Each instruction selects amongst the following transition rules, in which the previous and next states are shown on the left and right of a right pointing arrow, →. In some transition rules it is necessary to show explicitly the modification of an su-expression by the elaboration of one or more unknowns. A phrase of the form a⇒b will be used to indicate that a must be <u>modified</u> to give b, and that a and b are the same physical data structure.

To create (DeCLare) a new local environment at entry to a predicate case:

(,, l,(DCL n.c),,,,)→(,,(*u1 *u2 ...),c,,,,)

where there are exactly n unknowns in the list (*u1 ...), and *u1, ... are chosen to be unknowns which are not already in use in the registers.

Instructions for building su-expressions as actual and formal arguments in the A and F registers (used as evaluation stacks):

(a,,,(ALDC x.c),,,,)→((x.a),,,c,,,,) (Load constant)

(a,,l,(ALDV n.c),,,,)→((x.a),,l,c,,,,) (Load variable)

where x is the (n+1)thitem in the local environment

l = ($\underbrace{...}_{n \text{ items}}$ x ...)

```
((x y.a),,, (ACONS.c),,,,)→(((y.x).a),,,c,,,,)
((x.a),,, (ATOP.c),,,,)→(x,,,c,,,,)
(,f,, (FLDC x.c),,,,)→(,(x.f),,c,,,,)
(,f,, (FLDV n.c),,,,)→(,(x.f),,c,,,,)
       where x is the (n+1)th item in the local
       environment   l = (... x ...)
                          n items

(,(x y.f),, (FCONS.c),,,,)→(,((y.x).f),,c,,,,)
(,(x.f),,(FTOP.c),,,,)→(,x,,c,,,,)
```

Invoking primitive predicates:

```
((x y z),,,(PRIM ADD.c),,,,)→depending on x,y,z as follows
```

Cases:

```
x,y,z all numbers, x+y=z →((x y z),,,c,,,,)
x,y,z all numbers, x+y≠z→Fail, so backtrack (see below)
One of x,y,z unknown, e.g. x=*n→((x' y z),,,c,,,,)
      where *n⇒(z-y) and so x⇒x'
Two or three of x,y,z unknown→Terminate computation
with an error.

((x y z),,,(PRIM SUB.c),,,,)→depending on x,y,z in
a way analogous to ADD
((x y z),,,(PRIM MUL.c),,,,)→depending on x,y,z as
follows:
Cases:
x,y,z all numbers, x*y=z →((x y z),,c,,,,)
x,y,z all numbers, x*y≠z →Fail, so backtrack (see below)
z unknown, z=*n→((x y z'),,c,,,,)
        where *n⇒x*y and so z⇒ z'

x or y unknown, e.g. x=*n:
        y divides z →((x' y z),,,c,,,,)
               where *n⇒(z div y)and so x⇒x'
        y does not divide z→Fail, so backtrack (see below)
Two or more of x,y,z unknown→Terminate computation with
                              an error.
```

((x y z),,,(PRIM DIV.c),,,,) ➔ depending on x,y,z as follows

Cases:

x,y,z all numbers, x div y = z ➔((x y z),,,c,,,,)

x,y,z all numbers, x div y ≠ z ➔Fail, so backtrack (see below)

z unknown, z = *n ➔((x y z'),,,c,,,,)
where *n ⇒ (x div y) and so z ⇒ z'

x or y unknown ➔ Terminate computation with an error

((x y z),,,(PRIM REM.c),,,,) ➔depending on x,y,z as follows

Cases:

x,y,z numbers, x rem y=z ➔((x y z),,,c,,,,)

x,y,z numbers, x rem y≠z ➔Fail, so backtrack (see below)

z unknown, z=*n ➔((x y z'),,,c,,,,)
where *n ⇒(x rem y) and so z ⇒ z'

x or y unknown➔Terminate computation with an error.

((x),,,(PRIM ATOM.c),,,,) ➔depending on x as follows

Cases:

x is symbol or number ➔((x),,,c,,,,)

x is constructed, (y.z)➔Fail, so backtrack (See below)

x is unknown➔Terminate computation with an error.

((x y),,,(PRIM EQ.c),,,,) ➔depending on x,y as follows

Cases:

x and y both symbols or numbers and x=y
➔((x y),,,c,,,,)

x and y different values, or either is constructed
➔Fail, so backtrack (See below)

x or y unknown ➔Terminate computation with an error.

((x y),,,(PRIM LEQ.c),,,,)➔depending on x,y as follows

Cases:

x,y numbers, x ≤ y → ((x y),,,c,,,,)

x,y numbers, x > y → Fail, so backtrack (See below)

x or y unknown → Terminate computation with an error.

Invoking a defined predicate:

(,,1,(INVOKEn.c),du,,,) → (,,1,cp,(1 c.du),,,)

where cp is the (n+1)th predicate definition in De

De = (... cp ...)
n items

Selecting a case within a predicate and recording the decision point:

(a,,,(TRYCASE case.c),du,r,b,) → (a,,,case,du,NIL,(r a c du.b),)

If there are no more cases:

(,,,(ENDP),,,(x.b),) → The predicate fails, so backtrack to the most recent decision point on B (See below)

(,,,(ENDP),,, NIL,) → No remaining decision points, so computation terminates without a solution.

Unify actual and formal arguments:

(a,f,,(UNIFY.c),,r,,) → depending on a, f as follows

Cases:

a and f cannot be unified → Fail, so backtrack (See below)

a and f are unified successfully (See below)

→ (a',f',,c,,(u1 u2 ... .r),,)

where a' and f' are the elaborated forms of a and f, and u1, u2, ... (in the new value of R) are the elaborated forms of the unknowns in a and f which were modified in order to achieve the match.

Return successfully from a case, having satisfied all the conditions:

(,,l1,(ENDC),(l2 c.du),,,) → (,,l2,c,du,,,)

Successfully complete a computation:

(a,,l,(HALT),,,,) → Output the result - either the elaborated actual arguments a, or simply the elaborated unknowns l (corresponding to vars at initialisation). If further solutions are required then the computation can be resumed by backtracking (see below).

Before describing the instructions for handling negated conditions it is appropriate to give the transition which accomplishes backtracking, and to show how unification may be performed.

In order to backtrack to the most recent decision point, the machine state recorded in the first four items of B must be restored. That itself is straightforward, but in addition any unknowns which have been elaborated since the decision point must be reset to their unknown state. These elaborations are all recorded in R, and the resets are achieved by modifying each item in the list R to an unknown. Hence:

Firstly the resets:

R = (x y ...) ⇒ (*m *n ...)

where the unknown *m, *n, ... are not already in use in the machine.

And secondly the transition:

(a1,,,c1,du1,r1,(r2 a2 c2 du2.b),) → (a2,,,c2,du2,r2,b,)

The process of unification must achieve two results. Firstly, it must attempt to match the values in the A and F registers by finding a (minimal) set of modifications for unknowns in A and F (either succeeding or failing in this attempt), and secondly any modifications made during the attempt must be recorded in the R register. A simple way to perform the matching is to scan the A and F su-expressions (both are binary trees) in parallel and in prefix order. When an unknown leaf is encountered in either register it is elaborated to the corresponding subtree of the other register, and the subtrees are skipped as they are then known to be equal. The unification fails if a mismatch is found, or is successful if the scan completely covers the structures without mismatch. To give this process a more concrete form I shall use a register transition notation. Unification requires two extra registers, U1 and U2, and the transitions will be between states involving these and R, (U1, U2,R). From a machine state

(a,f,,(UNIFY.c),,r,,)

unification commences in the state ((a.NIL),(f.NIL),r), and the values a and f are gradually decomposed and compared.

The unification transitions are as follows:

(NIL, NIL, r) → Unification succeeds.

((x.u1),(y.u2),r) with x,y atoms of same value, or identical unknowns → (u1,u2,r)

((x.u1),(y.u2),r) with x,y unequal atoms or x atom, y constructed or y atom, x constructed → Unification fails

```
(((x1.x2).u1),((y1.y2).u2),r)→((x1 x2.u1),(y1 y2.u2),r)
((x.u1),(y.u2),r) with x unknown, x=*n,
                    and y atom or constructed
                  →(u1,u2,(x'.r))
                    and modify  *n⇒y and so x⇒x'
((x.u1),(y.u2),r) with y unknown, y=*n,
                    and x atom, constructed or unknown
                  →(u1, u2, (y'.r))
                    and modify *n⇒x and so y⇒y'
```

Note that this unification algorithm does not incorporate the occur check (see Robinson (1965), Warren (1977)). The check would have to be included in the final two transitions in order to ensure that the unknown which is to be modified is not also a leaf of the value with which it is being matched. Omission of the check simply means that the algorithm will be badly behaved for some rather unlikely programs.

The final machine transitions to be described are those for the instructions handling the execution of negated conditions.

Preparing to trap the backtracking of a condition which fails, in order to turn it into success:

```
(a,,l,(NSUCCTRAP p1 p2 NFAIL.c),du,r,b,ns)
        →(a,,l,(p1 p2 NFAIL.c),du,NIL,(NIL NIL (NSUCC)NIL),
                                (w l c r du b.ns))
          where p1, p2 are INVOKE, n or PRIM,op
          and w is a list (*m *n ...) of the unknowns
          in the actual arguments a.
```

Changing a negated condition failure into success, following backtracking, by restoring state from N:

```
(,,l1,(NSUCC),du1,r1,b1,(w l2 c2 r2 du2 b2.ns))
              → ( ,,l2,c2,du2,r2,b2,ns)
```

Note that the NSUCC instruction never appears explicitly in program code. It can only be executed after having been inserted into B as an artificial decision point by the NSUCCTRAP instruction.

Changing the success of a negated condition into either failure, or termination with an error (if any unknowns in the actual arguments have been bound):

```
(,,,(NFAIL.c1),,r1,b1,(w l2 c2 r2 du2 b2.ns))
              →depending on w as follows:
```

Cases:

All items in the list w are still unknowns

```
→(,,,(NFAIL.c1),,r2,b2,ns)
```

and then require failure, so backtrack,

One or more items in w have been elaborated

→Terminate the computation with an error.

D.4 An AlgolW software realisation of the PM

The following pages show one possible implementation of the PM, in AlgolW. This particular machine was used to perform the experiments reported in Chapters 6 and 7.

```
0000 1-     BEGIN   %*** PROLOG MACHINE - WITH NEGATION AND ARITHMETIC PRIMITIVES ***%
0001 --
0001 --     INTEGER PAGES,NO_OF_ELEMS,NO_OF_SYMBS,
0001 --             FIRST_ELEM,LAST_ELEM,
0001 --             FIRST_SYMB,LAST_SYMB;
0002 --     WRITE("NO. OF PAGES, NO. OF CONSES/NUMBS/UNKS ?"); IOCONTROL(2);
0004 --     READ( PAGES,NO_OF_ELEMS );
0005 --     IF (340*(PAGES-9))<NO_OF_ELEMS THEN
0005 2-     BEGIN NO_OF_ELEMS:=340*(PAGES-9);
0007 --        WRITE("NO. OF CONSES/NUMBS/UNKS TOO GREAT. REDUCED TO ",NO_OF_ELEMS);
0008 -2     END
0008 --     ELSE WRITE("NO. OF CONSES/NUMBS/UNKS : ",NO_OF_ELEMS); IOCONTROL(2);
0010 --     NO_OF_SYMBS:=2000; %9P OF SYMBS%
0011 --     FIRST_ELEM:=1; LAST_ELEM:=FIRST_ELEM+NO_OF_ELEMS-1;
0013 --     FIRST_SYMB:=LAST_ELEM+1; LAST_SYMB:=FIRST_SYMB+NO_OF_SYMBS-1;
0015 --
0015 2-     BEGIN
0016 --        INTEGER CONSTYPE,SYMBTYPE,NUMBTYPE,INDIRECTTYPE,
0016 --                UNKNOWN,UNKNOWN_VARS,NEXT_ELEM,NEXT_SYMB,NIL;
0017 --
0017 --        INTEGER ARRAY CAR,CDR(FIRST_ELEM::LAST_ELEM);
0018 --        INTEGER ARRAY FLAGS(FIRST_ELEM::LAST_SYMB);
0019 --        STRING(12) ARRAY SVAL(FIRST_SYMB::LAST_SYMB);
0020 --
0020 --        %ELEMENT REPRESENTATIONS:
0020 --           CONS : FLAGS=1, CAR & CDR.
0020 --           SYMB : FLAGS=2, STRING IN SVAL.
0020 --           NUMB : FLAGS=3, INTEGER VALUE IN CAR.
0020 --           UNKNOWN : FLAGS=4, VARIABLE'S SUBSCRIPT IS IN CDR.
0020 --           INDIRECT : FLAGS=5, CAR POINTS TO INDIRECT OBJECT,
0020 --                               CDR STILL CONTAINS SUBSCRIPT.
0020 --         ***NOTE: ARITHMETIC PRIMITIVES AND UNIFICATION RELY ON THESE FLAG CODES.
0020 --         FLAG IS -VE FOR "MARKED" DURING GARBAGE COLLECTION
0020 --        %
0020 --

0020 --        PROCEDURE INIT_LIST_STORAGE;
0021 3-        BEGIN CONSTYPE:=1; SYMBTYPE:=2; NUMBTYPE:=3;
0025 --           UNKNOWN:=4; INDIRECTTYPE:=5; UNKNOWN_VARS:=0;
0028 --           NEXT_ELEM:=FIRST_ELEM;
0029 --           NEXT_SYMB:=FIRST_SYMB;  NIL:=SYMB("NIL");
0031 --           FOR I:=FIRST_ELEM UNTIL LAST_ELEM DO CAR(I):=I+1;
0032 -3        END;
0033 --

0033 --        INTEGER PROCEDURE SIMPLECONS(INTEGER VALUE NEWCAR,NEWCDR);
0034 3-        BEGIN INTEGER TEMP;        %ONLY USED IN GETEXP%
0036 --           IF NEXT_ELEM>LAST_ELEM THEN
0036 4-              BEGIN WRITE("CONS/NUMB/UNK SPACE OVERFLOW DURING SU-EXP INPUT");
0038 --                 ASSERT FALSE;
0039 -4              END;
0040 --           TEMP:=NEXT_ELEM;  NEXT_ELEM:=CAR(NEXT_ELEM);
0042 --           CAR(TEMP):=NEWCAR;  CDR(TEMP):=NEWCDR;  FLAGS(TEMP):=CONSTYPE;
0045 --           TEMP
0045 -3        END;
0046 --

0046 --        INTEGER PROCEDURE CONS(INTEGER VALUE NEWCAR,NEWCDR);
0047 3-        BEGIN INTEGER TEMP;
0049 --           IF NEXT_ELEM>LAST_ELEM THEN GARBAGE_COLLECT;
0050 --           TEMP:=NEXT_ELEM;  NEXT_ELEM:=CAR(NEXT_ELEM);
0052 --           CAR(TEMP):=NEWCAR;  CDR(TEMP):=NEWCDR;  FLAGS(TEMP):=CONSTYPE;
0055 --           TEMP
0055 -3        END;
0056 --

0056 --        LOGICAL PROCEDURE ISCONS(INTEGER VALUE PTR);
0057 --        FIRST_ELEM<=PTR AND PTR<=LAST_ELEM AND ABS(FLAGS(PTR))=CONSTYPE;
0058 --

0058 --        INTEGER PROCEDURE SIMPLENUMB(INTEGER VALUE NEWNUMB);
0059 3-        BEGIN INTEGER TEMP;  %ONLY USED IN GETEXP%
0061 --           IF NEXT_ELEM>LAST_ELEM THEN
0061 4-              BEGIN WRITE("CONS/NUMB/UNK SPACE OVERFLOW DURING SU-EXP INPUT");
0063 --                 ASSERT FALSE;
0064 -4              END;
0065 --           TEMP:=NEXT_ELEM;  NEXT_ELEM:=CAR(NEXT_ELEM);
0067 --           CAR(TEMP):=NEWNUMB; FLAGS(TEMP):=NUMBTYPE;
0069 --           TEMP
0069 -3        END;
0070 --

0070 --        INTEGER PROCEDURE NUMB(INTEGER VALUE NEWNUMB);
0071 3-        BEGIN INTEGER TEMP;
0073 --           IF NEXT_ELEM>LAST_ELEM THEN GARBAGE_COLLECT;
0074 --           TEMP:=NEXT_ELEM;  NEXT_ELEM:=CAR(NEXT_ELEM);
0076 --           CAR(TEMP):=NEWNUMB;  FLAGS(TEMP):=NUMBTYPE;
0078 --           TEMP
0078 -3        END;
0079 --

0079 --        LOGICAL PROCEDURE ISNUMB(INTEGER VALUE PTR);
0080 --        FIRST_ELEM<=PTR AND PTR<=LAST_ELEM AND ABS(FLAGS(PTR))=NUMBTYPE;
0081 --

0081 --        INTEGER PROCEDURE SYMB(STRING(12) VALUE NEWSYMB);
0082 3-        BEGIN INTEGER TEMP;
0084 --           IF NEXT_SYMB>LAST_SYMB THEN
0084 4-              BEGIN WRITE("SYMBOL SPACE OVERFLOW");
0086 --                 ASSERT FALSE;
0087 -4              END;
0088 --           TEMP:=NEXT_SYMB;  NEXT_SYMB:=NEXT_SYMB+1;
0090 --           SVAL(TEMP):=NEWSYMB; FLAGS(TEMP):=SYMBTYPE;
0092 --           TEMP
0092 -3        END;
0093 --

0093 --        LOGICAL PROCEDURE ISSYMB(INTEGER VALUE PTR);
0094 --        FIRST_SYMB<=PTR AND PTR<=LAST_SYMB;
```

```
0095 --          INTEGER PROCEDURE SIMPLEUNK;
0096 3-          BEGIN INTEGER TEMP;
0098 --             IF NEXT_ELEM>LAST_ELEM THEN
0098 4-                BEGIN WRITE("CONS/NUME/UNK SPACE OVERFLOW DURING SU-EXP INPUT");
0100 --                   ASSERT FALSE;
0101 -4                END;
0102 --             UNKNOWN_VARS:=UNKNOWN_VARS+1;
0103 --             TEMP:=NEXT_ELEM;  NEXT_ELEM:=CAR(NEXT_ELEM);
0105 --             CDR(TEMP):=UNKNOWN_VARS;  FLAGS(TEMP):=UNKNOWN;
0107 --             TEMP
0107 -3          END;
0108 --

0108 --          INTEGER PROCEDURE UNK;
0109 3-          BEGIN INTEGER TEMP;
0111 --             IF NEXT_ELEM>LAST_ELEM THEN GARBAGE_COLLECT;
0112 --             UNKNOWN_VARS:=UNKNOWN_VARS+1;
0113 --             TEMP:=NEXT_ELEM;  NEXT_ELEM:=CAR(NEXT_ELEM);
0115 --             CDR(TEMP):=UNKNOWN_VARS;  FLAGS(TEMP):=UNKNOWN;
0117 --             TEMP
0117 -3          END;
0118 --

0118 --          LOGICAL PROCEDURE ISUNK(INTEGER VALUE UNK);
0119 --          FIRST_ELEM<=UNK AND UNK<=LAST_ELEM AND ABS(FLAGS(UNK))=UNKNOWN;
0120 --

0120 --          PROCEDURE CHANGETOUNK(INTEGER VALUE NEWUNK);
0121 --             FLAGS(NEWUNK):=UNKNOWN;
0122 --

0122 --          LOGICAL PROCEDURE ISINDIRECT(INTEGER VALUE IND);
0123 --          FIRST_ELEM<=IND AND IND<=LAST_ELEM AND ABS(FLAGS(IND))=INDIRECTTYPE;
0124 --

0124 --          PROCEDURE CHANGETOINDIRECT(INTEGER VALUE NEWIND,ITEM);
0125 3-          BEGIN FLAGS(NEWIND):=INDIRECTTYPE;
0127 --                CAR(NEWIND):=ITEM;
0128 --                R:=CONS(NEWIND,R);
0129 -3          END;
0130 --

0130 --          INTEGER PROCEDURE FORCE(INTEGER VALUE ITEM);
0131 3-          BEGIN
0132 --             WHILE FLAGS(ITEM)=INDIRECTTYPE DO ITEM:=CAR(ITEM);
0133 --             ITEM
0133 -3          END;
0134 --

0134 --          LOGICAL PROCEDURE ISNIL(INTEGER VALUE X);
0135 --          FIRST_SYMB<=X AND X<=LAST_SYMB AND SVAL(X)="NIL";
```



```
0136 --          INTEGER ARRAY STACK(1::1000);  INTEGER ST_PTR;
0138 --

0138 --          PROCEDURE GARBAGE_COLLECT;
0139 3-          BEGIN
0140 --             MARK_FROM(DE); MARK_FROM(A); MARK_FROM(F);
0143 --             MARK_FROM(L); MARK_FROM(C); MARK_FROM(R);
0146 --             MARK_FROM(DU); MARK_FROM(E); MARK_FROM(N);
0149 --             MARK_FROM(NEGTRAP); MARK_FROM(W);
0151 --
0151 --             NEXT_ELEM:=LAST_ELEM+1;
0152 --             FOR I:=FIRST_ELEM UNTIL LAST_ELEM DO
0152 --               IF FLAGS(I)<0 THEN FLAGS(I):=-FLAGS(I)
0152 4-                            ELSE BEGIN CAR(I):=NEXT_ELEM;
0154 -4                                       NEXT_ELEM:=I;  END;
0156 --
0156 --             IF NEXT_ELEM>LAST_ELEM THEN
0156 4-                BEGIN WRITE("CONS/NUME/UNK SPACE OVERFLOW");
0158 -4                   ASSERT FALSE;  END;
0160 --
0160 -3          END GARBAGE_COLLECT;
0161 --

0161 --          PROCEDURE MARK_FROM(INTEGER VALUE PTR);
0162 3-          BEGIN ST_PTR:=2;   STACK(1):=PTR;
0165 --
0165 --             WHILE ST_PTR¬=1 DO
0165 4-             BEGIN ST_PTR:=ST_PTR-1;   PTR:=STACK(ST_PTR);
0168 --                WHILE (ISINDIRECT(PTR) OR ISCONS(PTR)) AND FLAGS(PTR)>0 DO
0168 --                 IF ISCONS(PTR)
0168 5-                THEN BEGIN FLAGS(PTR):=-CONSTYPE;
0170 --                        IF ST_PTR>1000 THEN
0170 6-                           BEGIN WRITE("GARBAGE COLLECTOR STACK OVERFLOW");
0172 --                              ASSERT FALSE;
0173 -6                           END;
0174 --                        STACK(ST_PTR):=CDR(PTR); ST_PTR:=ST_PTR+1;
0176 --                        PTR:=CAR(PTR);
0177 -5                     END
0177 55               ELSE BEGIN FLAGS(PTR):=-INDIRECTTYPE; PTR:=CAR(PTR); END;
0181 --                IF ISNUMB(PTR) THEN FLAGS(PTR):=-NUMBTYPE ELSE
0181 --                IF ISUNK(PTR) THEN FLAGS(PTR):=-UNKNOWN;
0182 -4             END;
0183 --
0183 -3          END MARK_FROM;
```

```
0184 --          STRING(12) TOKEN,TYPE;
0185 --

0185 --          PROCEDURE SCAN;
0186 33          BEGIN GETTOKEN(TOKEN,TYPE); IF TYPE="ENDFILE" THEN TOKEN:=")" END;
0189 --

0189 --          PROCEDURE GETEXP(INTEGER RESULT E);
0190 --          IF TOKEN="*"
0190 33          THEN BEGIN SCAN; E:=QUERYVAR(TOINTEGER(TOKEN)); SCAN; END
0194 --          ELSE IF TOKEN="("
0194 33              THEN BEGIN SCAN; GETEXPLIST(E); ASSERT TOKEN=")"; SCAN END
0198 --              ELSE IF TYPE="NUMERIC"
0198 33                  THEN BEGIN E:=SIMPLENUME(TOINTEGER(TOKEN)); SCAN END
0200 33                  ELSE BEGIN E:=SYME(TOKEN); SCAN END;
0203 --

0203 --          INTEGER PROCEDURE QUERYVAR(INTEGER VALUE I);
0204 3-          BEGIN INTEGER VARS;
0206 --              IF ISNIL(QUERYVARIABLES)
0206 44                  THEN BEGIN W:=SIMPLEUNK; QUERYVARIABLES:=SIMPLECONS(W,NIL); END;
0210 --              VARS:=QUERYVARIABLES;
0211 --              WHILE I>1 DO
0211 4-              BEGIN IF ISNIL(CDR(VARS))
0212 55                      THEN BEGIN W:=SIMPLEUNK; CDR(VARS):=SIMPLECONS(W,NIL); END;
0216 --                  VARS:=CDR(VARS);  I:=I-1;
0218 -4              END;
0219 --
0219 --              CAR(VARS)
0219 -3          END QUERYVAR;
0220 --

0220 --          PROCEDURE GETEXPLIST(INTEGER RESULT E);
0221 3-          BEGIN E:=SIMPLECONS(0,0); GETEXP(CAR(E));
0224 44              IF TOKEN="." THEN BEGIN SCAN; GETEXP(CDR(E)) END ELSE
0226 -3              IF TOKEN=")" THEN CDR(E):=NIL ELSE GETEXPLIST(CDR(E)) END;
0227 --

0227 --          PROCEDURE PUTEXP(INTEGER VALUE E);
0228 3-          BEGIN E:=PUTFORCE(E);
0230 --              IF ISUNK(E) THEN PUTTOKEN(VARNAME(CDR(E))) ELSE
0230 --              IF ISSYMB(E) THEN PUTTOKEN(SVAL(E)) ELSE
0230 --              IF ISNUMB(E) THEN PUTTOKEN(TOSTRING(CAR(E))) ELSE
0230 4-              BEGIN PUTTOKEN("(");
0232 --                  WHILE ISCONS(E) DO
0232 55                      BEGIN PUTEXP(CAR(E)); E:=PUTFORCE(CDR(E)) END;
0235 --                  IF ISNIL(E) THEN ELSE
0235 55                      BEGIN PUTTOKEN("."); PUTEXP(E) END;
0238 --                  PUTTOKEN(")");
0239 -4              END;
0240 -3          END;
0241 --

0241 --          INTEGER PROCEDURE PUTFORCE(INTEGER VALUE IND);
0242 3-          BEGIN WHILE ISINDIRECT(IND) DO IND:=CAR(IND);
0244 --              IND
0244 -3          END;
0245 --


0245 --          INTEGER PROCEDURE TOINTEGER(STRING(12)VALUE T);
0246 3-          BEGIN STRING(1)D; INTEGER I,S; LOGICAL NEG; S:=0;
0251 44              IF T(0|1)="-" THEN BEGIN NEG:=TRUE; I:=1 END
0253 44              ELSE BEGIN NEG:=FALSE; I:=0 END; D:=T(I|1);
0257 --              WHILE D¬=" " DO
0257 4-                  BEGIN S:=10*S+(DECODE(D)-DECODE("0")); I:=I+1;
0260 --                      IF I>11 THEN D:=" " ELSE D:=T(I|1);
0261 -4                  END;
0262 --              IF NEG THEN -S ELSE S
0262 -3          END;
0263 --

0263 --          STRING(12) PROCEDURE TOSTRING(INTEGER VALUE I);
0264 --          IF I=0 THEN "0" ELSE
0264 3-          BEGIN STRING(12)T; LOGICAL NEG; INTEGER L;
0268 --              NEG:=I<0; T:=" ";
0270 --              I:=ABS(I);L:=2+TRUNCATE(LOG(I+.1));
0272 --              FOR J:= L-1 STEP -1 UNTIL 1 DO
0272 44                  BEGIN T(J|1):=CODE(DECODE("0")+I REM 10); I:=I DIV 10 END;
0275 --              IF NEG THEN T(0|1):="-"; T
0276 -3          END;
0277 --

0277 --          STRING(12) PROCEDURE VARNAME(INTEGER VALUE I);
0278 3-          BEGIN STRING(12) T1,T2; T1:=TOSTRING(I); T2:="*";
0282 --              IF T1="0" THEN T2:="*0" ELSE T2(1|9):=T1(1|9);
0283 --              T2
0283 -3          END;
0284 --

0284 --          PROCEDURE INIT_SYNTAX; SCAN;
```

ALGOL W : PROLOG.MACHINE   INPUT/OUTPUT OF TOKENS   1981 JUNE

```
0286 --         STRING(1)ARRAY INBUFFER(1::256); INTEGER INBUFPTR,INBUFEND;
0288 --         LOGICAL EOF;
0289 --

0289 --         PROCEDURE GETCHAR;
0290 3-         BEGIN IF INBUFPTR>INBUFEND THEN
0291 4-                 BEGIN GETLINE(INBUFFER,INBUFEND,EOF);
0293 -4                     INBUFPTR:=1 END;
0294 --             CH:=INBUFFER(INBUFPTR); INBUFPTR:=INBUFPTR+1;
0296 -3         END;
0297 --
0297 --         STRING(1)CH;
0298 --

0298 --         PROCEDURE GETTOKEN(STRING(12)RESULT TOKEN,TYPE);
0299 3-         BEGIN TOKEN:=" "; WHILE¬EOF AND CH=" " DO GETCHAR;
0302 --             IF EOF THEN TYPE:="ENDFILE" ELSE
0302 --             IF "0"<=CH AND CH<="9" OR CH="-" THEN
0302 4-                 BEGIN INTEGER I; TYPE:="NUMERIC";
0305 --                     TOKEN(0|1):=CH; I:=1; GETCHAR;
0308 --                     WHILE "0"<=CH AND CH<="9" DO
0308 55                        BEGIN ASSERT I<12; TOKEN(I|1):=CH; I:=I+1; GETCHAR END
0312 -4                 END ELSE
0312 --             IF "A"<=CH AND CH<="Z" THEN
0312 4-                 BEGIN INTEGER I; TYPE:="ALPHANUMERIC";
0315 --                     TOKEN(0|1):=CH; I:=1; GETCHAR;
0318 --                     WHILE "A"<=CH AND CH <="9" DO
0318 5-                        BEGIN
0319 --                            ASSERT I<12; TOKEN(I|1):=CH; I:=I+1; GETCHAR
0322 -5                        END
0322 -4                 END ELSE
0322 4-             BEGIN IF CH="*" THEN TYPE:="STAR" ELSE TYPE:="DELIMITER";
0324 -3                 TOKEN(0|1):=CH; GETCHAR END END;
0326 --
0326 --         STRING(1)ARRAY OUTBUFFER(1::80); INTEGER OUTBUFPTR;
0328 --

0328 --         PROCEDURE PUTCHAR(STRING(1)VALUE C);
0329 3-         BEGIN IF OUTBUFPTR=80 THEN FORCELINEOUT;
0331 -3            OUTBUFPTR:=OUTBUFPTR+1; OUTBUFFER(OUTBUFPTR):=C END;
0333 --

0333 --         PROCEDURE FORCELINEOUT;
0334 33         BEGIN PUTLINE(OUTBUFFER,OUTBUFPTR); OUTBUFPTR:=0 END;
0337 --

0337 --         PROCEDURE PUTTOKEN(STRING(12)VALUE T);
0338 3-         BEGIN INTEGER LEN; LEN:=12;
0341 --            WHILE LEN>0 AND T(LEN-1|1)=" " DO LEN:=LEN-1;
0342 -3            FOR I:=0 UNTIL LEN-1 DO PUTCHAR(T(I|1)); PUTCHAR(" ") END;
0344 --

0344 --         PROCEDURE INIT_LEXICAL;
0345 3-         BEGIN GETLINE(INBUFFER,INBUFEND,EOF); INBUFPTR:=1; GETCHAR;
0349 -3            OUTBUFPTR:=0 END;
```

ALGOL W : PROLOG.MACHINE   INPUT/OUTPUT OF LINES   1981 JUNE

```
0350 --         PROCEDURE GET(STRING(1)ARRAY LINE(*);INTEGER RESULT LEN;
0352 --                                              INTEGER VALUE MOD,LNUM,DEV);
0353 --         FORTRAN "READ";
0354 --

0354 --         PROCEDURE PUT(STRING(1)ARRAY LINE(*); INTEGER VALUE LEN,MOD,
0355 --                                    LNUM,DEV);
0356 --         FORTRAN "WRITE";
0357 --

0357 --         PROCEDURE GETLINE(STRING(1)ARRAY LINE(*);INTEGER RESULT LEN;
0359 --                                              LOGICAL RESULT EOF);
0360 3-         BEGIN LINE(1):=" "; GET(LINE,LEN,0,0,0);
0363 --            EOF :=R_CODE>0;
0364 --            LEN:=NUMBER(BITSTRING(LEN) SHR 16);
0365 -3         END;
0366 --

0366 --         PROCEDURE PUTLINE(STRING(1)ARRAY LINE(*); INTEGER VALUE LEN);
0368 --         PUT(LINE,NUMBER(BITSTRING(LEN) SHL 16),0,0,1);
```

```
0369 --          INTEGER ARRAY UASTACK,UFSTACK(1::1000);  INTEGER USTK_PTR;

0371 --          LOGICAL PROCEDURE UNIFY;
0372 3-          BEGIN INTEGER CARA,CARF; INTEGER C;  LOGICAL FLAG;
0376 --
0376 --             FLAG:=TRUE;  USTK_PTR:=1;  UASTACK(1):=A;  UFSTACK(1):=F;
0380 --
0380 --             WHILE FLAG AND USTK_PTR¬=0 DO
0380 4-             BEGIN CARA:=UASTACK(USTK_PTR); CARF:=UFSTACK(USTK_PTR);
0383 --                IF CARA=CARF THEN USTK_PTR:=USTK_PTR-1 ELSE
0383 5-                BEGIN C:=FLAGS(CARA)+(FLAGS(CARF)-1)*4;
0385 --                   CASE C OF
0385 6-                   BEGIN
0386 7-                   BEGIN COMMENT CONSA CONSF;
0387 --                         IF USTK_PTR=1000 THEN
0387 08                            BEGIN WRITE("UNIFICATION STACK OVERFLOW"); ASSERT FALSE; END;
0391 --                         UASTACK(USTK_PTR):=FORCE(CDR(CARA));
0392 --                         UASTACK(USTK_PTR+1):=FORCE(CAR(CARA));
0393 --                         UFSTACK(USTK_PTR):=FORCE(CDR(CARF));
0394 --                         UFSTACK(USTK_PTR+1):=FORCE(CAR(CARF));
0395 -7                        USTK_PTR:=USTK_PTR+1;    END;
0397 --                   FLAG:=FALSE; COMMENT SYMEA CONSF;
0398 --                   FLAG:=FALSE; COMMENT NUMEA CONSF;
0399 7-                   BEGIN COMMENT UNKA CONSF;
0400 --                         CHANGETOINDIRECT(CARA,CARF);
0401 -7                        USTK_PTR:=USTK_PTR-1; END;
0403 --                   FLAG:=FALSE; COMMENT CONSA SYMBF;
0404 --                   IF SVAL(CARA)¬=SVAL(CARF) THEN FLAG:=FALSE COMMENT SYMBA SYMBF;
0404 --                                              ELSE USTK_PTR:=USTK_PTR-1;
0405 --                   FLAG:=FALSE; COMMENT NUMEA SYMBF;
0406 7-                   BEGIN COMMENT UNKA SYMEF;
0407 --                         CHANGETOINDIRECT(CARA,CARF);
0408 -7                        USTK_PTR:=USTK_PTR-1; END;
0410 --                   FLAG:=FALSE; COMMENT CONSA NUMBF;
0411 --                   FLAG:=FALSE; COMMENT SYMEA NUMBF;
0412 --                   IF CAR(CARA)¬=CAR(CARF) THEN FLAG:=FALSE COMMENT NUMBA NUMBF;
0412 --                                             ELSE USTK_PTR:=USTK_PTR-1;
0413 7-                   BEGIN COMMENT UNKA NUMEF;
0414 --                         CHANGETOINDIRECT(CARA,CARF);
0415 -7                        USTK_PTR:=USTK_PTR-1; END;
0417 7-                   BEGIN COMMENT CONSA UNKF;
0418 --                         CHANGETOINDIRECT(CARF,CARA);
0419 -7                        USTK_PTR:=USTK_PTR-1; END;
0421 7-                   BEGIN COMMENT SYMEA UNKF;
0422 --                         CHANGETOINDIRECT(CARF,CARA);
0423 -7                        USTK_PTR:=USTK_PTR-1; END;
0425 7-                   BEGIN COMMENT NUMEA UNKF;
0426 --                         CHANGETOINDIRECT(CARF,CARA);
0427 -7                        USTK_PTR:=USTK_PTR-1; END;
0429 7-                   BEGIN COMMENT UNKA UNKF;
0430 --                         CHANGETOINDIRECT(CARF,CARA);
0431 -7                        USTK_PTR:=USTK_PTR-1; END;
0433 -6                   END;
0434 -5                END;
0435 -4             END UNIFICATION_CYCLE;
0436 --
0436 --             FLAG  %SUCCESS OR FAILURE RESULT%
0436 -3          END UNIFY_PROCEDURE;
```



```
0437 --          PROCEDURE RESTORE(INTEGER VALUE INFO);
0438 3-          WHILE ¬ISNIL(INFO) DO BEGIN CHANGETOUNK(CAR(INFO));
0440 -3                                      INFO:=CDR(INFO); END;
0442 --
0442 --          PROCEDURE POPBACK;
0443 3-          BEGIN RESTORE(R); R:=CAR(B); B:=CDR(B);
0447 --                A:=CAR(B); B:=CDR(B);
0449 --                C:=CAR(B); B:=CDR(B); DU:=CAR(B); B:=CDR(B);
0453 -3          END;
0454 --
0454 --          PROCEDURE NOTEVARIABLES;
0455 3-          BEGIN INTEGER CARA;
0457 --                W:=NIL;
0458 --                UASTACK(1):=A; USTK_PTR:=1;
0460 --                WHILE USTK_PTR¬=0 DO
0460 4-                BEGIN CARA:=UASTACK(USTK_PTR);
0462 --                      IF ISCONS(CARA)
0462 5-                      THEN BEGIN IF USTK_PTR=1000 THEN
0463 6-                                 BEGIN WRITE("NOTEVARIABLE STACK OVERFLOW");
0465 -6                                    ASSERT FALSE;  END;
0467 --                                 UASTACK(USTK_PTR):=FORCE(CDR(CARA));
0468 --                                 UASTACK(USTK_PTR+1):=FORCE(CAR(CARA));
0469 --                                 USTK_PTR:=USTK_PTR+1;
0470 -5                           END
0470 --                      ELSE
0470 --                      IF ISSYME(CARA) OR ISNUMB(CARA)
0470 --                      THEN USTK_PTR:=USTK_PTR-1
0470 55                     ELSE BEGIN W:=CONS(CARA,W); USTK_PTR:=USTK_PTR-1; END;
0474 -4                END;
0475 -3          END NOTEVARIABLES;
0476 --
0476 --          LOGICAL PROCEDURE CHECKVARIABLES;
0477 3-          BEGIN
0478 --             WHILE ¬ISNIL(A) AND ISUNK(CAR(A)) DO A:=CDR(A);
0479 --             ISNIL(A)
0479 -3          END;
```

```
0480 --          INTEGER PROCEDURE PRIMITIVE(INTEGER VALUE RATOR);
0481 3-          BEGIN INTEGER OP1,OP2,OP3;  INTEGER C;
0484 --             CASE RATOR OF (              %NOTE: 1=SUCCESS 2=FAILURE 3=ERROR%
0484 4-             BEGIN ADD : OP1:=CAR(A); OP2:=CAR(CDR(A)); OP3:=CAR(CDR(CDR(A)));
0488 --                         C:=(FLAGS(OP1)-3)*4+(FLAGS(OP2)-3)*2+FLAGS(OP3)-2;
0489 --                         CASE C OF (
0489 --                         IF CAR(OP3)=CAR(OP1)+CAR(OP2) THEN 1 ELSE 2,
0489 55                         BEGIN W:=NUMB(CAR(OP1)+CAR(OP2)); CHANGETOINDIRECT(OP3,W); 1 END,
0492 55                         BEGIN W:=NUMB(CAR(OP3)-CAR(OP1)); CHANGETOINDIRECT(OP2,W); 1 END,
0495 --                         3, %OP2 AND OP3 UNKNOWN%
0495 55                         BEGIN W:=NUMB(CAR(OP3)-CAR(OP2)); CHANGETOINDIRECT(OP1,W); 1 END,
0498 -4                         3,3,3 %OP1/3, OP1/2, OP1/2/3 UNKNOWN% ) END,
0498 --
0498 4-             BEGIN SUB : OP1:=CAR(A); OP2:=CAR(CDR(A)); OP3:=CAR(CDR(CDR(A)));
0502 --                         C:=(FLAGS(OP1)-3)*4+(FLAGS(OP2)-3)*2+FLAGS(OP3)-2;
0503 --                         CASE C OF (
0503 --                         IF CAR(OP3)=CAR(OP1)-CAR(OP2) THEN 1 ELSE 2,
0503 55                         BEGIN W:=NUMB(CAR(OP1)-CAR(OP2)); CHANGETOINDIRECT(OP3,W); 1 END,
0506 55                         BEGIN W:=NUMB(CAR(OP1)-CAR(OP3)); CHANGETOINDIRECT(OP2,W); 1 END,
0509 --                         3, %OP2 AND OP3 UNKNOWN%
0509 55                         BEGIN W:=NUMB(CAR(OP3)+CAR(OP2)); CHANGETOINDIRECT(OP1,W); 1 END,
0512 -4                         3,3,3 %OP1/3, OP1/2, OP1/2/3 UNKNOWN% ) END,
0512 --
0512 4-             BEGIN MUL : OP1:=CAR(A); OP2:=CAR(CDR(A)); OP3:=CAR(CDR(CDR(A)));
0516 --                         C:=(FLAGS(OP1)-3)*4+(FLAGS(OP2)-3)*2+FLAGS(OP3)-2;
0517 --                         CASE C OF (
0517 --                         IF CAR(OP3)=CAR(OP1)*CAR(OP2) THEN 1 ELSE 2,
0517 55                         BEGIN W:=NUMB(CAR(OP1)*CAR(OP2)); CHANGETOINDIRECT(OP3,W); 1 END,
0520 --                         IF CAR(OP3) REM CAR(OP1)=0
0520 5-                            THEN BEGIN W:=NUMB(CAR(OP3) DIV CAR(OP1));
0522 -5                                       CHANGETOINDIRECT(OP2,W); 1 END
0523 --                            ELSE 2,
0523 --                         3, %OP2 AND OP3 UNKNOWN%
0523 --                         IF CAR(OP3) REM CAR(OP2)=0
0523 5-                            THEN BEGIN W:=NUMB(CAR(OP3) DIV CAR(OP2));
0525 -5                                       CHANGETOINDIRECT(OP1,W); 1 END
0526 --                            ELSE 2,
0526 -4                         3,3,3 %OP1/3, OP1/2, OP1/2/3 UNKNOWN% ) END,
0526 --
0526 4-             BEGIN DIV_: OP1:=CAR(A); OP2:=CAR(CDR(A)); OP3:=CAR(CDR(CDR(A)));
0530 --                         IF ISUNK(OP1) OR ISUNK(OP2) THEN 3 ELSE
0530 --                         IF ISUNK(OP3)
0530 5-                         THEN BEGIN W:=NUMB(CAR(OP1) DIV CAR(OP2));
0532 -5                                    CHANGETOINDIRECT(OP3,W); 1 END
0533 --                         ELSE
0533 --                         IF CAR(OP1) DIV CAR(OP2)=CAR(OP3) THEN 1
0533 -4                         ELSE 2  END,
0533 --
0533 4-             BEGIN REM_: OP1:=CAR(A); OP2:=CAR(CDR(A)); OP3:=CAR(CDR(CDR(A)));
0537 --                         IF ISUNK(OP1) OR ISUNK(OP2) THEN 3 ELSE
0537 --                         IF ISUNK(OP3)
0537 5-                         THEN BEGIN W:=NUMB(CAR(OP1) REM CAR(OP2));
0539 -5                                    CHANGETOINDIRECT(OP3,W); 1 END
0540 --                         ELSE
0540 --                         IF CAR(OP1) REM CAR(OP2)=CAR(OP3) THEN 1
0540 -4                         ELSE 2  END,

0540 4-             BEGIN ATOM: OP1:=CAR(A);
0542 --                         IF ISCONS(OP1) THEN 2 ELSE
0542 -4                         IF ISUNK(OP1) THEN 3 ELSE 1  END,
0542 --
0542 4-             BEGIN EQ  : OP1:=CAR(A); OP2:=CAR(CDR(A));
0545 --                         IF ISSYMB(OP1) AND ISSYMB(OP2) AND SVAL(OP1)=SVAL(OP2)
0545 --                         OR ISNUMB(OP1) AND ISNUMB(OP2) AND CAR(OP1)=CAR(OP2)
0545 --                         THEN 1 ELSE
0545 --                         IF ISUNK(OP1) OR ISUNK(OP2) THEN 3
0545 -4                         ELSE 2  END,
0545 --
0545 4-             BEGIN LEQ : OP1:=CAR(A); OP2:=CAR(CDR(A));
0548 --                         IF ISNUMB(OP1) AND ISNUMB(OP2) AND CAR(OP1)<=CAR(OP2)
0548 --                         THEN 1 ELSE
0548 --                         IF ISUNK(OP1) OR ISUNK(OP2) THEN 3
0548 -4                         ELSE 2  END
0548 --             ) %END MAIN CASE%
0548 -3          END PRIMITIVE;
```

```
0549 --         INTEGER DE,A,F,L,C,R,DU,B,N,NEGTRAP,W;
0550 --

0550 --         PROCEDURE APPLY(INTEGER VALUE SPEC,DEFS,QUERYARGS,QUERYVARS);
0551 3-         BEGIN INTEGER TEMP; LOGICAL HALTED,SOMESOLUTIONS;
0554 --            DE:=DEFS; A:=QUERYARGS; F:=NIL; L:=QUERYVARS; C:=SPEC;
0559 --            R:=NIL; DU:=NIL; B:=NIL; N:=NIL;
0563 --            NEGTRAP:=SIMPLECONS(NIL,SIMPLECONS(NIL, %CODE 17 = NEGSUCCEED%
0563 --                SIMPLECONS(SIMPLECONS(SIMPLENUMB(17),NIL),SIMPLECONS(NIL,NIL))));
0564 --            HALTED:=SOMESOLUTIONS:=FALSE;
0565 --
0565 --            WHILE ¬HALTED DO
0565 4-            BEGIN
0566 --               CASE CAR(CAR(C)) OF
0566 5-              BEGIN
0567 --
0567 6-                  BEGIN DCLLOCALS : L:=NIL;
0569 --                                    FOR I:=1 UNTIL CAR(CAR(CDR(C))) DO
0569 77                                          BEGIN W:=UNK; L:=CONS(W,L); END;
0573 -6                                   C:=CDR(CDR(C)); END;
0575 --
0575 6-                  BEGIN ENDCASE   : L:=CAR(DU); C:=CAR(CDR(DU));
0576 -6                                    DU:=CDR(CDR(DU)); END;
0580 --
0580 6-                  BEGIN INVOKE    : TEMP:=DE;
0582 --                                    FOR I:=1 UNTIL CAR(CAR(CDR(C))) DO TEMP:=CDR(TEMP);
0583 --                                    DU:=CONS(CDR(CDR(C)),DU); DU:=CONS(L,DU);
0585 -6                                   C:=CAR(TEMP); END;
0587 --
0587 66                  BEGIN ALDC      : A:=CONS(CAR(CDR(C)),A); C:=CDR(CDR(C)); END;
0591 --
0591 6-                  BEGIN ALDV      : TEMP:=L;
0593 --                                    FOR I:=1 UNTIL CAR(CAR(CDR(C))) DO TEMP:=CDR(TEMP);
0594 -6                                   A:=CONS(FORCE(CAR(TEMP)),A); C:=CDR(CDR(C)); END;
0597 --
0597 6-                  BEGIN ACONS     : W:=A;  A:=CDR(A);  CDR(W):=CAR(W);  CAR(W):=CAR(A);
0602 --                                    CAR(A):=W;
0603 -6                                   C:=CDR(C); END;
0605 --
0605 66                  BEGIN ATOP      : A:=CAR(A); C:=CDR(C); END;
0609 --
0609 66                  BEGIN FLDC      : F:=CONS(CAR(CDR(C)),F); C:=CDR(CDR(C)); END;
0613 --
0613 6-                  BEGIN FLDV      : TEMP:=L;
0615 --                                    FOR I:=1 UNTIL CAR(CAR(CDR(C))) DO TEMP:=CDR(TEMP);
0616 -6                                   F:=CONS(FORCE(CAR(TEMP)),F); C:=CDR(CDR(C)); END;
0619 --
0619 6-                  BEGIN FCONS     : W:=F;  F:=CDR(F);  CDR(W):=CAR(W);  CAR(W):=CAR(F);
0624 --                                    CAR(F):=W;
0625 -6                                   C:=CDR(C); END;
0627 --
0627 66                  BEGIN FTOP      : F:=CAR(F); C:=CDR(C); END;
0631 --
0631 6-                  BEGIN UNIF      : IF UNIFY THEN C:=CDR(C)
0632 -6                                   ELSE POPBACK;  END;

0634 6-                  BEGIN TRYCASE   : B:=CONS(DU,B); B:=CONS(CDR(CDR(C)),B);
0637 --                                    B:=CONS(A,B); B:=CONS(R,B);
0639 -6                                   R:=NIL; C:=CAR(CDR(C)); END;
0642 --
0642 6-                  BEGIN ENDPRED   : IF ISNIL(B)
0643 7-                                    THEN BEGIN IF ¬SOMESOLUTIONS THEN PUTTOKEN("FAILURE");
0645 -7                                               HALTED:=TRUE; END
0646 -6                                   ELSE POPBACK; END;
0648 --
0648 6-                  BEGIN HALT      : IF ISNIL(QUERYVARS)
0649 --                                        THEN PUTTOKEN("SATISFIED")
0649 7-                                        ELSE BEGIN TEMP:=QUERYVARS;
0651 8-                                                WHILE ¬ISNIL(TEMP) DO BEGIN PUTEXP(CAR(TEMP));
0653 -8                                                                            TEMP:=CDR(TEMP); END;
0655 -7                                             END;
0656 --                                    FORCELINEOUT;
0657 --                                    NO_OF_SOLUTIONS:=NO_OF_SOLUTIONS-1;
0658 --                                    SOMESOLUTIONS:=TRUE;
0659 --                                    IF NO_OF_SOLUTIONS=0 OR ISNIL(B) THEN HALTED:=TRUE
0659 -6                                   ELSE POPBACK; END;
0661 --
0661 6-                  BEGIN NEGSUCCTRAP: N:=CONS(B,N); N:=CONS(DU,N); N:=CONS(R,N);
0665 --                                    N:=CONS(CDR(CDR(CDR(CDR(C)))),N);
0666 --                                    N:=CONS(L,N);
0667 --                                    NOTEVARIABLES; %LIST APPEARS IN W%
0668 --                                    N:=CONS(W,N);
0669 -6                                   R:=NIL; B:=NEGTRAP; C:=CDR(C);  END;
0673 --
0673 6-                  BEGIN NEGSUCCEED: N:=CDR(N); L:=CAR(N); N:=CDR(N); C:=CAR(N);
0678 --                                    N:=CDR(N); R:=CAR(N); N:=CDR(N); DU:=CAR(N);
0682 -6                                   N:=CDR(N); B:=CAR(N); N:=CDR(N); END;
0686 --
0686 6-                  BEGIN NEGFAILORERR: A:=CAR(N);
0688 --                                    IF CHECKVARIABLES %TRUE IF NONE CHANGED%
0688 7-                                    THEN BEGIN N:=CDR(CDR(CDR(N))); R:=CAR(N);
0691 --                                               N:=CDR(CDR(N)); B:=CAR(N);
0693 --                                               N:=CDR(N); POPBACK;
0695 -7                                         END
0695 7-                                    ELSE BEGIN PUTTOKEN("ERROR");
0697 --                                              WRITE("ERROR: NEGATIVE LITERAL BOUND A VARIABLE");
0698 --                                              HALTED:=TRUE;
0699 -6                                         END;  END;
0701 --
0701 6-                  BEGIN ADDSUBETC: CASE PRIMITIVE(CAR(CAR(CDR(C)))) OF
0702 7-                                       BEGIN C:=CDR(CDR(C)); %SUCCESS%
0704 --                                             POPBACK;          %FAILURE%
0705 8-                                             BEGIN PUTTOKEN("ERROR");
0707 --                                                   WRITE("ERROR: BAD ARGUMENTS TO PRIMITIVE");
0708 --                                                   HALTED:=TRUE;
0709 -8                                             END
0709 -6                                       END;  END;
0711 --
0711 -5              END;
0712 -4            END EXEC_CYCLE;
0713 --
0713 -3         END;
```

```
ALGOL W : PROLOG.MACHINE           CONTROL OPERATION                                  1981 JUNE

0714 --          INTEGER PROGRAM,SPEC,DEFS,QUERYARGS,QUERYVARIABLES;
0715 --          INTEGER NO_OF_SOLUTIONS;
0716 --
0716 --          WRITE("LOOK FOR HOW MANY SOLUTIONS ?"); IOCONTROL(2);
0718 --          READON(NO_OF_SOLUTIONS);
0719 --          WRITE("NO.OF SOLUTIONS SOUGHT :",NO_OF_SOLUTIONS); IOCONTROL(2);
0721 --
0721 --          INIT_LEXICAL;INIT_SYNTAX;INIT_LIST_STORAGE;
0724 --          GETEXP(PROGRAM);  QUERYVARIABLES:=NIL;  GETEXPLIST(QUERYARGS);
0727 --          SPEC:=CAR(PROGRAM); DEFS:=CDR(PROGRAM);
0729 --          APPLY(SPEC,DEFS,QUERYARGS,QUERYVARIABLES);
0730 --          FORCELINECUT;
0731 --
0731 -2      END MAIN_BLOCK;
0732 -1      END.
```

Appendix E.

Appendix E.

Tabulated results from test executions on the Lispkit Machine.

E.1 Assessing the Lispkit Machine (Chapter 6).

Tables 1-6 cover the statistics LS1-5 obtained from executing the naive reverse, reverse with accumulating parameter, quicksort, iterative summing, higher order summing and powering programs on the LM. The s-expression storage management component was set at 50000 heap cells, with the exception of an extra series of experiments (Table 1, rightmost column) for which the allocation was set at 4500 cells.

Here is a recap of the behavioural measures recorded by the statistics:

1. LS1 is the total number of function applications executed. This is the number of AP and RAP instructions executed by the LM, and thus includes where and whererec expressions.
2. LS2 is the total number of LM instructions executed.
3. LS3 counts the total number of Apply loop steps (= LS2) and environment lookup steps. LS3 is proportional to the number of IBM 370 instructions and Storage operations executed by Apply.
4. LS4 is the total number of s-expression storage allocation and access requests executed by Apply.
5. LS5 counts the number of steps executed by the Storage management component. It counts the allocation and access operations (= LS4), and the number of garbage collector marking and scanning steps, and is proportional to the number of IBM 370 instructions executed by the Storage management.

E.2 Assessing LISPINT1 and LISPINT2 (Chapter 7).

Tables 7 and 8 show the statistics LIS1 and LIS2 for the execution of naive reverse and reverse with accumulating parameter on LISPINT1 on the LM. The s-expression storage was set at 50000 heap cells.

Both LIS1 and LIS2 are counts of numbers of LM instructions executed, with the following significance:

1. LIS1 is the number of LM instructions executed by the test program running directly on the LM. It is proportional to the work performed by the program as predicted by a model interpretation.
2. LIS2 is the number of LM instructions executed by LISPINT1 during the interpretation. It is proportional to the amount of work performed by LISPINT1 as predicted by a model interpretation.

A rough measure of the total "problem state" CPU time in seconds is also included for comparision; this includes the time for operations such as heap initialisation, s-expression input and output, collection and output of statistics.

Tables 9 and 10 show the statistics LIS3 and LIS4 for the execution of naive reverse and reverse with accumulating parameter on LISPINT2 on the LM. The s-expression storage was set at 50000 and 5400 heap cells respectively. LIS3 and LIS4 correspond to LIS1 and LIS2 respectively. Again a rough measure of CPU time is included in the tables.

Appendix E. Table 1. Assessing the LM. Naive reverse. 50000 Heap cells.

List length	LS1	LS2	LS3	LS4	LS5	LS5 (4500 cells)
1	4	47	69	921	921	921
10	67	1064	1833	21342	21342	21342
20	232	3809	6643	76522	76522	81469
40	862	14399	25263	289482	289482	310376
50	1327	22244	39073	447262	447262	484583
100	5152	86969	153123	1749162	1852358	1936689
150	11477	194194	342173	3906062	4168875	4480692
200	20302	343919	606223	6917962	7397795	8097341
250	31627	536144	945273	10784862	11538758	13241032
300	45452	770869	1359323	15506762	16648259	20349385

Appendix E. Table 2. Assessing the LM.

Reverse with accumulating parameter. 50000 Heap cells.

List length	LS1	LS2	LS3	LS4	LS5
1	5	49	73	980	980
20	24	372	624	7459	7459
50	54	882	1494	17689	17689
100	104	1732	2944	34739	34739
150	154	2582	4394	51789	51789
200	204	3432	5844	68839	68839
250	254	4282	7294	85889	85889
300	304	5132	8744	102939	102939
500	504	8532	14544	171139	171139
1000	1004	17032	29044	341639	341639

Appendix E. Table 3. Assessing the LM.

Quicksort. 50000 Heap cells.

List length	LS1	LS2	LS3	LS4	LS5
1	3	38	52	731	731
3	11	198	335	3881	3881
7	35	690	1225	13597	13597
15	99	2018	3653	39861	39861
31	259	5362	9805	106053	106053
63	643	13426	24701	265765	265765
127	1539	32306	59677	639845	639845
255	3587	75570	139997	1497317	1552062
511	8195	173106	321373	3430885	3643895
1023	18435	390194	725597	7735269	8240558

Appendix E. Table 4. Assessing the LM.

Iterative summing. 50000 Heap cells.

m = 1 n	LS1	LS2	LS3	LS4	LS5
1	2	17	26	354	354
20	21	340	634	7327	7327
50	51	850	1594	18337	18337
100	101	1700	3194	36687	36687
150	151	2550	4794	55037	55037
200	201	3400	6394	73387	73387
250	251	4250	7994	91737	91737
300	301	5100	9594	110087	110087
500	501	8500	15994	183487	183487
1000	1001	17000	31994	366987	366987

Appendix E. Table 5. Assessing the LM.

Higher order summing. 50000 Heap cells.

m = 1 n	LS1	LS2	LS3	LS4	LS5
1	5	49	89	1045	1045
20	100	980	2008	21831	21831
50	250	2450	5038	54651	54651
100	500	4900	10088	109351	109351
150	750	7350	15138	164051	164051
200	1000	9800	20188	218751	218751
250	1250	12250	25238	273451	273451
300	1500	14700	30288	328151	328151
500	2500	24500	50488	546951	546951
1000	5000	49000	100988	1093951	1614072

Appendix F. Table 6. Assessing the LM.

Powering. 50000 Heap cells.

n = 2 k	LS1	LS2	LS3	LS4	LS5
1	4	47	74	959	959
2	6	82	130	1663	1663
3	10	162	251	3230	3230
5	34	682	1013	13268	13268
7	130	2822	4115	54374	54374
10	1026	22952	33208	440521	440521
12	4098	92052	133030	1765771	1866675
14	16386	368512	532372	7067725	7521906
15	32770	737142	1064843	14137262	15096403

Appendix E. Table 7. Assessing LISPINT1.

Naive reverse. 50000 Heap cells.

List length	LIS1	LIS2	CPU seconds
0	19	779	1.19
1	47	3843	1.40
2	92	31487	3.25
3	154	521619	40.07
4	Not completed. More than 200 CPU seconds required.		

Appendix E. Table 8. Assessing LISPINT1.

Reverse with accumulating parameter. 50000 Heap cells

List length	LIS1	LIS2	CPU seconds
0	32	1635	1.25
1	49	4078	1.43
2	66	10556	1.90
3	83	29139	3.24
4	100	84037	7.83
5	117	247880	21.10
6	134	738558	60.34

Appendix E. Table 9. Assessing LISPINT2.

Naive reverse. 50000 Heap cells.

List length	LIS3	LIS4	CPU seconds
0	19	783	1.19
1	47	2246	1.30
2	92	4660	1.46
3	154	8025	1.68
4	233	12341	1.97
6	442	23826	2.74
8	719	39115	3.78
10	1064	58208	5.36
20	3809	210733	16.90
50	22244	1238908	97.10

Appendix E. Table 10. Assessing LISPINT2.

Reverse with accumulating parameter. 5400 Heap cells

List length	LIS3	LIS4	CPU seconds
0	32	1448	0.65
1	49	2361	0.72
2	66	3274	0.87
3	83	4137	0.93
4	100	5100	1.08
6	134	6926	1.21
8	168	8752	1.43
10	202	10578	1.65
20	372	19708	2.85
50	892	47098	10.37

Appendix F.

Appendix F.

Tabulated results from test executions on the Prolog Machine.

F.1 Assessing the Prolog Machine (Chapter 6)

Tables 1-5 cover the statistics PS1-7 obtained from executing the naive reverse, reverse with accumulating parameter, quicksort, iterative summing and powering programs on the PM. The su-expression storage management component was set at 50000 heap cells for cons, number and unknown allocations. Statistics were collected only until the production of the first result.

Here is a recap of the behavioural measures recorded by the statistics:

1. PS1 counts the total number of predicate cases executed during the computation. It is found by counting the number of UNIFY instructions which the PM executes.
2. PS2 is simply the total number of PM instructions executed.
3. PS3 counts the total number of Apply loop steps (= PS2), the steps in local environment building and lookup, and the number of predicate definition lookup steps. PS3 is proportional to the number of IBM 370 instructions, Support operations, and Storage operations executed by Apply.
4. PS4 is the total number of Support operations requested by Apply. These are the unification, backtracking, and data structure checking routines.
5. PS5 is proportional to the number of IBM 370 instructions and Storage operations executed by the Support component. It is found by counting the internal looping steps of the Support routines and adding the number of support operations requested (=PS4).

6. PS6 counts the number of Storage operations executed by the Support and Apply components.
7. PS7 is proportional to the number of IBM 370 instructions executed by the Storage component. It is found by counting the internal looping steps of the garbage collector and forcing routines, and adding the number of Storage operations requested (= PS6).

F.2 Assessing PROLOGINT (Chapter 7).

Tables 6 and 7 show the statistics PIS1 and PIS2 for the execution of naive reverse and reverse with accumulating parameter on PROLOGINT on the PM. The su-expression storage was set at 70000 and 23000 heap cells respectively. Statistics were collected only until production of the first result.

Both PIS1 and PIS2 are counts of numbers of PM instructions executed, with the following significance:
1. PIS1 is the number of PM instructions executed by the test program running directly on the PM. It is a representation of the work performed by the program as predicted by a model interpretation.
2. PIS2 is the number of PM instructions executed by PROLOGINT during interpretation of the test program. It is a representation of the work done by PROLOGINT as predicted by a model interpretation.

A rough measure of the total "problem state" CPU time in seconds is also included for comparison; this includes the time for heap initialisation, su-expression input and output, collection and output of statistics.

Appendix F. Table 1. Assessing the PM. Naive reverse. 50000 Heap cells.

List length	PS1	PS2	PS3	PS4	PS5	PS6	PS7
1	5	80	123	6	32	2433	2440
5	37	644	1115	52	252	24897	24956
10	122	2159	3840	177	842	88077	88291
20	442	7889	14240	652	3072	331437	332261
30	962	17219	31240	1427	6702	730797	732631
40	1682	30149	54840	2502	11732	1286157	1289401
50	2602	46679	85040	3877	18162	1997517	2002571
60	3722	66809	121840	5552	25992	2864877	2975131
65	4357	78224	142715	6502	30432	3357057	3727401
66	4490	80615	147088	6701	31362	3460173	4248289
67	Heap overflow						

Appendix F. **Table 2.** **Assessing the PM.**

Reverse with accumulating parameter. **50000 Heap cells.**

List length	PS1	PS2	PS3	PS4	PS5	PS6	PS7
1	4	69	112	5	28	1940	1947
20	42	753	1347	62	294	23125	23208
50	102	1833	3297	152	714	56575	56778
100	202	3633	6547	302	1414	112325	112728
150	302	5433	9797	452	2114	168075	168678
200	402	7233	13047	602	2814	223825	224628
250	502	9033	16297	752	3514	279575	280578
300	602	10833	19547	902	4214	335325	336528
500	1002	18033	32547	1502	7014	558325	560328
1000	2002	36033	65047	3002	14014	1115825	1119828

Appendix F. Table 3. Assessing the PM. Quicksort. 50000 Heap cells.

List length	PS1	PS2	PS3	PS4	PS5	PS6	PS7
1	6	115	208	7	46	3521	3532
3	19	403	765	27	173	14539	14582
7	55	1227	2349	87	553	47827	47970
15	147	3371	6457	247	1565	136907	137322
31	371	8651	16553	647	4093	360075	361178
63	899	21195	40505	1607	10157	896427	899194
127	2115	50251	95929	3847	24301	2149163	2246430

Appendix F. Table 4. Assessing the PM. Iterative summing. 50000 Heap cells.

m = 1 n	PS1	PS2	PS3	PS4	PS5	PS6	PS7
1	2	35	57	2	16	910	915
20	40	1156	2033	97	434	32545	32721
50	100	2926	5153	247	1094	82495	82941
100	200	5876	10353	497	2194	165745	166641
150	300	8826	15553	747	3294	248995	250341
200	400	11776	20753	997	4394	332245	334041
250	500	14726	25953	1247	5494	415495	417741
300	600	17676	31153	1497	6594	498745	501441
500	1000	29476	51953	2497	10994	831745	836241
1000	2000	58976	103953	4997	21994	1664245	1761089

Appendix F. Table 5. Assessing the PM. Powering. 50000 Heap cells.

n = 2 k	PS1	PS2	PS3	PS4	PS5	PS6	PS7
1	5	107	193	8	41	4241	4254
2	9	197	339	16	73	9314	9336
3	17	394	710	32	132	20640	20683
5	65	1644	3252	128	466	93316	93497
7	257	6746	13894	512	1772	391100	391851
8	513	13577	28215	1024	3505	790112	791628
9	1025	27256	56936	2048	6966	1589316	1592365
10	2049	54631	114457	4096	13883	3183904	3282384
11	Heap overflow						

Appendix F. Table 6. Assessing PROLOGINT.

Naive reverse. 70000 Heap cells.

List length	PIS1	PIS2	CPU seconds
0	29	2470	1.49
1	80	8698	1.92
2	167	20467	2.73
3	290	37777	3.93
4	449	60628	5.53
6	875	122953	14.63
7	1142	162427	25.05
8	1445	207442	47.17

Appendix F. Table 7. Assessing PROLOGINT.

Reverse with accumulating parameter. 23000 Heap cells

List length	PIS1	PIS2	CPU seconds
0	33	3135	0.93
1	69	8702	1.31
2	105	14269	1.69
4	177	25403	3.06
6	249	36537	4.71
8	321	47671	6.58
10	393	58805	10.02

APPENDIX G. BIBLIOGRAPHY.

Appendix G. Bibliography.

Anderson, T., Lee, P.A., and Shrivastava, S.K. (1978) :
A Model of Recoverability in Multilevel Systems.
IEEE Transactions on Software Engineering,
Vol. SE-4, No.6, 486-494.

Boyer, R.S., and Moore, J.S. (1972) :
The Sharing of Structure in Theorem Proving Programs.
Machine Intelligence 7 (Eds. Meltzer and Michie), 101-116,
Edinburgh University Press.

Boyer, R.S., and Moore, J.S. (1975) :
Proving Theorems about Lisp Functions.
JACM, Vol. 22, No. 1, 129-144.

Burstall, R.M. (1977) :
Design Considerations for a Functional
Programming Language.
Infotech State of the Art Conference, 45-57, Copenhagen.

Burstall, R.M., and Darlington, J. (1973) :
A System which Automatically Improves Programs.
Proceedings 3rd International Joint Conference on
Artificial Intelligence, 479-485, Stanford, California.

Burstall, R.M., and Darlington, J. (1977) :
A Transformation System for Developing Recursive Programs.
JACM, Vol. 24, No. 1, 44-67.

Church, A. (1941) :
The Calculi of Lambda-Conversion.
Annals of Mathematical Studies, No. 6,
Princeton University Press.

Church, A. (1952) :
Introduction to Mathematical Logic.
Princeton University Press.

Clark, K.L. (1978) :
Negation as Failure.
Logic and Databases (Eds. Gallaire and Minker), 293-322,
Plenum Press, New York.

Clark, K.L. (1979) :
Predicate Logic as a Computational Formalism.
Research Monograph, Imperial College, London.

Clark, K.L., and Darlington, J. (1980) :
Algorithm Classification through Synthesis.
Computer Journal, Vol. 23, No. 1, 61-65.

Clark, K.L., and McCabe, F. (1979) :
Programmer's Guide to IC-Prolog.
CCD Report 79/7, Imperial College, London.

Clark, K.L., and McCabe, F. (1979) :
The Control Facilities of IC-Prolog.
Expert Systems in the Microelectronic Age (Ed. Michie),
122-149, AISB Summer School, Edinburgh University Press.

Clark, K.L., and Tarnlund, S.-A. (1977) :
A First Order Theory of Data and Programs.
Proceedings IFIP 77, 939-944, North-Holland.

Davis, R.E. (1980) :
Runnable Specifications as a Design Tool.
Proceedings of Logic Programming Workshop. 106-117,
Debrecen, Hungary.

Deliyanni, A., and Kowalski, R. (1979) :
Logic and Semantic Networks.
Comm. ACM, Vol. 22, No. 3, 184-192.

Hansson, A., and Tarnlund, S.-A. (1980) :
Program Transformation by a Function that maps
Simple Lists onto D-Lists.
Proceedings of Logic Programming Workshop, 225-229,
Debrecen, Hungary.

Henderson, P. (1978) :
Lispkit System - A Software Kit.
Computing Laboratory Technical Report No. 129,
University of Newcastle upon Tyne.

Henderson, P. (1980) :
Functional Programming: Application and Implementation.
Prentice-Hall International, Series in Computer Science,
London.

Henderson, P., and Morris, J.H. (1976) :
A Lazy Evaluator.
Proceedings 3rd Symposium on the Principles of
Programming Languages, 95-103, Atlanta.

Jones, S.B. (1979) :
Predicate Logic Programming Style for Database
Manipulation and General Computation.
Computing Laboratory Memo MRM/151,
University of Newcastle upon Tyne.

Jones, S.B. (1980) :
Structured Programming Techniques in Prolog.
Proceedings of Logic Programming Workshop, 322-333,
Debrecen, Hungary.

Kowalski, R. (1974) :
Predicate Logic as a Programming Language.
Proceedings IFIP 74, 569-574, North-Holland, Amsterdam.

Kowalski, R. (1978) :
Logic for Data Abstraction.
Logic and Database (Eds. Gallaire and Minker), 77-102,
Plenum Press, New York.

Kowalski, R. (1979) :
Logic for Problem Solving.
Elsevier North-Holland, Artificial Intelligence Series,
New York.

Kowalski, R. (1979) :
Algorithm = Logic + Control.
Comm. ACM, Vol. 22, No. 7, 424-436.

Landin, P.J. (1964) :
The Mechanical Evaluation of Expressions.
Computing Journal, Vol. 6, No. 4, 308-320.

Landin, P.J. (1965) :
A Correspondence between Algol 60 and Church's Lambda
Notation: Parts I and II.
Comm. ACM, Vol. 8, Nos. 2 and 3, 89-101 and 158-165.

Landin, P.J. (1966) :
The Next 700 Programming Languages.
Comm. ACM, Vol. 9, No. 3, 157-166.

Manna, Z., and Vuillemin, J. (1972) :
Fixpoint Approach to the Theory of Computation.
Comm. ACM, Vol. 15, No. 7, 528-536.

McCarthy, J. (1960) :
Recursive Functions of Symbolic Expressions and their
Computation by Machine.
Comm. ACM, Vol. 3, No. 4, 185-195.

McCarthy, J., et al. (1962) :
The Lisp 1.5 Programmer's Manual.
MIT Press, Cambridge, Mass.

McCarthy, J., (1963) :
A Basis for a Mathematical Theory of Computation.
Computer Programming and Formal Systems (Eds. Braffort and Hirschberg), 33-70, North-Holland, Amsterdam.

Mellish, C.S. (1980) :
An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter.
Proceedings of Logic Programming Workshop, 21-32, Debrecen, Hungary.

Moss, C. (1980) :
The Comparison of several Prolog systems.
Proceedings of Logic Programming Workshop, 198-200, Debrecen, Hungary.

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

Paterson, M.S. (1976) :
Linear Unification.
Proceedings 8th Annual ACM Symposium on the Theory of Computing, 181-186, Pennsylvania.

Reynolds, J.C. (1972) :
Definitional Interpreters for Higher Order Programming Languages.
Proceedings ACM Annual Conference, 717-740, Boston.

Robinson, J.A. (1963) :
Theorem Proving on the Computer.
JACM, Vol. 10, April, 163-174.

Robinson, J.A. (1965) :
A Machine Oriented Logic Based on the Resolution Principle.
JACM, Vol. 12, No. 1, January, 23-41.

Robinson, J.A. (1967) :
A Review of Automatic Theorem Proving.
Annual Symposia in Applied Mathematics, Vol. 19, 1-18, American Mathematical Society.

Robinson, J.A. (1971) :
Computational Logic: The Unification Computation.
Machine Intelligence 6 (Eds. Meltzer and Michie), 63-72, Edinburgh University Press.

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

Roussel, P. (1975) :
Prolog: Manuel de Reference et d'Utilisation.
Groupe d'Intelligence Artificielle,
Universite d'Aix-Marseille, Luminy.

Sickel, S., and McKeeman, W. (1980) :
Hoare's Program FIND Revisited (Abstract).
Proceedings of Logic Programming Workshop, 224,
Debrecen, Hungary.

Steele, G.L., and Sussman, G.J. (1980) :
Design of a Lisp Based Microprocessor.
Comm. ACM, Vol. 23, No. 11, 628-645.

Sussman, G.J. (1980) :
Lisp Chip - Scheme 79.
Proceedings of Joint SRC/University of Newcastle upon Tyne
Workshop on VLSI: Machine Architecture and
High Level Languages (Ed. Treleaven), 31-36,
Computing Laboratory Technical Report No. 156,
University of Newcastle upon Tyne.

Turner, D.A. (1979) :
A New Implementation Technique for Applicative Languages.
Software Practice and Experience, Vol. 9, 31-49.

Warren, D.H.D. (1977) :
Implementing Prolog.
Research Reports 39 and 40,
Dept. of Artificial Intelligence, Edinburgh University.

Warren, D.H.D. (1977) :
Logic Programming and Compiler Writing.
Research Report 44,
Dept. of Artificial Intelligence, Edinburgh University.

Warren, D.H.D., Pereira, L.M., and Pereira, F. (1977) :
Prolog - The Language and its Implementation
compared with Lisp.
SIGPLAN Notices, Vol. 12, No. 8, 109-115.

Additional reference:

McCabe, F. (1980) :
MicroProlog Reference Manual. Imperial College, London.