SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 10, 97–125 (1980)

Logic Programming and Compiler Writing

DAVID H. D. WARREN

Department of Artificial Intelligence, University of Edinburgh, Hope Park Square, Edinburgh EH8 9NW, Scotland

SUMMARY

The concept of 'logic programming', and its practical application in the programming language Prolog, are explained from first principles. The ideas are illustrated by describing in detail one sizable Prolog program which implements a simple compiler. The advantages and practicability of using Prolog for 'real' compiler implementation are discussed.

KEY WORDS Logic Programming Prolog Compiler Compiler specification Compiler implementation

INTRODUCTION

This paper aims to provide an introduction to the concept of 'logic programming'[1, 2, 3] for people with experience in other programming languages. The emphasis is on those aspects which have been put to practical use in the programming language Prolog[4, 5] developed at the University of Marseille. The ideas are illustrated by discussing at length one main example, consisting of a very simple compiler written in Prolog. Although this 'toy' compiler has been made as simple as possible for didactic purposes, the techniques employed are taken from a 'real' implementation in Prolog of a compiler in practical use. This example has been chosen with the additional purpose of demonstrating the particular advantages of Prolog for compiler writing. The reader is expected to be broadly familiar with various conventional programming languages, but no knowledge of symbolic logic is assumed. Some acquaintance with the issues involved in writing a compiler would be an advantage.

LOGIC PROGRAMMING

The principal idea[6] behind logic programming is that an algorithm can be usefully analysed into a logical component and a control component:

algorithm = logic + control

Roughly speaking, the logical component defines *what* the algorithm does, and the control component prescribes *how* it is done efficiently.

The logical component can be expressed as statements of symbolic logic. For this purpose, one normally only needs to consider a restricted part of logic reduced to a standard form known as 'Horn clauses'. The language of this subset will now be described from a conventional programming standpoint. The notation and terminology will be that used in Prolog.

0038–0644/80/0210–0097$01.00 *Received 18 January 1979*

Syntax

The data objects of the language are called *terms*. A term is a *constant*, or a *variable* or a *compound term*.

The constants include *integers* such as:

0 1 999

and *atoms* such as:

a nil = := '*Algol*–68'

The symbol for an atom can be any sequence of characters, which in general must be written in quotes unless there is no possibility of confusion with other symbols (such as variables, integers). As in conventional programming languages, constants are definite elementary objects and correspond to proper nouns in natural language.

Variables will be distinguished by an initial capital letter, e.g.

*X Value A A*1

If a variable is only referred to once, it does not need to be named and may be written as an 'anonymous' variable indicated by a single underline character.

A variable should be thought of as standing for some definite but unspecified object. This is analogous to the use of a pronoun in natural language. Note that a variable is not simply a writeable storage location as in most programming languages. Compare instead the variable of pure Lisp, which is likewise a 'stand-in' for a data object rather than a location to be assigned to.

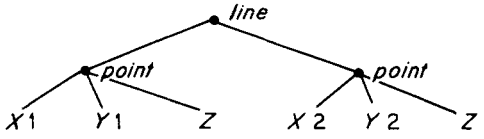
The structured data objects of the language are the compound terms. A compound term comprises a *functor* (called the *principal* functor of the term) and a sequence of one or more terms called *arguments*. A functor is characterized by its *name*, which is an atom, and its *arity* or number of arguments. For example, the compound term whose functor is named '*point*' of arity 3, with arguments *X*, *Y*, and *Z*, is written:

point(*X*, *Y*, *Z*)

Functors are generally analogous to common nouns in natural language. One may think of a functor as a record type and the arguments of a term as fields of a record. Compound terms are usefully pictured as trees. For example, the term:

line(*point*(*X*1, *Y*1, *Z*), *point*(*X*2, *Y*2, *Z*))

would be pictured as the structure:



Sometimes it is convenient to write a compound term using an optional infix notation, e.g.

X + *Y* (*P*; *Q*)

instead of:

+(*X*, *Y*) ;(*P*, *Q*)

Finally, note that an atom is treated as a functor of arity 0.

Suppose we wish to give a formal definition of a data type called a 'dictionary'. A dictionary will be either the atom '*void*' or a compound term of the form:

dic(⟨*term* 1⟩, ⟨*term* 2⟩, ⟨*term* 3⟩, ⟨*term* 4⟩)

where the arguments ⟨*term* 3⟩ and ⟨*term* 4⟩ are also dictionaries whilst ⟨*term* 1⟩ and ⟨*term* 2⟩ are of unrestricted type. (Here, and throughout this paper, names in angular brackets are used as 'meta-variables' to symbolize constructs of the 'object language' being discussed, cf. the non-terminal symbols of a Backus–Naur form (BNF) grammar.) The required definition of the data type 'dictionary' is expressed in logic by the following two statements:

dictionary(*void*).
dictionary(*dic*(*X*, *Y*, *D*1, *D*2)) :– *dictionary*(*D*1), *dictionary*(*D*2).

Here '*dictionary*(_)' is a special kind of functor called a *predicate*, analogous to a verb in natural language. (Predicates are distinguished from other functors only by the contexts in which they occur.) A term with a predicate as principal functor is called a *boolean* term, and is analogous to a simple statement in natural language.

In general, statements of logic can be considered to be a shorthand for descriptive statements of natural language. A statement of the form:

⟨*P*⟩ :– ⟨*Q*⟩, ⟨*R*⟩, ⟨...⟩.

should be read as:

⟨*P*⟩ if ⟨*Q*⟩ and ⟨*R*⟩ and ⟨...⟩.

Thus the two statements above might be read as:

'*void*' is a dictionary.
'*dic*(*X*, *Y*, *D*1, *D*2)' is a dictionary if *D*1 is a dictionary and *D*2 is a dictionary.

Any variables in a statement are interpreted as standing for arbitrary objects, so a more precise reading of the second statement would be:

> For any *X*, *Y*, *D*1 and *D*2, '*dic*(*X*, *Y*, *D*1, *D*2)' is a dictionary if *D*1 is a dictionary and *D*2 is a dictionary.

Note that the variables in different statements are completely independent even if they have the same name, i.e. the 'lexical scope' of a variable is restricted to a single statement.

The kind of logic statements we are considering are called *clauses*. For our purposes, a clause comprises a *head* and a *body*. The head is a boolean term and the body is a sequence of zero or more boolean terms called *goals*. In general a clause is written:

⟨*head*⟩ :– ⟨*goal* 1⟩, ⟨*goal* 2⟩, ⟨...⟩.

If the number of goals is zero, we speak of a *unit clause*, and this is written:

⟨*head*⟩.

Semantics

The semantics of the language we have described should be clear from its informal interpretation. However, it is useful to have a precise definition. The semantics will tell

us which boolean terms can be considered true according to some given clauses. Thus in the case of our clauses for '*dictionary*(_)', we shall know that a term ⟨*term*⟩ is a dictionary if the boolean term:

dictionary(⟨*term*⟩)

is true.

Here then is a recursive definition of what will be called the *declarative semantics* of clauses.

> A term is *true* if it is the head of some clause instance and each of the goals (if any) of that clause instance is true, where an *instance* of a clause (or term) is obtained by substituting, for each of zero or more of its variables, a new term for all occurrences of the variable.

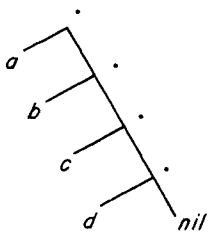
The unary predicate '*dictionary*(_)' specified a data type. More generally, predicates are used to express relationships between objects. For example, we might use '*concatenated*(⟨1⟩, ⟨2⟩, ⟨3⟩)' to mean that list ⟨3⟩ consists of the elements of list ⟨1⟩ followed by the elements of list ⟨2⟩. Thus

concatenated((*a.b.c.d.nil*),
(1.2.3.*nil*),
(*a.b.c.d*.1.2.3.*nil*))

is true, where a list is either the atom '*nil*' or a term formed from the binary functor '.' whose second argument is a list, i.e.

list(*nil*).
list(.(*X*, *L*)) :– list(*L*).

In general, as above, we write the functor '.(_,_)' as a right-associative infix operator so that, for example, the first list mentioned is equivalent to the standard form '.(*a*,.(*b*,.(*c*,.(*d*,*nil*))))' and should be pictured as:



The following clauses define the predicate '*concatenated*(_,_,_)':

concatenated(*nil*, *L*, *L*).
concatenated((*X*.*L*1), *L*2, (*X*.*L*3)) :– *concatenated*(*L*1, *L*2, *L*3).

The clauses may be read as:

> The empty list concatenated with any list L is simply L. A non-empty list consisting of X followed by remaining elements $L1$ concatenated with list $L2$ is the list consisting of X followed by remaining elements $L3$ where $L1$ concatenated with $L2$ is $L3$.

So far we have looked on clauses as a means of *specifying* relationships between objects. This is the traditional view of the purpose of logic.

Now consider what has to be done for relationships expressed in logic to be computed efficiently. For example, given terms ⟨*term* 1⟩ and ⟨*term* 2⟩, how can one find a term ⟨*term* 3⟩ such that:

concatenated(⟨*term* 1⟩, ⟨*term* 2⟩, ⟨*term* 3⟩)

is true? The major discovery of 'logic programming' is that the clauses themselves can often provide the basis of the procedures required. In such cases, it is only necessary to supply suitable control information to specify how the clauses are to be used effectively. In brief, logic has a 'procedural interpretation'.

The procedural interpretation treats a predicate as a *procedure name*, the head of a clause as a *procedure entry point* and a goal as a *procedure call*. A *procedure* is a set of clauses with the same head predicate. For example, the clauses for '*concatenated*(_,_,_)' can be considered to be a procedure for concatenating the elements of two given lists (amongst other uses). The procedure has two entry points corresponding to whether or not the first of the two input lists is empty. One of the clauses makes a recursive call to the same procedure.

Before we go on to consider the kind of control provided in Prolog, we should observe that not all sets of clauses make equally effective procedures. Some clauses would require unrealistically sophisticated control information to be of practical use. Much of the art of logic programming is to formulate the problem in such a way that it can be solved efficiently using the control mechanisms available. This soon comes quite naturally to someone with programming experience, as really it is just what one does in any other programming language; the ingenuity required is no greater, and usually less. Indeed, as we shall see, one of the main attractions of logic programming is that often a natural specification of an algorithm and a good implementation are one and the same.

THE PROGRAMMING LANGUAGE PROLOG

Introduction

A remarkably simple form of control suffices for many practical applications of logic programming. This point was first realized at Marseille and is the basis of the programming language Prolog developed there. From now on we shall restrict our attention to Prolog.

If we think back to the declarative semantics of clauses, it is clear that the order of the goals in a clause and the order of the clauses themselves, are both irrelevant to the declarative interpretation. However, these orderings are generally significant in Prolog as they constitute the main control information. In other respects a Prolog program is just a set of clauses.

When the Prolog system is executing a procedure call, the clause ordering determines the order in which the different entry points of the procedure are tried. The goal ordering fixes the order in which the procedure calls in a clause are executed. The 'productive' effect of a Prolog computation arises from the process of 'matching' a procedure call against a procedure entry point.

Really there are two different ways of looking at the meaning of a Prolog program. We have already discussed the declarative interpretation which Prolog inherits from logic. The alternative way is to consider, as for a conventional programming language, the

sequence of steps which take place when the program is executed. This is defined by the *procedural semantics* of Prolog. This semantics will tell us what happens when a goal (procedure call) is executed. The result of the execution will be to produce true instances of the goal (if there are any). Thus the procedural semantics is governed by the declarative. Here then is an exact description of the procedural semantics.

> To *execute* a goal, the system searches for the first clause whose head *matches* or *unifies* with the goal. The *unification* process[7] finds the most general common instance of the two terms, which is unique if it exists. If a match is found, the matching clause instance is then *activated* by executing in turn, from left to right, each of the goals of its body (if any). If at any time the system fails to find a match for a goal, it *backtracks*, i.e. it rejects the most recently activated clause, undoing any substitutions made by the match with the head of the clause. Next it reconsiders the original goal which activated the rejected clause, and tries to find a subsequent clause which also matches the goal.

Let us now return to the clauses for '*concatenated*(_,_,_)':

concatenated(*nil*, *L*, *L*).
concatenated((*X*. *L*1), *L*2, (*X*.*L*3)) :– concatenated(*L*1, *L*2, *L*3).

and see how they can be used to concatenate two lists. Suppose we wish to concatenate the lists (*a*.*b*.*nil*) and (1.2.*nil*). This will be achieved by executing the goal:

concatenated((*a*.*b*.*nil*), (1.2.*nil*), *Z*)

The result of the execution will be to substitute the required value for the variable *Z*. The goal matches only the second clause, and becomes instantiated to:

concatenated((*a*.*b*.*nil*), (1.2.*nil*), (*a*.*Z*1))

since this is the most general common instance of the original goal and the head of the matching clause. The name given to the new variable *Z*1 is arbitrary. The body of the matching clause instance gives us a new goal (or recursive procedure call):

concatenated((*b*.*nil*), (1.2.*nil*), *Z*1)

The process is repeated, a second time giving rise to a further goal:

concatenated(*nil*, (1.2.*nil*), *Z*2)

which this time matches only the first clause. Execution is now complete as there are no outstanding goals to be executed. The original goal has been instantiated to:

concatenated((*a*.*b*.*nil*), (1.2.*nil*), (*a*.*b*.1.2.*nil*))

a true boolean term. Thus the effect of the execution is to instantiate *Z* to:

(*a*.*b*.1.2.*nil*)

the term originally sought.

Here we have used '*concatenated*(⟨1⟩, ⟨2⟩, ⟨3⟩)' as a procedure which takes two 'inputs' ⟨1⟩ and ⟨2⟩ and returns one 'output' ⟨3⟩. However, the procedure is much more flexible than this. For example, if ⟨3⟩ is also provided as input, 'concatenated(_,_,_)' acts as a procedure which *checks* whether ⟨3⟩ is the concatenation

of ⟨1⟩ and ⟨2⟩. Thus execution of the goal:

concatenated((*a.nil*), (*b.nil*), (*a.nil*))

will fail whereas:

concatenated((*a.nil*), (*b.nil*), (*a.b.nil*))

will succeed.

More striking is the behaviour when only ⟨3⟩ is provided as input. For example, consider what happens when the goal:

concatenated(*L*, *R*, (*a.b.nil*))

is executed. This goal will match both clauses for 'concatenated(_,_,_)'. The first match returns an immediate result:

L = *nil*
R = (*a.b.nil*)

Notice how the result returned consists of *two* 'output' values. If this result is subsequently rejected, backtracking will cause the second possible match for the original goal to be considered. The match instantiates the top goal to:

concatenated((*a.L*1), *R*, (*a.b.nil*))

and a new goal is produced:

concatenated(*L*1, *R*, (*b.nil*))

This goal again matches both clauses. The first match produces another solution to the original goal:

L = (*a.nil*)
R = (*b.nil*)

In this way backtracking causes the procedure to generate all possible pairs of lists L and R which, when concatenated, yield (*a.b.nil*).

These examples have illustrated a number of characteristic features of Prolog procedures. Firstly, when a procedure returns, the result sent back may consist of more than one value, just as, in the conventional way, more than one value may be provided as input. Furthermore, the input and output positions do not have to be fixed in advance and may vary from one call of the procedure to another. In effect, Prolog procedures can be 'multi-purpose'. These features will play an important part in the compiler which is the main example of Prolog to be discussed later.

The Logical Variable

The flexibility of Prolog procedures can be seen as a special case of a more general phenomenon. The variable in Prolog behaves in a particularly pleasing way, which is governed by the high-level pattern matching process of unification. Let us consider a simple but somewhat artificial example using the '*concatenated*(_,_,_)' procedure. The task is to 'treble' a given list to produce a list consisting of three consecutive copies of the original, e.g.

(*a.b.c.nil*) → (*a.b.c.a.b.c.a.b.c.nil*)

One way to define this is to say that the list LLL is the treble of the list L if LLL consists of L concatenated with a list LL which is the result of concatenating L with itself, i.e.

treble(L, LLL) :–
    concatenated(L, LL, LLL),
    concatenated(L, L, LL).

In most list processing languages one would have to perform the second step first. That is, the doubled list LL would first be constructed and then another copy of L would be concatenated on the front. The same effect would be achieved in Prolog by expressing the two goals in the opposite order. However, the Prolog clause also functions perfectly well as it stands. Let us see how this is, by executing the goal:

treble((*a.b.nil*), X)

Immediately we get the pair of goals:

concatenated((*a.b.nil*), LL, X),
    concatenated((*a.b.nil*), (*a.b.nil*), LL)

The first of these goals has an uninstantiated variable as its second argument, but nevertheless the execution proceeds in the familiar way, recursing twice to hit the bottom of the recursion with the subgoal:

concatenated(*nil*, LL, $X2$)

The net result of executing this final subgoal is that LL is left uninstantiated and the original X is instantiated to:

X = (*a.b.LL*)

Thus the result of the original '*treble*' goal has been partially constructed, but the value to be returned contains the uninstantiated variable LL. Execution of the goal:

concatenated((*a.b.nil*), (*a.b.nil*), LL)

completes the picture by 'filling in' the correct value of LL:

LL = (*a.b.a.b.nil*)

Thus we get finally the correct result:

X = (*a.b.a.b.a.b.nil*)

We refer to the variable in Prolog as the 'logical' variable to draw attention to its special behaviour exemplified above. Basically there is no assignment as such in Prolog, and a variable's value, once specified, cannot be changed (except through backtracking). However, the variable's value need not be fixed immediately, and may remain unspecified for as long as is required. In particular, if a variable corresponds to a component of a data structure to be output by a procedure, the value of the variable can be left unspecified when the procedure 'returns'. The value may then later be filled in by another procedure in the course of the normal matching process.

The logical variable has the further necessary feature that when two uninstantiated variables are matched together, they become linked as one; any value subsequently given to one variable simultaneously instantiates the other. From a conventional programming standpoint, one can imagine a 'pointer' or 'reference' to one variable

being assigned to the other, with subsequent 'dereferencing' being carried out automatically where required.

Consider how the processing of the 'treble' example might be simulated in a conventional language (e.g. Algol–68, Pop–2, Lisp); i.e. what steps would correspond to execution of the goals:

concatenated((*a.b.nil*), *LL*, *X*),
concatenated((*a.b.nil*), (*a.b.nil*), *LL*)

in that order? The effect of the first goal would have to be simulated by creating a new list (*a.b.dummy*) with an arbitrary value '*dummy*' as the remainder of the list. This list would be assigned to the variable *X* and a pointer to the location containing the arbitrary value would be assigned to *LL*. For the second goal, one would create the list (*a.b.a.b.nil*) and assign it to the location indicated by the pointer previously assigned to *LL*. In this way the arbitrary value 'dummy' would be overwritten to complete (*a.b.a.b.a.b.nil*) as the value of *X*. In the style of Algol–68, these steps might be written as:

list *dummy*;
ref list *LL* := *dummy*;
ref list *X* := *concatenate*([*a*, *b*], *LL*);
value of *LL* := *concatenate*([*a*, *b*], [*a*, *b*]);

where the arguments and result of procedure '*concatenate*' are of mode '**ref list**'.

The original Prolog version achieves the same effect, but without the programmer having to bother about assignments and references. In fact it is the Prolog system which takes care of these machine-oriented details. The Prolog programmer understands the 'treble' procedure primarily from its declarative reading; from the declarative point of view, even the order of the two goals is irrelevant, let along the procedural details involved in execution.

Prolog programming requires a certain change of outlook on the part of the programmer, but this is soon acquired with a little practice. The programmer comes to appreciate that Prolog's logical variable provides much of the power of assignment and references, but in a higher-level, easier-to-use form. In a similar way, the disciple of 'structured programming', working with a conventional language, finds that 'well-structured' control primitives leave little need for **goto** and that the program is generally easier to understand if **gotos** are avoided.

An Example—Looking up Entries in a Dictionary

To complete this introduction to Prolog, we will now consider an example which will have application in compiler writing. The example involves the data type 'dictionary' introduced earlier. A dictionary will provide an efficient representation of a set of pairs of *names* with *values*. Thus the dictionary:

dic(⟨*name*⟩, ⟨*value*⟩, ⟨*dic*-1⟩, ⟨*dic*-2⟩)

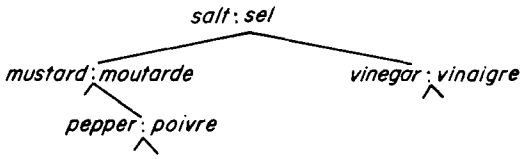
pairs ⟨*name*⟩ with ⟨*value*⟩, together with all the pairings provided by subdictionaries ⟨*dic*-1⟩ and ⟨*dic*-2⟩. We assume that the dictionary is *ordered*, so that all names in ⟨*dic*-1⟩ are before ⟨*name*⟩, and all in ⟨*dic*-2⟩ are after, and both ⟨*dic*-1⟩ and ⟨*dic*-2⟩ are themselves ordered. (Thus no names can be repeated in an ordered dictionary.) The actual ordering relation is arbitrary, but may be thought of as alphabetical order.

Ordering relationships will be expressed using the familiar symbol '<' for the 2-place predicate 'is before' and '>' for 'is after'.

As an example, the following is an (alphabetically) ordered dictionary pairing English words with their French equivalents:

dic(*salt*, *sel*,
  dic(*mustard*, *moutarde*,
    void,
    dic(*pepper*, *poivre*, *void*, *void*)),
  dic(*vinegar*, *vinaigre*, *void*, *void*))

This term is more easily visualized as the tree structure:



Because our dictionaries are ordered, it is possible to find quickly the value (if any) associated with a given name, without searching through the entire dictionary. So let us now write a Prolog procedure to 'look-up' a name in a dictionary and find its paired value. The predicate defined will be

lookup(⟨1⟩, ⟨2⟩, ⟨3⟩)

meaning '*name* ⟨1⟩ is paired with *value* ⟨3⟩ in *dictionary* ⟨2⟩'. Given a dictionary:

dic(⟨*name*⟩, ⟨*value*⟩, ⟨*dic*-1⟩, ⟨*dic*-2⟩)

we clearly have to distinguish three cases. If the name sought is ⟨*name*⟩ itself, then the required value is simply ⟨*value*⟩, i.e.

lookup(*Name*, *dic*(*Name*, *Value*,_,_), *Value*).

Note the use of two 'anonymous' variables for the components of the dictionary which are not relevant to this case. In the other two cases, we have to look for the required name in one of the two subdictionaries of the initial dictionary. If the name sought is before ⟨*name*⟩, then we must look in the first subdictionary, i.e.

lookup(*Name*, *dic*(*Name* 1,_, *Before*,_), *Value*) :-
  Name < *Name* 1, *lookup*(*Name*, *Before*, *Value*).

A similar clause deals with the case where the name sought is after ⟨*name*⟩; i.e.

lookup(*Name*, *dic*(*Name* 1,_,_,*After*), *Value*) :-
  Name > *Name* 1, *lookup*(*Name*, *After*, *Value*).

We have explained these clauses in a procedural way, having in mind the particular goal of looking up a given name in a given dictionary to find an unknown value. The control information built into the Prolog clauses reflects this aim, i.e. the order of the clauses, and the order of the goals in the body of each clause, is chosen to be appropriate for the type of goal in mind. Thus, of the three clauses, it is natural to consider the first clause first, since it may give an immediate result without further recursive procedure calls. Again, in the last two clauses, it is sensible to make the test comparing the order of

names as the first goal in each clause, since then the recursive call of '*lookup*' will only be made on the appropriate subdictionary.

Note that the control information is not strictly essential; if a different clause and goal ordering were used, valid results would still eventually be obtained, but the '*lookup*' procedure would not go 'straight' to the required result—without the right control, the procedure would perform an extremely wasteful exploration of irrelevant parts of the dictionary.

How can one be so sure that valid results will be obtained whatever the control information? The reason is that the clauses for '*lookup*' have a proper declarative interpretation, and the Prolog execution mechanism is guaranteed only to produce answers which accord with the declarative interpretation. Although we explained the clauses 'procedurally', they can be understood entirely declaratively as simple statements about dictionaries. For example, the third clause might be read as:

'If a name *Name* has a value *Value* in a dictionary called *After*, and *Name* 1 is a name which is ordered earlier than *Name*, then *Name* has value *Value* in any dictionary of the form '*dic*(*Name* 1,_,_,*After*)' '.

Of course, the statement would still be true if the condition on the order of *Name* and *Name* 1 were omitted. As it stands, the statement is true, but less general than it might be. However, if attention is restricted to ordered dictionaries, the three clauses for '*lookup*' are sufficiently general to cover all possible instances of the '*lookup*' relationship. It is generally desirable in Prolog programming to make the logical statements comprising the program no more general than is necessary to give just the truths required. In this way, the Prolog system is prevented from considering irrelevant alternatives. This principle could be thought of as a further form of control information—the system's attention is directed (in fact, restricted) to a small but adequate subset of all the correct statements which could be made.

A SIMPLE COMPILER WRITTEN IN PROLOG

Overview

Let us now look at how Prolog can be applied to the task of writing a compiler. We shall only consider a simplified example. Imagine we require a compiler to translate from a small Algol-like language to the machine language of a typical one-accumulator computer. The source language has assignment, IF, WHILE, READ and WRITE statements plus a selection of arithmetic and comparison operators restricted

Table I. Target language instructions

(1) Arithmetic etc. literal op.	(2) Arithmetic etc. memory op.	(3) Control transfer	(4) Input–output etc.
ADDC	ADD	JUMPEQ	READ
SUBC	SUB	JUMPNE	WRITE
MULC	MUL	JUMPLT	HALT
DIVC	DIV	JUMPGT	
LOADC	LOAD	JUMPLE	
	STORE	JUMPGE	
		JUMP	

to integers. (A BNF grammar of the language appears later.) The target language instructions are given in Table I. Each instruction has one (explicit) operand which *either* (1) is an integer constant, or (2) is the address of a storage location, or (3) is the address of a point in the program, or (4) is to be ignored. Most of the instructions also have a second implicit operand which is either the accumulator or its contents. In addition, there is a pseudo-instruction BLOCK which reserves a number of storage locations as specified by its integer operand.

As an illustration of the compiler's function, here is a simple source program (to compute factorials):

```
READ VALUE;
COUNT := 1;
RESULT := 1;
WHILE COUNT<VALUE DO
    (COUNT := COUNT+1;
     RESULT := RESULT*COUNT);
WRITE RESULT
```

Table II is the straightforward translation into machine language which the compiler will produce. (The columns headed *symbol* are not part of the compiler's output and are merely comments for the reader.)

Table II. Instruction translation into machine language

symbol:	*address*	*instruction*	*operand*	*:symbol*
	1	READ	21	VALUE
	2	LOADC	1	
	3	STORE	19	COUNT
	4	LOADC	1	
	5	STORE	20	RESULT
LABEL1	6	LOAD	19	COUNT
	7	SUB	21	VALUE
	8	JUMPGE	16	LABEL2
	9	LOAD	19	COUNT
	10	ADDC	1	
	11	STORE	19	COUNT
	12	LOAD	20	RESULT
	13	MUL	19	COUNT
	14	STORE	20	RESULT
	15	JUMP	6	LABEL1
LABEL2	16	LOAD	20	RESULT
	17	WRITE	0	
	18	HALT	0	
COUNT	19	BLOCK	3	
RESULT	20			
VALUE	21			

Compilation will be performed in five stages (see Figure 1) of which we shall only look at the middle three.

The first stage, lexical analysis, involves grouping the characters of the source text into a list of basic symbols called 'tokens' (represented by Prolog atoms and integers).

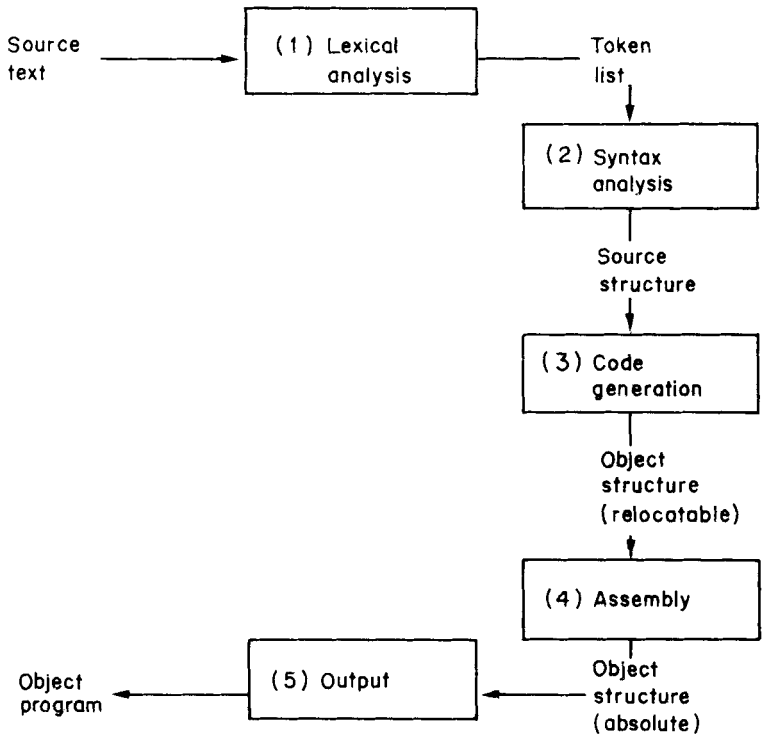


Figure 1. Compilation stages

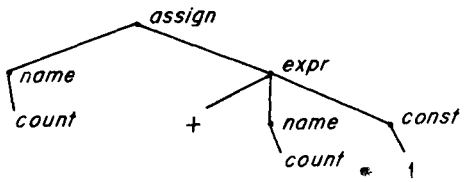
This stage is relatively uninteresting and will not be discussed further. The second stage, syntax analysis, is responsible for parsing the token list. Essentially, the effect of the analysis is to recognize the abstract program structure encoded in the characters of the source text and give this structure a name. The name will be a Prolog term. For example, the name of the statement:

COUNT := COUNT+1

will be:

assign(*name*(*count*), *expr*(+, *name*(*count*), *const*(1)))

which can also be pictured as the tree:



Since the syntax analysis stage is not our main topic, the discussion of it will be postponed to a later section.

The third stage, code generation, produces the basic structure of the object program, but machine addresses are left in a 'symbolic' form. These addresses are computed and filled in by the fourth stage, assembly.

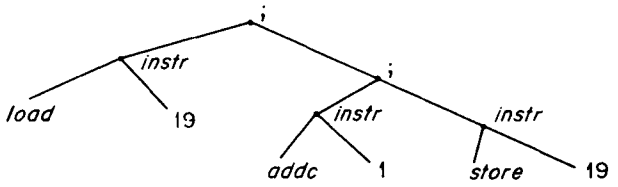
We shall not go into the less interesting final stage of outputting an actual obj program (as a bit string say). The result of the assembly stage will be a Prolog te which names the object program structure. For example, the name for:

```
LOAD  19
ADDC  1
STORE  19
```

will be:

(instr(load, 19); instr(addc, 1); instr(store,19))

where the binary functor ';' has been written as a right-associating infix operator, the term can be pictured as:



Note that the ';' functor is only used to indicate sequencing, and that the same seque can be expressed by different terms, e.g.

(a; (b; c)) and *((a; b); c)*

Compiling the Assignment Statement

Consider first the problem of compiling the assignment statement:

⟨*name*⟩ := ⟨*expression*⟩

The code for this will have the form:

⟨*expression code*⟩
STORE ⟨*address*⟩

where ⟨*expression code*⟩ is the code to evaluate the arithmetic expression ⟨*expressi* yielding a result in the accumulator. The STORE instruction stores this resul ⟨*address*⟩, the address of the location named ⟨*name*⟩.

We want to make this semi-formal specification precise by translating it into a Pr clause. Now the Prolog term which names the source form is:

assign(name(X), Expr)

where *X* and *Expr* are Prolog variables which correspond to the BNF non-termi ⟨*name*⟩ and ⟨*expression*⟩ in the semi-formal specification. Similarly, a Prolog t naming the target form is:

(Exprcode; instr(store, Addr))

where *Exprcode* and *Addr* are Prolog variables corresponding to ⟨*expression code*⟩ ⟨*address*⟩. We have to define the relationship between *X*, *Expr*, *Exprcode* and *A* Suppose the source language names are to be mapped into machine addresse accordance with a dictionary *D*. Then one necessary condition is expressed by

Prolog goal:

lookup(*X*, *D*, *Addr*)

The condition relating *Expr* and *Exprcode* may be expressed by the goal:

encodeexpr(*Expr*, *D*, Exprcode)

where the meaning of the predicate '*encodeexpr*(⟨1⟩, ⟨2⟩, ⟨3⟩)' is '⟨3⟩ is the code for the expression ⟨1⟩ conforming to dictionary ⟨2⟩'. If '*encodestatement*(⟨1⟩, ⟨2⟩, ⟨3⟩)' is a similar predicate meaning '⟨3⟩ is the code for the statement ⟨1⟩ conforming to dictionary ⟨2⟩', then the complete Prolog clause we require is:

```
encodestatement(assign(name(X), Expr), D,
        (Exprcode;
        instr(store, Addr))
):-
   lookup(X, D, Addr),
   encodeexpr(Expr, D, Exprcode).
```

All we have done so far is to make precise the informal rule for compiling an assignment statement. Now the resulting clause is not only an exact statement of the rule, but will also actually be the part of the compiler responsible for implementing the rule. The clause represents one case of the procedure '*encodestatement*(⟨1⟩, ⟨2⟩, ⟨3⟩)' which takes as input a statement ⟨1⟩ and a dictionary ⟨2⟩ and produces as output object code ⟨3⟩.

If we regard the clause as just a statement of a rule, the ordering of the two goals in the body of the clause is irrelevant. Now usually the order is very important when we want also to use the clause as part of a practical procedure. However in this case, as for many of the other clauses which make up the compiler, it will become clear that the clause will function perfectly well whichever order is chosen.

Compiling Arithmetic Expressions

We already know the clauses for '*lookup*', so let us move on to the clauses for '*encodeexpr*'. For reasons which will become clearer later, '*encodeexpr*' is defined in terms of another predicate:

```
encodeexpr(Expr, D, Code) :-
    encodesubexpr(Expr, 0, D, Code).
```

The extra (integer) argument of '*encodesubexpr*' provides information about the context in which the expression occurs, and is zero unless the expression is a subexpression of another expression. Let us now look at the clauses for '*encodesubexpr*' and see how they embody rules for translating the different types of arithmetic expression.

If the expression is just a constant ⟨*const*⟩ then the instruction:

LOADC ⟨*const*⟩

has the desired effect of loading the constant into the accumulator. Similarly, if the expression is a location named ⟨*name*⟩ then the instruction:

LOAD ⟨*addr*⟩

loads the current value of the location, where ⟨*addr*⟩ is the location's address. These

two rules are expressed in Prolog by the clauses:

```
encodesubexpr(const(C),_,_, instr(loadc, C) ).

encodesubexpr(name(X),_,D, instr(load, Addr) ):-
  lookup(X, D, Addr).
```

The final possibility is a composite expression of the form:

⟨*expression* 1⟩⟨*operator*⟩⟨*expression* 2⟩

If ⟨*expression* 2⟩ is simply a constant or location name, the code generated for the composite expression takes the form:

⟨*expression* 1 *code*⟩
⟨*instruction*⟩

where ⟨*expression* 1 *code*⟩ is the translation of ⟨*expression* 1⟩ and ⟨*instruction*⟩ is the appropriate machine instruction which applies ⟨*operator*⟩ to the value in the accumulator and operand ⟨*expression* 2⟩. For example:

⟨*expression*⟩ + 7

translates to:

⟨*expression code*⟩
ADDC 7

The clauses which express this more generally are:

```
encodesubexpr(expr(Op, Expr1, Expr2), N, D,
        (Expr1 code;
        Instruction)
):-
  apply(Op, Expr2, D, Instruction),
  encodesubexpr(Expr1, N, D, Expr1 code).

apply(Op, const(C),_, instr(Opcode, C) ):-
  literalop(Op, Opcode).

apply(Op, name(X), D, instr(Opcode, Addr) ):-
  memoryop(Op, Opcode),
  lookup(X, D, Addr).

literalop(+, addc).     memoryop(+, add).
literalop(-, subc).     memoryop(-, sub).
literalop(*, mulc).     memoryop(*, mul).
literalop(/, divc).     memoryop(/, div).
```

Notice how the information residing in the clauses for '*literalop*' and '*memoryop*' would conventionally be treated as tables of data rather than procedures.

(The following covers the more general case where ⟨*expression* 2⟩ is composite, and may be skipped on first reading.)

In this more general case, the code will have to be of the form:

⟨*expression* 2 *code*⟩
STORE ⟨*temporary*⟩
⟨*expression* 1 *code*⟩
⟨*op-code*⟩⟨*temporary*⟩

where ⟨*expression* 2 *code*⟩ evaluates ⟨*expression* 2⟩ and the result is stored temporarily at address ⟨*temporary*⟩; ⟨*expression* 1⟩ is then evaluated and the instruction of type ⟨*op-code*⟩ applies ⟨*operator*⟩ to the pair of values respectively contained in the accumulator and previously stored at location ⟨*temporary*⟩. Note that if ⟨*expression* 1 *code*⟩ itself requires temporary storage locations, these must all be different from ⟨*temporary*⟩. These requirements are met by the clause:

```
encodesubexpr(expr(Op, Expr1, Expr2), N, D,
        (Expr2code;
        instr(store, Addr);
        Expr1code;
        instr(Opcode, Addr))
):-
   complex(Expr2),
   lookup(N, D, Addr),
   encodesubexpr(Expr2, N, D, Expr2code),
   N1 is N+1,
   encodesubexpr(Expr1, N1, D, Expr1code),
   memoryop(Op, Opcode).

complex(expr(_,_,_)).
```

(Here the goal '$N1$ is $N+1$' means '$N1$ is the value of the arithmetic expression $N+1$'.) The procedure's extra argument N is an integer which is used as a name to be looked up in the dictionary D. In this way the compiler uses integers as 'private' names for the temporary storage locations it requires. In other respects, temporaries are treated just like any other locations defined in the actual source program, and are recorded in the same dictionary. Notice how any temporaries required for the evaluation of ⟨*expression* 1⟩ are named by the integers $N+1$, $N+2$, etc., and thus are distinct from the temporary named by the integer N which is used to preserve the previously calculated value of ⟨*expression* 2⟩ while ⟨*expression* 1⟩ is being evaluated.

Compiling the Other Statement Types

Now let us consider a statement type which is, in itself, slightly more complex to compile—the IF statement:

IF ⟨*test*⟩ THEN ⟨*then*⟩ ELSE ⟨*else*⟩

The code for this will take the form:

⟨*test code*⟩
⟨*then code*⟩
JUMP ⟨*label* 2⟩
⟨*label* 1⟩:
⟨*else code*⟩
⟨*label* 2⟩:

where ⟨*test code*⟩ causes a jump to ⟨*label* 1⟩ if the test proves false. As in an assembly language program, we have used labels to indicate the instructions whose addresses are ⟨*label* 1⟩ and ⟨*label* 2⟩.

The Prolog formulation of this is:

```
encodestatement(if(Test, Then, Else), D,
        (Testcode;
         Thencode;
         instr(jump, L2);
       label(L1);
         Elsecode;
       label(L2))
):-
   encodetest(Test, D, L1, Testcode),
   encodestatement(Then, D, Thencode),
   encodestatement(Else, D, Elsecode).
```

Notice that the clause does not fix the addresses $L1$ and $L2$, but merely indicates constraints on their values through labelling the object code. One can think of the output from the procedure '*encodestatement*' as being relocatable code. The output term will contain free variables L1 and L2 whose values will not be fixed until stage 4 of compilation—the assembly stage. This is an example of the use of the logical variable to delay specifying certain parts of a data structure.

The clauses for '*encodetest*' are as follows:

```
encodetest(test(Op, Arg1, Arg2), D, Label,
        (Exprcode;
        instr(Jumpif, Label))
):-
   encodeexpr(expr(-, Arg1, Arg2), D, Exprcode),
   unlessop(Op, Jumpif).

unlessop(=, jumpne).
unlessop(<, jumpge).
unlessop(>, jumple).
unlessop(≠, jumpeq).
unlessop(≤, jumpgt).
unlessop(≥, jumplt).
```

The test is effected by computing the difference of the two operands to be compared, and then applying a conditional jump instruction. '*Label*' is the address to jump to if the test fails. The meaning of the clauses should be clear by analogy with cases previously discussed.

The clauses for translating the remaining statement types are as follows:

```
encodestatement(while(Test, Do), D,
        (label(L1);
           Testcode;
           Docode;
           instr(jump, L1);
         label(L2))
```

```
):-
   encodetest(Test, D, L2, Testcode),
   encodestatement(Do, D, Docode).

encodestatement(read(name(X)), D, instr(read, Addr) ):-
   lookup(X, D, Addr).

encodestatement(write(Expr), D,
        (Exprcode;
        instr(write, 0))
):-
   encodeexpr(Expr, D, Exprcode).

encodestatement((S1; S2), D, (Code1; Code2) ):-
   encodestatement(S1, D, Code1),
   encodestatement(S2, D, Code2).
```

Notice how the 'serial' statement:

⟨*statement* 1⟩; ⟨*statement* 2⟩

is treated as just another statement type.

Constructing the Dictionary

Now that we have considered all the elements of the source language, it remains to describe how a program as a whole is compiled. Many of the clauses already stated have referred to a common dictionary *D*. So far we have tacitly assumed that this dictionary (or symbol table) has been constructed in advance and supplied as 'input' to each procedure which translates source language constructs. Now it happens that, with a little care, we can arrange for the dictionary to be built up in the course of the main translation process (stage 3). The clauses for '*lookup*' not only do the job of consulting existing dictionary entries, but will also serve to insert new entries as required. In fact '*lookup*' is a good example of a 'multi-purpose' procedure. Its very useful and rather remarkable behaviour depends on the full flexibility of the logical variable.

Let us first restate the '*lookup*' clauses (with a slight change in the first clause, to be discussed shortly):

```
lookup(Name, dic(Name, Value,_,_), Value):-!.
lookup(Name, dic(Name1,_, Before,_), Value):-
   Name<Name1, lookup(Name, Before, Value).
lookup(Name, dic(Name1,_,_, After), Value):-
   Name>Name1, lookup(Name, After, Value).
```

To see how the '*lookup*' procedure can be used to create a dictionary, consider the effect of executing the goals:

```
lookup(salt, D, X1),
lookup(mustard, D, X2),
lookup(vinegar, D, X3),
lookup(pepper, D, X4),
lookup(salt, D, X5)
```

in that order, assuming all the variables are initially uninstantiated, even the dictionary argument D. One can interpret these goals as saying: 'construct a dictionary D such that "*salt*" is paired with $X1$ and "*mustard*" is paired with $X2$ and ...'. The first goal is immediately solved by the first clause for '*lookup*' giving:

$$D = dic(salt, X1, D1, D2)$$

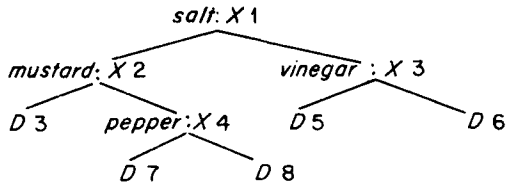
and leaving $X1$ uninstantiated. Thus variable D is now instantiated to a partially specified dictionary. The second of the original goals is executed next. Execution proceeds initially as for a normal call of '*lookup*' and produces the recursive call:

$$lookup(mustard, D1, X2)$$

Now, since $D1$ is uninstantiated, this goal is solved immediately, giving:

$$D = dic(salt, X1, dic(mustard, X2, D3, D4), D2)$$

In this way '*lookup*' is inserting new entries in a partially specified dictionary. By the time of executing the fifth of the original goals, D is instantiated to a dictionary which may be pictured as:



The effect of the fifth goal is to leave the dictionary unaltered; the only result is the instantiation:

$$X5 = X1$$

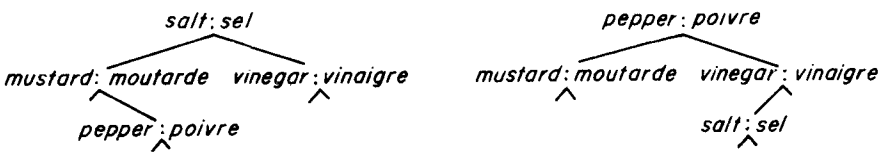
Thus both values paired with '*salt*' are guaranteed to be the same.

We have seen that:

1. The dictionary can be built up as we go along, starting from a free variable, and with free variables as the terminal nodes of the dictionary at every stage;
2. The values which are paired with the names in the dictionary can be left unspecified until later—their places are taken by variables, and different variables representing the same value will be identified where necessary.

As used in the compiler, the '*lookup*' procedure builds up a dictionary associating storage location names with free variables representing their addresses. These addresses are only filled in during the assembly stage of compilation.

We shall now consider the meaning of, and reason for, the extra '!' in the first clause of '*lookup*'. A fundamental reason for the change is that an ordered dictionary for a given set of pairings is not unique. For example, the two ordered dictionaries diagrammed below embody the same set of associations:



In theory, the '*lookup*' procedure could choose to build either of these, or any other equivalent dictionary. This is reflected in the fact that a '*lookup*' goal such as:

lookup(*salt*, *D*, *X*1)

with an uninstantiated variable as second argument will match not only the first clause for '*lookup*' but also either of the other two. These alternative matches in principle allow different but equivalent forms of dictionary to be constructed.

Obviously we wish to limit the choice to just one of these equivalent forms. Moreover, the generation of alternative forms may be highly inefficient, if not impossible. This is because a match of, say,

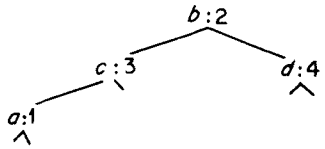
lookup(*salt*, *D*, *X*1)

against the second clause gives rise to the goal:

salt < *Name*1

with *Name*1 uninstantiated. Now in theory this goal should generate any name which is ordered later than '*salt*'. In practice, it is highly undesirable to execute a goal with such a large set of alternative solutions, and the actual implementation of '<' may well be such as to make the goal impossible to execute.

The alert reader will also have noticed that the declarative meaning of the clauses for '*lookup*' does not guarantee a dictionary of the type we require—that is, an ordered dictionary (with no name repeated). For example, if ⟨*dic*⟩ is the dictionary pictured as:



then '*lookup*(*a*, ⟨*dic*⟩, 1)' is true, but ⟨*dic*⟩ is not ordered. Strictly speaking, a check should be made somewhere in the compiler that the dictionary created and used during compilation is indeed ordered. This check is tiresome and in practice unnecessary.

All of these various potential drawbacks to the 'creative' use of '*lookup*' are circumvented by inserting the *cut* operator '!' as a pseudo-'goal' in the first clause. The cut operator is an additional control device provided by Prolog, which should be ignored when reading a clause declaratively. (With certain usages of cut, there is no meaningful declarative reading for the 'clause'; however this does not apply to any of the clauses in this paper.) When a 'cut' pseudo-goal has been executed, if backtracking should later return to that point, the effect is to fail immediately the 'parent' goal, i.e. the goal which activated the clause containing the cut. In other words, the cut operation commits the Prolog system to all choices made since the parent goal was invoked. For our '*lookup*' procedure, the cut means 'if a match is obtained against the first clause, don't ever try any of the subsequent clauses'.

Given Prolog's procedural semantics, it is not difficult to see how the qualification expressed by the cut symbol ensures that '*lookup*' constructs a unique ordered dictionary starting from an initially uninstantiated variable. The dictionary is 'unique' except that the terminal nodes are free variables which really represent unspecified subdictionaries. All of these variables must finally be instantiated to '*void*' in order to obtain the smallest possible dictionary meeting the required conditions.

Compiling The Whole Program, And The Assembly Stage

The translation of a complete source program (or, rather, its abstract structure) into an object program (structure) with absolute addresses is expressed by the following clause:

```
compile(Source,
        (Code;
        instr(halt, 0);
        block(L))
):-
    encodestatement(Source, D, Code),
    assemble(Code, 1, N0),
    N1 is N0+1,
    allocate(D, N1, N),
    L is N-N1.
```

(A goal, such as '*L is N−N1*' above, of the form '⟨*var*⟩ is ⟨*expr*⟩' means that ⟨*var*⟩ is the value of the arithmetic expression ⟨*expr*⟩.) The result of compiling the program *Source* is a sequence of instructions *Code* followed by a HALT instruction and then a block of storage for the variables used in *Source*. Unlike most of the compiler clauses described so far, the particular order of the goals in this clause is essential control information.

Stage 3 of compilation (relocatable code generation) is represented by the goal:

encodestatement(*Source*, *D*, *Code*)

Observe that when this goal is invoked, dictionary *D* is completely unspecified, i.e. *D* is still a free variable. So stage 3 really returns two outputs—the code and the dictionary.

Strictly speaking, for logical soundness, the clause for '*compile*' should contain an extra goal, say:

ordereddictionary(*D*)

to check that *D* is indeed an ordered dictionary. We may imagine this goal being inserted after the '*encodestatement*' goal. However, as noted previously, this check can be dispensed with in practice.

At the end of stage 3, *Code* still contains many free variables—representing the yet to be specified addresses of writeable locations and labelled instructions. Thus stage 3 makes extensive use of the full power of the logical variable to delay fixing of addresses until stage 4. The goal:

assemble(*Code*, 1, *N*0)

computes the addresses of labelled instructions and returns *N*0, the address of the end of *Code*. *N*1 is therefore the address of the start of the block of storage locations. The goal:

allocate(*D*, *N*1, *N*)

is responsible for laying out the storage required for the source language symbols contained in dictionary *D*. It fills in the corresponding addresses and returns *N*, the address of the end of the storage block. Finally, the length *L* of the storage block is calculated from *N* and *N*1.

The procedure for '*assemble*' is neat and simple:

```
assemble((Code1; Code2), N0, N):-
  assemble(Code1, N0, N1),
  assemble(Code2, N1, N).
assemble(instr(_,_), N0, N):- N is N0+1.
assemble(label(N), N, N).
```

Note that '*assemble*(⟨1⟩, ⟨2⟩, ⟨3⟩)' means that ⟨2⟩ is the start address and ⟨3⟩ the end address of the sequence of instructions ⟨1⟩.

The procedure for '*allocate*' has a similar character:

```
allocate(void, N, N):-!.
allocate(dic(Name, N1, Before, After), N0, N):-
  allocate(Before, N0, N1),
  N2 is N1+1,
  allocate(After, N2, N).
```

Observe that the layout of the source symbols will be in dictionary order.

Note that the dictionary input to '*allocate*' from '*encodestatement*' is incomplete in the sense that the terminal nodes are still variables. The '*allocate*' procedure in fact chooses the smallest possible dictionary, i.e. the one which contains only symbols actually occurring in the source program. If it chose otherwise, the object program would still be correct but would contain extra unused storage locations. The proper choice is achieved by placing the clause for the '*void*' case first, with a cut '!' to prevent any possibility of backtracking considering other alternatives, cf. the use of cut in '*lookup*'.

We have now looked at all the clauses needed to perform the code generation and assembly stages of the compiler. Except where otherwise noted, the particular order in which these clauses are stated is unimportant, i.e. the performance will be virtually the same whichever order is chosen.

Syntax Analysis

We shall now show very briefly how the parser, or syntax analysis stage of compilation, is programmed in Prolog. Reference 3 gives a much fuller introduction to the basic method we use for writing parsers in logic. The theory of this method is described by Colmerauer[2] from whom the technique originated. The clauses we require are closely related to the following BNF grammar of the source language:

```
⟨program⟩      ::= ⟨statements⟩
⟨statements⟩   ::= ⟨statement⟩ |
                   ⟨statement⟩; ⟨statements⟩
⟨statement⟩    ::= ⟨name⟩:=⟨expr⟩ |
                   IF ⟨test⟩ THEN ⟨statement⟩ ELSE ⟨statement⟩ |
                   WHILE ⟨test⟩ DO ⟨statement⟩ |
                   READ ⟨name⟩ |
                   WRITE ⟨expr⟩ |
                   (⟨statements⟩)
⟨test⟩         ::= ⟨expr⟩⟨comparison op⟩⟨expr⟩
⟨expr⟩         ::= ⟨expr⟩⟨op 2⟩⟨expr 1⟩ |
                   ⟨expr 1⟩
```

⟨*expr* 1⟩ : : = ⟨*expr* 1⟩⟨*op* 1⟩⟨*expr* 0⟩ |
⟨expr 0⟩
⟨*expr* 0⟩ : : = ⟨*name*⟩ |
⟨*integer*⟩ |
(⟨*expr*⟩)
⟨*comparison op*⟩ : : = = | < | > | ≤ | ≥ | ≠
⟨*op* 2⟩ : : = * | /
⟨*op* 1⟩ : : = + | −

The essential idea behind the translation into Prolog is that a BNF non-terminal becomes a predicate of three arguments:

⟨*non-terminal*⟩(⟨*start*⟩, ⟨*end*⟩, ⟨*name*⟩)

meaning 'the token list ⟨*start*⟩ commences with a phrase of type ⟨*non-terminal*⟩ ending at a point where the list of remaining tokens is ⟨*end*⟩; the structure of the phrase is identified by ⟨*name*⟩'. Now because the grammar contains some left recursive rules, and for other efficiency reasons, parts of the grammar are rewritten to facilitate left-to-right top-down parsing. Some of the resulting predicates have to be given an additional argument which is the name of the preceding phrase. For example:

restexpr(⟨*n*⟩, ⟨*start*⟩, ⟨*end*⟩, ⟨*name*0⟩, ⟨*name*⟩)

means that the token list ⟨*start*⟩ commences with the remainder of an arithmetic expression of precedence ⟨*n*⟩ ending at ⟨*end*⟩ and the whole expression is named ⟨*name*⟩ if the preceding subexpression is named ⟨*name*0⟩. Here then is the Prolog translation of the BNF grammar:

```
program(Z0, Z, X) :- statements(Z0, Z, X).

statements(Z0, Z, X) :- statement(Z0, Z1, X0), reststatements(Z1, Z, X0, X).

reststatements((';'.Z0), Z, X0, (X0; X)) :- statements(Z0, Z, X).
reststatements(Z, Z, X, X).

statement((V.':='.Z0), Z, assign(name(V), Expr)) :-
   atom(V), expr(Z0, Z, Expr).
statement((if.Z0), Z, if(Test, Then, Else)) :-
   test(Z0, (then.Z1), Test),
   statement(Z1, (else.Z2), Then),
   statement(Z2, Z, Else).
statement((while.Z0), Z, while(Test, Do)) :-
   test(Z0, (do.Z1), Test),
   statement(Z1, Z, Do).
statement((read.V.Z), Z, read(name(V))) :- atom(V).
statement((write.Z0), Z, write(Expr)) :- expr(Z0, Z, Expr).
statement(('('.Z0), Z, S) :- statements(Z0, (')'.Z), S).

test(Z0, Z, test(Op, X1, X2)) :-
   expr(Z0, (Op.Z1), X1), comparisonop(Op),
   expr(Z1, Z, X2).
```

```
expr(Z0, Z, X) :- subexpr(2, Z0, Z, X).

subexpr(N, Z0, Z, X) :- N > 0, N1 is N - 1,
    subexpr(N1, Z0, Z1, X0),
    restexpr(N, Z1, Z, X0, X).
subexpr(0, (X.Z), Z, name(X)) :- atom(X).
subexpr(0, (X.Z), Z, const(X)) :- integer(X).
subexpr(0, ('('.Z0), Z, X) :- subexpr(2, Z0,(')'.Z),X).

restexpr(N, (Op.Z0), Z, X1, X) :- op(N, Op), N1 is N - 1,
    subexpr(N1, Z0, Z1, X2),
    restexpr(N, Z1, Z, expr(Op, X1, X2), X).
restexpr(N, Z, Z, X, X).

comparisonop(=).
comparisonop(<).
comparisonop(>).
comparisonop(≤).
comparisonop(≥).
comparisonop(≠).

op(2, *).      op(1, +).
op(2, /).      op(1, -).
```

THE ADVANTAGES OF PROLOG FOR COMPILER WRITING

This section summarizes the particular advantages of Prolog as a language for writing compilers. Many of the advantages should be clear from the main example discussed above. It is important to take into account, not just the compiler which is the end product, but also the work which must go into initially designing and building it and into subsequently 'maintaining' it.

So let us review how one might set about constructing a compiler. Initially, the picture is just of a black box with source programs as input and correctly translated object programs as output. The first consideration is to decide how the output is related to the input. It is natural to examine the structure of the source language and to devise for each element of the language a rule for translating it into target language code. These rules form a specification of the compiler's function. The final and generally more laborious stage of compiler construction involves implementing procedures which efficiently carry out the translation process in accordance with the specification.

The major advantage of implementation in Prolog is that it is possible for the final stage to be almost trivial. For a compiler such as the sample one discussed, it is not a great exaggeration to say that

'the specification *is* the implementation'.

Thus the procedures which make up the compiler consist of clauses, each of which can generally be interpreted as a rule describing a possible translation of some particular construct of the source language into the target language. The burden of the implementation stage reduces to ensuring that the specification can be used as an

efficient implementation. This requires the addition of suitable control information (i.e. choosing the ordering of clauses and goals) and may involve some rewriting of parts of the specification to allow an efficient procedural interpretation.

The closeness of implementation and specification brings many benefits:

1. The implementation is more readable and may be virtually self-documenting.
2. The correctness (or otherwise) of the implementation is more easily apparent and the scope for error is greatly reduced. As long as each clause is a valid rule for translating the source language, one can be confident that the compiler will not generate erroneous code.
3. Compiler modifications and source language extensions are more readily incorporated, since the compiler consists of small independently-meaningful units (clauses) which are directly related to the structure of the source language.

There are a number of conventional programming language features which would normally have to be used in a compiler implementation, but which do not appear explicitly in a Prolog implementation. These include assignment, references (pointers), operations for creating data structures, operations for selecting from data structures, conditional or test instructions, and the **goto** instruction. Of course, all these features are being used implicitly, behind the scenes, by the Prolog system. In effect, the Prolog system assumes much of the responsibility for 'coding up' the implementation. This relieves the programmer of tedious details and protects him against errors commonly associated with the low-level features mentioned, e.g.

1. Referring to a non-existent component of a data structure.
2. Attempting to use the value of a variable before it has been assigned.
3. Attempting to use a value which is obsolete, such as a 'dangling reference' to storage which has been de-allocated.
4. Overwriting a value or part of a data structure which is still needed elsewhere in the program.
5. Omitting to test for a special case before dropping through to the **else** clause of a conditional.

In a conventional language, errors such as these typically produce bugs which are difficult to trace and eradicate. At best the program will halt immediately with some error message, which may or may not help the programmer to pinpoint the bug. More usually, the bug will only manifest itself later in the processing, by which time the root cause will be difficult to determine.

Such situations cannot arise with the basic Prolog language covered here, since none of the low-level features mentioned is present in the language. Moreover, the (procedural) semantics of Prolog is totally defined; a syntactically correct program is guaranteed to be legal, and is incapable of performing, or even attempting to perform, any invalid or undefined operation. If there is a 'bug' in a Prolog program, it merely means that the program, while being perfectly legal, does not do exactly what the programmer intended. The actual behaviour is entirely predictable and therefore the 'bug' is normally found relatively easily. A totally defined semantics is of great practical significance and is almost unique among programming languages.

To summarize, Prolog has the following advantages as a compiler-writing tool:

1. Less time and effort is required.
2. There is less likelihood of error.
3. The resulting implementation is easier to 'maintain' and modify.

THE PRACTICABILITY OF PROLOG FOR COMPILER WRITING

Granted that Prolog is a very congenial language for compiler writing, the question naturally arises whether an implementation in Prolog can perform well enough to be practically useful. The answer obviously depends on how efficiently Prolog itself is implemented.

The first Prolog system was an interpreter written in Fortran at the University of Marseille.[8] This proved to be surprisingly fast. More recently, building on the techniques developed at Marseille, two colleagues and I have implemented a Prolog compiler for the DECsystem–10 machine.[9, 10]

The machine code generated by this compiler is reasonably efficient and is not so very different from that which might be produced by a compiler for a conventional list- or record-processing language. The principal effect of the compilation is to translate the head of each clause into instructions which will do the work of matching against any goal. Of the two terms involved in the matching, it is the clause head which is compiled, since this is uninstantiated prior to the matching, unlike the goal. Because the variables in the head are uninstantiated prior to the matching, their first occurrences can be compiled into simple assignment operations.

The code generated for a compound term has to distinguish between two cases. If the subterm matches against a variable, a new data structure must be constructed and assigned to the variable. This case is handled by an out-of-line subroutine. The other case concerns matching against a non-variable. This is performed essentially by in-line code. It comprises a test for matching functors (record types), followed by the compiled form of each of the subterms of the compound term. This code will be responsible for accessing subcomponents of the matching data structure.

Many Prolog procedures consist of a number of clauses giving a definition by cases—each clause accounts for a different possible form of the input. This characteristic is particularly evident in compiler writing as illustrated above. For example, the clauses for '*encodestatement*' each match a different statement type. Here, and more generally, it is natural to place the principal input as first argument of the predicate. Our Prolog compiler capitalizes on this fact by compiling in a fast 'switch' or 'computed **goto**', branching on the form (principal functor) of the first argument. Thus instead of trying each clause in turn, the code automatically selects only appropriate clauses (often just one).

As far as the general efficiency of Prolog is concerned, space economy is more likely to be a limitation than speed. From our discussion of the (basic) language it is clear that the responsibility for storage management falls entirely on the system and not on the programmer. In meeting this responsibility, the Prolog compiler employs a certain degree of sophistication.

In particular, it automatically classifies variables into two types ('local' and 'global') with storage allocated from different areas analogous to the 'stack' and 'heap' of Algol–68. Local storage is recovered automatically by a conventional stack mechanism when a procedure returns, provided the procedure has no more alternative multiple results to generate through backtracking. In addition, a second stack mechanism associated with backtracking ultimately recovers *all* storage, both local and global. Thus a garbage collector is not an essential part of the system, although one is provided. This is in contrast to the situation for the 'heap' of Algol–68 and similar languages, where storage can only be recovered by the potentially very expensive process of garbage collection.

Although the automatic storage management of basic Prolog is quite effective, it is not adequate on its own for really large tasks. For example, it is currently unrealistic to expect a compiler written in basic Prolog to compile a sizable program in one step, as, unaided by the user, the storage requirements would exhaust main memory. A technique which can be adopted at present is to compile small units of the program, e.g. 'lines', 'blocks' (or in the case of Prolog itself 'clauses'), using the 'pure' methods we have described; the compiler as a whole consists of a number of 'pure' procedures linked together using more *ad hoc* (and conventional) methods. The *ad hoc* parts are written using extensions beyond the basic Prolog language, which are outside the scope of this paper. The essential feature of the 'impure' code is that use is made of further storage areas and external files, all of which have to be managed directly by the programmer. This approach to compiler writing in Prolog enables one to produce a practical implementation, large parts of which are written in the basic Prolog language, with all the advantages discussed above.

Given the theme of this paper, it should come as no surprise that the Prolog compiler is itself written in Prolog, using the very principles which are the subject of this paper. Data on this 'bootstrapped' compiler may therefore give some idea of the kind of performance attainable with Prolog as the implementation language. Note that the compiler does not attempt any sophisticated optimization.

The compiler generates about 2 machine instructions (= 2 machine words, of 36 bits each) per source symbol (i.e. constant, functor or variable). It takes typically around 10·6 seconds to generate 1000 words of code. The amount of 'short-term' (Prolog-controlled) working storage required during compilation is rarely more than 5K words, i.e. this is a normal bound on the amount of storage required to compile any one clause. (The remaining 'long-term' (programmer-controlled) working storage is needed primarily for a global symbol table, the size of which depends on the number of different functors in the program being compiled.) The total code of the compiler itself (which is not overlayed) is about 25K words.

Briefly, the performance indicated by these figures is reasonably acceptable, although naturally it falls short of what can be attained in a low-level language with efficiency as the only objective. Nevertheless, the performance is not out of line with that of certain other items of software on the DECsystem–10 (e.g. the manufacturer's assembler, 'MACRO').

Prolog is a promising language for software implementation where the main priority is to have a correctly working system available quickly, or where the system specification is liable to change. Better performance can certainly be obtained from an implementation in a lower-level language; for this, a preliminary Prolog formulation can serve as a very useful prototype. It is likely that most of the improvement will be attributable to a few relatively simple but heavily used procedures (e.g. lexical analysis, dictionary lookup), and so a mixed language approach may be an attractive possibility. An alternative view (which I favour) is to look for more sophisticated ways of compiling special types of Prolog procedure, guided probably by extra pragmatic information provided by the Prolog programmer.

ACKNOWLEDGEMENTS

I am indebted to the originators of Prolog and logic programming in general. In particular, the possibilities of Prolog for compiler writing were first recognized by

Alain Colmerauer, and the compilation method I have described is (necessarily) very similar in its essentials to the example he describes. The elegant assembly technique is directly based on his work.

My colleagues, Luis Pereira and Fernando Pereira, of the National Civil Engineering Laboratory, Lisbon, provided much encouragement and practical assistance in implementing the Prolog system referred to in the paper. Thanks are also due to them, and to Derek Brough and Chris Mellish, for helpful comments on earlier drafts of this paper.

The work was supported by SRC grant B/RG 9972.

REFERENCES

1. *R. A. Kowalski, Logic for Problem Solving, DCL Memo 75,* Department of Artificial Intelligence, University of Edinburgh, 1974. (To be published by North-Holland as part of a book of the same title.)
2. A. Colmerauer, *Les Grammaires de Métamorphose,* Groupe d'Intelligence Artificielle, Marseille–Luminy, 1975. (Appears as '*Metamorphosis Grammars*' in: L. Bolc (Ed.), *Natural Language Communication with Computers*, Springer, 1978.)
3. M. H. van Emden, *Programming with Resolution Logic*, Report CS–75–30, Department of Computer Science, University of Waterloo, Canada, 1975.
4. P. Roussel, *Prolog: Manuel de Référence et d'Utilisation*, Groupe d'Intelligence Artificielle, Marseille–Luminy, 1975.
5. L. M. Pereira, *User's Guide to DECsystem–10 Prolog*, Divisão de Informatica, Lab. Nac. de Engenharia Civil, Lisbon, 1977.
6. R. A. Kowalski, *Algorithm = Logic + Control*, Department of Computing and Control, Imperial College, London, 1977.
7. J. A. Robinson, 'A machine-oriented logic based on the resolution principle', *Journ. ACM*, **12**, 23–44 (1965).
8. G. Battani, and H. Meloni, *Interpréteur du Langage de Programmation Prolog*, Groupe d'Intelligence Artificielle, Marseille–Luminy, 1973.
9. D. H. D. Warren, L. M. Pereira and F. Pereira, *Prolog—the Language and its Implementation compared with Lisp,* Proc. ACM Symposium on Artificial Intelligence and Programming Languages, Rochester, N.Y., 1977.
10. D. H. D. Warren, *Implementing Prolog—Compiling Predicate Logic Programs*, Research Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.