PROGRAMMING WITH RESOLUTION LOGIC*

by

M.H. van Emden

Research Report CS-75-30

Department of Computer Science

University of Waterloo
Waterloo, Ontario, Canada

November, 1975

*) preprint for:

van Emden, M. H. Programming with resolution logic. Machine Representations of Knowledge (eds. E. W. Elcock and D. Michie). Dordrecht: D. Reidel Publishing Company (in press).

PROGRAMMING WITH RESOLUTION LOGIC

M.H. van Emden *)

0. Introduction

In the late sixties the newly discovered resolution principle for first-order predicate logic aroused great expectations among workers in Artificial Intelligence. As it turned out later, research in resolution theorem-proving did not at that time have a broad enough scope to support the requirements of constructing 'intelligent' and 'purposeful' computer programs. By about 1970 resolution logic had fallen into disrepute and the emphasis in AI research shifted towards the development of new program languages of very high level intended to be suitable for programming more directly than hitherto possible the 'knowledge', reasoning power, and goal-directed behaviour required by an 'intelligent' computer program.

This development has taken an interesting turn with the more recent discovery of a program language within resolution logic. The new program languages for AI research have many interesting features, but they also give the impression of being rather experimental, _ad hoc_ precursors to a single coherent language which will supercede all of them. One is tempted to speculate whether resolution logic might the basis of such a language.

*) address: Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

Whether or not this will be the case, the discovery of resolution logic as a program language is an important development in several respects;

a) resolution logic shares with some of the new program languages for AI research the property of being more descriptive, or of higher level, than conventional procedure-oriented program languages,

b) by specifying algorithms in resolution logic one can use to advantage a separation of an algorithm specification into two distinct components: the 'logic' component and the 'control' component,

c) resolution logic as a program language has a precisely defined and simple semantics,

d) in theoretical studies of computation, resolution logic as a program language is an attractive alternative to the usual formalisms such as Kleene's recursion equations, Turing machines, the formal grammars of the Chomsky hierarchy, and so on.

In this paper I am mainly concerned with a) and b). As for c), I will just state a syntax and a semantics, because this does not take much space anyway. The tempting possibilities in d) have to be left out altogether.

As for a), the first question to addressed is whether it is at all possible to use resolution logic as a program language. It seemed to me best to illustrate the affirmative answer with an example of a problem typical of conventional programming: sorting a sequence. The most important thing is to show that a resolution theorem-prover can solve straightforward problems in a straightforward way. The choice of the example should not suggest that I consider sorting a proper problem-domain for program languages of very high level.

The second question to be addressed is in what sense resolution logic is an advance with respect to conventional procedure-oriented program languages. I argue that all advances in programming, the conventional languages included, can be regarded as steps toward automatic programming. This takes me into the slippery questions of why one program language is of 'higher level' than another, or more 'descriptive', or less 'imperative'. I shall argue that this distinction can be made more precise by analyzing specifications of algorithms into a logic component and a control component and that it is precisely the possibility of doing this when programming with resolution logic that contributes to its being an advance in the development of program languages.

1. The rôle of specification language in automatic programming.

Automatic programming means the automation of program writing. To take a specific example, think of writing PL-1 programs, possibly a profitable target of automation in programming. The result of successful automation would be a machine (a "PL-1 machine") writing PL-1 (the target language) programs. Such a machine would still have to be told, no matter how automatic otherwise, what the product of its activity is expected to do, for instance in the form of a specification of input-output behaviour. Machines being what they are (for the time being), such a specification would have to be written in a formal language (specification language).

This is reminiscent of the existing situation where the writing of macnine-code programs has been automated. There exist machines which accept a "specification" in a formal language (for instance PL-1) and produce machine-code programs that are expected to comply with these specifications. This shows that, if understood in a certain way, automatic programming has been going on for a long time already. Its purpose is to produce with less effort better programs. Would a PL-1 machine achieve any progress towards this goal ?

The PL-1 machine would operate in an environment (schematically shown in figure 1), which would only make sense if specifications can be better written in specification language than in PL-1: the PL-1 machine would act as an interface between specification language and machine code. However, the PL-1 language was intended (insofar as it developed purposefully at all) as an interface between a human programmer and machine code; why should it be adopted for the other purpose ?

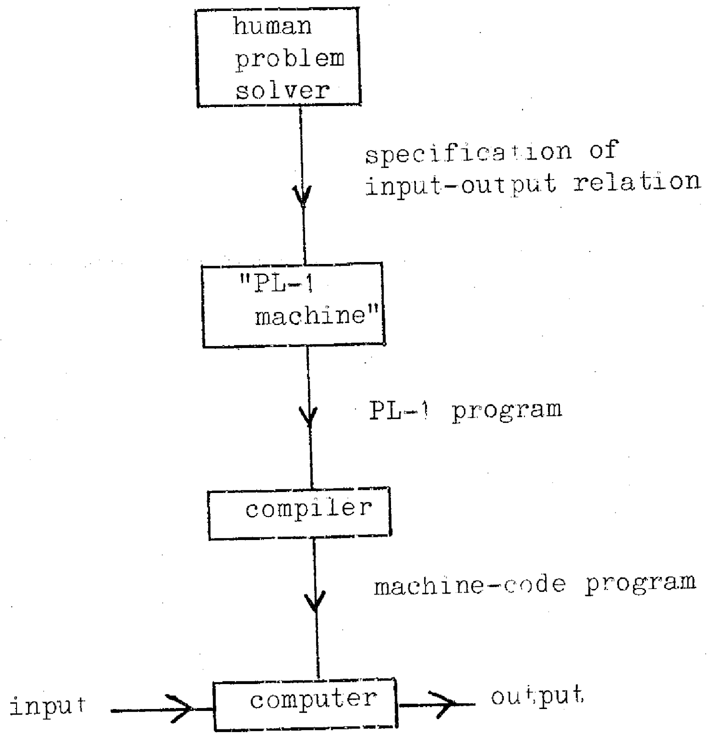


figure 1: Environment of the "PL-1 machine"

Either one needs a language intermediate between specification language and machine code and then it seems better not to adopt PL-1, but to start with a clean slate. Or one does not need any intermediate language and PL-1 is itself a candidate specification language. Again, it seems better to start from scratch and to look for a language especially suited for this purpose. Of course, these considerations apply not only to PL-1 but also to other conventional program languages. In either case automatic programming will not turn out to be an unprecedented innovation but a further step towards the use of more powerful programming tools as assemblers, interpreters, and compilers have been in the past.

2. Two aspects of algorithm specification.

The preceding observations suggest that there may not exist a clear-cut distinction between a specification language for automatic programming and a higher-level program language. They also suggest that any step towards automatic programming will be one in an ongoing evolution towards more powerful tools for computer-aided problem-solving. Indeed, the pioneers in compiler design already flew the banner of 'Automatic Programming' [1].

Although no clear-cut distinctions will emerge here, it is useful to compare two aspects of algorithm specification: the imperative aspect is typical for the lower level of programming as is the descriptive aspect for the higher level. In a machine-code program it is spelled out how things are done, but it is always very hard to see without additional explanations what is being done. This is an extreme case of an imperative specification of an algorithm. At the other extreme, in a specification it is only explained what is to be done and it is the problem of automatic programming to convert this into commands saying how.

Strictly speaking, a language like PL-1 or Algol 60 is completely imperative: every statement corresponds to commands to be executed. However, the value of such a language lies in the fact that in a well-written program it is possible to see without additional explanations what is being done: such a program has descriptive value as well as an imperative effect. Some of the imperative aspects have disappeared from the program, like the details of storage allocation and the commands involved in procedure invocation. The ABSYS language [10, 12, 13] is an interesting experiment that allows algorithms to be specified in a more descriptive manner. In this respect it was a forerunner of some of the new program languages for AI research [6].

Predicate logic is usually regarded as a purely descriptive language: at most able to express what is to be done by a program and not how to do it.

Yet, with respect to a given proof procedure, a specification in logic has implications for the imperative aspect, as will become clear by comparing with each other the two versions of the sorting example below.

Although the descriptive and imperative aspects of algorithm specification may be hard to disentangle, I think the distinction is useful for characterizing what constitutes a higher level program language: one that has less commitment to the imperative aspects of the algorithms to be specified and, by being more descriptive, is easier to write in and to understand for the human problem-solver.

3. The contributions of Green and of Kowalski

Green [14] has given a very useful definition of four different tasks in automatic programming by representing them as problems in automatic deduction. He specified the input-output behaviour of the required program as a set A of axioms in first-order predicate logic containing a predicate symbol R such that $A \models R(s,t)$ (A logically implies $R(s,t)$) if and only if the program is to give output t for input s. Thus, A defines (with respect to the predicate symbol R) a relation in the mathematical sense between inputs and outputs.

The generality of relations (as compared to functions as usually studied in mathematics) is suitable here: the required program, as a map from inputs to outputs, need not be total (an output may not exist for some inputs) and it need not be determinate (an input may be followed by any of more than one possible output). Even if the program computes a total function, there is no disadvantage in specifying it as a relation.

Green distinguishes checking, simulation, verification, and synthesis as tasks in automatic programming. He shows that an automatic theorem-prover can in principle accomplish these (given a suitable set A of axioms, not necessarily the same for each task) in the process of proving a theorem of a particular form. Figure 3.1 hows how this form determines which of the four tasks is to be carried out.

Form of theorem to be proved	Possible answers	Task
$R(a,b)$	yes no	checking
$\exists x. R(a,x)$	yes, $x = b$ no	simulation
$\forall x. R(x, g(x))$	yes no, $x = c$	verification (of program g)
$\forall x. \exists y. R(x,y)$	yes, $y = f(x)$ no, $x = c$	synthesis (of program f)

Figure 3.1: Green's tasks in automatic programming

Note in this figure that only 'synthesis' corresponds to automatic programming as described in section 1. The specification language is predicate logic and f is the synthesised program in a target language embodied in the function symbols of the specifying axioms. Synthesis appears to be a difficult problem. Before attacking it, let us pause and consider whether there is not a way around.

To find such a way, we should ask: what is the purpose of a program, and can it not be achieved in another way ? The answer is, a program is to cause computations to be done automatically on a computer and, yes, it can be done in another way: by simulation. As we see in figure3.1,for given input a, the automatic theorem-prover will produce the output b that would have been generated by the program synthesised from the axioms A. But then, why would we need the synthesised program if, for any input, we can get by simulation the required output without the program ?

The possibility is at least worth investigating, although at the time of Green's work it did not seem to be the most promising approach. In order to make simulation a practically interesting possibility, both development of theorem-proving technique and an increased understanding of the pragmatics of predicate logic were needed.

These requirements have been met in the meantime. The SL-resolution proof procedure of Kowalski and Kuehner [25], or each of several related proof procedures [26, 34], can be adapted to act like a program-language interpreter for a suitably constructed sentence. The 'procedural interpretation' for resolution logic is Kowalski's [21] great contribution to the pragmatics of first-order predicate logic.

Since several years these ideas have been realized in the PROLOG system developed by Colmerauer and his colleagues in the University of Marseille [9]. The action of the PROLOG system is in principle that of the simulation variety of automatic programming according to Green (see figure 3.1). As will be explained later, the system can also be regarded as an interpreter for a generalized Algol with backtracking.

The use of predicate logic discussed in this paper has some features in common with that of Hayes [15], who arrives at a program language by adding 'control information' to axioms of logic in order to obtain computationally favourable behaviour from a resolution proof procedure. The work of Hayes suggests a fruitful approach to the study of algorithms: to decompose the specification A of an algorithm into a 'logic' component L (to be specified by a sentence in resolution logic) and a 'control' component C, specified in some other way. Kowalski [24] has formulated the decomposition as

$$A = L + C$$

and showed an example of two algorithms whose specificatons A_1 and A_2 are related as follows:

$$A_1 = L + C_1 \quad \text{and} \quad A_2 = L + C_2$$

That is, the algorithms specified by A_1 and A_2 are different, but only in the control component. In this paper we shall see that A_1 can be a specification of a sorting algorithm according to the 'quicksort' principle and A_2 can be a specification of a permutation generator. Becuase the control components C_1 and C_2 are comparatively inconspicuous we shall see the interesting phenomenon of a 'quicksort' and a permutation generator with almost the same specification.

The example on sorting has been included to draw attention to the fact that an autonomous resolution proof procedure is capable of computationally acceptable behaviour. This is incompatible with widely-held opinions on this point. To give an example of such an opinion, I quote Hayes [15]:

> "However, there is every evidence, both practical and theoretical, that an autonomous resolution theorem-prover will never be sufficiently powerful to cope with complex problems. The practical evidence is abundant in the literature on computational logic."

I cannot make less abundant the practical evidence here alluded to. What I can do is to add some evidence for the contrary opinion that autonomous resolution proof procedures can be computationally useful. Of course, the examples in this paper do not solve complex problems. They suggest, however, that predicate logic can be used as a high-level program language. Hence an autonomous resolution proof procedure (for instance, Kowalski's LUSH system to be described in this paper), acting on a sentence in logic which is pragmatically sound according to the procedural interpretation, can cope with problems at least as complex as can be programmed in a high-level program language.

But, as a program language, resolution logic is of a higher level, more descriptive, less imperative, than languages like PL-1 or Algol 60. In fact, the reader may recognize several of the features of the new program languages for AI research [6]. Resolution logic remains a natural language for stating facts and making inferences, in addition to its newly descovered use in effectively simulating the execution of a program. All these considerations suggest that in resolution logic as powerful problem-solvers will be programmed as in any other program language for AI research.

It remains, of course, a requirement that specifications in predicate logic be pragmatically sound according to the procedural interpretation. That is part of the art of programming in logic. It need not be a superhuman achievement to express a complex problem in pragmatically sound axioms. In fact, this has been done in PROLOG with remarkably little fuss for several ambitious programming tasks. These include natural-language understanding systems [9, 33], formula-manipulation and symbolic-integration systems [3, 19], and a STRIPS-style problem-solver [37]. A comparison has been published [37] between the last-mentioned program and the original STRIPS system. In the examples tried, the PROLOG problem-solver was considerably faster. A more important advantage of PROLOG is suggested by the fact that writing and testing the program required about one man-week.

4. Resolution logic as a language for stating problems

In general, a syntax for first-order predicate logic comprises a language expressing sentences, an inference system consisting of axioms and rules of inference, and a proof procedure relating the use of the inference system to the sentence to be proved. The usual language and inference system ('classical first-order predicate logic') are found, for instance, in [20].

For the purpose of automatic deduction a different syntax has evolved. Its language is the 'clausal form' of first-order predicate logic. Its inference system is the one invented by J.A. Robinson [35], using his 'resolution principle' and his 'unification algorithm'. I will refer to the language of clausal form together with the resolution rule inference as 'resolution logic'.

Resolution logic has been criticized as being unsuitable for problem-solving because of an alleged inherent lack of goal-directedness in its deductions and for the alleged difficulty, or impossibility, of taking domain-specific knowledge into account for the control of the course of a deduction. Now the control is in the domain of the proof procedure. No particular proof procedure is by necessity associated with resolution logic. The criticisms mentioned above are possibly based on a mistaken belief that proof procedures associated with resolution logic must be of the 'uniform' or 'saturation' variety.

Several researchers [4,5,21,28,29,36] have realized the rich variety of possible proof procedures which can be used in resolution logic. They developed procedures guided by heuristic principles or by the desirability of a procedure being guided by domain-specific knowledge. In this paper I will be concerned with one of the approaches mentioned, namely Kowalski's.

Resolution logic can be explained in terms of classical logic, as is done in [32]. I believe that resolution logic has advantages other than the one of being suitable for use on a computer: for example its simplicity, which makes it also suitable for a self-contained definition of language and semantics. A pragmatics will be given in terms of classical logic.

4.1 A language for resolution logic (see [21])

A sentence is a set of clauses. A clause is a set of literals L_i written as

$$L_1 \ldots L_n$$

except when the set is empty: it is then called the null clause and is written as $\square$

A literal is either a positive literal +A or a negative literal -A, where A is an atomic formula. An atomic formula is written as

$$P(t_1,\ldots,t_m), \qquad m = 0,1,\ldots,$$

where P is a predicate symbol and $t_1,\dots,t_m$ are terms. A term is either a variable or an expression $f(t_1,\dots,t_k)$, $k=0,1,\dots$, where f is a k-place function symbol and the t_i are terms. A constant is a 0-place function symbol. For the sets of predicate symbols, function symbols, and variables one is free to choose any three mutually disjoint sets of symbols.

In the examples of this paper, a predicate symbol is a word starting with an upper-case letter; a variable is a word starting with a lower-case letter; a constant symbol is an underlined word.

4.2 A semantics for resolution logic (see [11])

The semantics of a language determines the meaning of a sentence of the language: it deals with the relationship between a sentence of the language and a universe: a set of objects endowed with a structure which is partially determined by the structure of the language. An interpretation assigns a meaning to the variable-free terms constructible from the function symbols in the sentence and to the predicate symbols in the sentence. The remaining part of semantics determines whether the sentence is true in a given interpretation.

As a structured set of objects to serve as universe I consider here the so-called Herbrand universe: the set of all variable-free terms that can be constructed from the constants and other function symbols of the sentence.

The set of all atomic formulas $P(t_1,\dots,t_k)$ such that the predicate symbol P occurs in the sentence and such that $t_1,\dots,t_k$ are in its Herbrand universe, is called the Herbrand base $\hat{H}$ of the sentence.

Any subset I of $\hat{H}$ determines a Herbrand interpretation in the following way. A variable-free term denotes itself (note that the term occurs itself in the Herbrand universe). The meaning of an n-place predicate symbol P is the following n-ary relation over the Herbrand universe

$$\{(t_1,\dots,t_n) : P(t_1,\dots,t_n) \in I\}$$

The following rules determine whether the sentence is true in the interpretation determined by I. If this is the case, then I is said to be a model of the sentence. A sentence is said to be unsatisfiable if it has no model.

A variable-free literal +A is true iff $A \in I$.

A variable-free literal -A is true iff $A \notin I$.

A variable-free clause is true iff at least one of its literals is true.

A clause is true iff every one of its variable-free instances is true.

A sentence is true iff each of its clauses is true.

4.3 A pragmatics for resolution logic

Semantics determines meaning in a strict sense only involving a relationship between the language and a universe. The sense of meaning that also involves the user of the language, for instance in the form of explanations helping her to understand a sentence, belong to the pragmatics of the language.

The concepts of classical predicate logic are intuitively understood by many people; it is therefore useful to express the meaning of sentences of resolution logic in terms of classical logic, as has been done by Kowalski [21] in the following way.

The meaning of a sentence $\{ C_1,\ldots,C_n \}$ is the conjunction

C_1 and ... and C_n.

Suppose a clause contains variables $x_1,\ldots,x_k$ and suppose it is written such that no negative literal precedes a positive one. The meaning of the clause

$+B_1\ldots+B_m-A_1\ldots-A_n$

is a universally quantified implication:

for all $x_1,\ldots,x_K$

B_1 or ... or B_m is implied by A_1 and ... and A_n

It may be helpful to have a special reading for a clause when

$m = 0$ or $n = 0$:

If $n = 0$, read

for all $x_1,\ldots,x_k$, B_1 or...or B_m.

If $m = 0$, read

for no $x_1,\ldots,x_k$, A_1 and...and A_n,

or, equivalently, read

for all $x_1,\ldots,x_k$, not A_1 or...or not A_n.

If $n=0$ and $m = 0$ the clause is the null clause and it is to be read as a contradiction.

Example

Let CAT be a sentence of resolution logic:

```
CAT = {+Cat(nil,y,y)
      ,+Cat(cons(u,x),y,cons(u,z))-Cat(x,y,z)
      }
```

According to the above pragmatics the sentence

CAT ∪{-Cat(cons(a,nil),cons(b,nil),cons(a,cons(b,nil)))}...(4.1)

is to be read as

for all y, Cat(nil,y,y)

and for all u,x,y,z,Cat(cons(u,x),y,cons(u,z)) is implied by Cat(x,y,z)

and not Cat(cons(a,nil),cons(b,nil),cons(a,cons(b,nil)))

...(4.2)

The sentence CAT can be used to define the catenation relation among linear lists.

Let me first explain how to recognize linear lists here. A linear list I define to be either the empty list or a nonempty list of which the 'head' is an atom and of which the 'tail' is a linear list (possibly empty). Now consider the Herbrand universe of (4.1). Think of the constant nil as the empty list and think of the contants a and b as atoms. Think of a term cons (α_1,α_2) in the HErbrand universe as the linear list of which the head is α_1 and of which the tail is α_2, provided **that** α_1 is an atom and α_2 is a linear list (possibly nil).

In this way the term

cons(a,cons(b,nil))

is to be thought of as the linear list containing the atoms a and b in that order. This is an awfully cumbersome notation. For the benefit of this and all further examples in the paper I will streamline the notation as follows. Write '.' instead of 'cons', write it in infix notation instead of in prefix notation and adopt the convention that, in the absence of brackets, pairs associate from the right to the left. That is, instead of cons (a,cons(b,nil)), I write a.b.nil and this means a.(b.nil) rather than (a.b).nil. This notational simplification comes from PROLOG. Using it, the sentence (4.1) becomes

```
{+Cat(nil,y,y)
,+Cat(u.x,y,u.z) -Cat(x,y,z)                          ...(4.3)
,-Cat(a.nil,b.nil,a.b.nil)
}
```

In order to appreciate the sentence CAT as a way to define the catenation relation among linear lists, note that <u>a</u>.<u>b</u>.<u>nil</u> is the result of catenating <u>a</u>.<u>nil</u> and <u>b</u>.<u>nil</u> in that order. The sentence (4.3) is unsatisfiable because it denies this fact. This suggests that we can define in general the catenation relation as the set of triples α_1, α_2, α_3 of linear lists such that

$$\text{CAT} \cup \{-\text{Cat}(\alpha_1, \alpha_2, \alpha_3)\}$$

is unsatisfiable.

4.4 <u>The problem of sorting stated in resolution logic</u>

```
SORT = {+Sort(x,y) -Perm(x,y) -Ord(y)
       ,+Perm(nil,nil)
       ,+Perm(x.y,z)  -Cat(v1,x.v2,z) -Cat(v1,v2,v) -Perm(y,v)
       ,+Ord(nil),+Ord(x.nil)
       ,+Ord(x.y.z) -Inf(x,y) -Ord(y.z)
       }
```

The sentence SORT will be used to define the sortedness relation among linear lists. The clauses of SORT are intended to be read as follows:

1) y is the sorted version of x if y is a permutation of x and if y is ordered
2) <u>nil</u> is a permutation of <u>nil</u>
3) z is a permutation of x.y if v is the result of deleting x from z (see Cat as defined by CAT) and if v is a permutation of y
4) the empty list is ordered
5) any one-element list is ordered
6) a list of at least two atoms is ordered if the first two are in order (see Inf as defined below) and if the tail of that list is ordered

The basic ordering relation between atoms, as expressed in Inf, still has to be defined. Such a definition will have to depend on which atoms there are. For instance, if atoms are letters of the alphabet then the definition can be a sentence containing a clause for each pair of letters in the ordering relation:

$$
\begin{array}{llll}
\mathrm{INF} = \{ & +\mathrm{Inf}(\underline{a},\underline{a}),\ +\mathrm{Inf}(\underline{b},\underline{b}),\ +\mathrm{Inf}(\underline{c},\underline{c}),\ \ldots,\ +\mathrm{Inf}(\underline{z},\underline{z}) \\
 & ,+\mathrm{Inf}(\underline{a},\underline{b}),\ +\mathrm{Inf}(\underline{b},\underline{c}),\ \ldots \\
 & ,+\mathrm{Inf}(\underline{a},\underline{c}),\ \ldots \\
 & ,\ \ldots \qquad\qquad\qquad \ldots \\
 & \qquad\qquad \ldots \qquad ,\ +\mathrm{Inf}(\underline{c},\underline{z}) \\
 & \quad \ldots \qquad ,\ +\mathrm{Inf}(\underline{b},\underline{z}) \\
 & ,+\mathrm{Inf}(\underline{a},\underline{z}) \\
 & \}
\end{array}
$$

A part of the data contained in INF can be deduced when the properties of transitivity and reflexitivity are given as in

$$
\begin{array}{ll}
\mathrm{INF}\ 1 = \{ & +\mathrm{Inf}(x,x)\ -\mathrm{Letter}(x) \\
 & ,+\mathrm{Inf}(x,y)\ -P(x,y) \\
 & ,+P(\underline{a},\underline{b}),\ +P(\underline{b},\underline{c}),\ldots,+P(\underline{y},\underline{z}) \\
 & ,+P(x,z)\ -P(x,y)\ -P(y,z) \\
 & \qquad\qquad -\mathrm{Letter}(x)\ -\mathrm{Letter}(y)\ -\mathrm{Letter}(z) \\
 & ,+\mathrm{Letter}(\underline{a}),\ldots,+\mathrm{Letter}(\underline{z}) \\
 & \}
\end{array}
$$

INF 1 has fewer clauses but a proof precedure will in general have to go through several steps in order to prove what is required. In INF anything that can be deduced from it is explicitly present. INF 1 has the disadvantage that a long deduction may often be necessary, for instance, if $\underline{a}$

and $\underline{z}$ have to be compared often. It may be useful in such a case to add redundantly to INF1 the ready-made fact + Inf $(\underline{a},\underline{z})$.

Such an arrangement would be reminiscent of Michie's 'memo-functions' [27]. Those familiar with this mechanism will see that it can be used to produce hybrids of INF and INF1 that adapt advantageously to a given pattern of usage.

It should be noted that in PROLOG there is no need for the programmer to give a definition of Inf at all: the system calls instead a brief FORTRAN subroutine for the comparison of characters or numbers. Such 'built-in' predicates exist for several other generally useful functions.

The sentence A = SORT∪CAT∪INF defines the relation of sortedness among linear lists in the following way:

$A \cup \{-\mathrm{Sort}(\alpha_1,\alpha_2)\}$ is unsatisfiable if α_1 and α_2 are linear lists and if α_2 is the sorted version of α_1. Moreover,

$$A \cup \{-\mathrm{Sort}(\alpha_1,y)\} \qquad \ldots (4.4)$$

is unsatisfiable for any linear list α_1 (made up of atoms accounted for in INF). Later we shall see that the sentence (4.4) can cause a proof procedure to construct α_2, a sorted version of α_1. For the moment, when we have not yet discussed proof procedures, this construction is best described semantically.

Because (4.4) is unsatisfiable there exists, by a theorem of Hebrand, an unsatisfiable finite set of variable-free instances of clauses in this sentence. All that need be said now about the proof procedure is that it will construct such a set and that this set contains only one instance of -Sort (α_1,y). The substitution for y in that instance is α_2, a sorted version of α_1.

This is in fact the realization in resolution logic of Green's simulation method of automatic programming (see figure 3.1). In the simulation method one can make a computer sort lists without anyone ever having to write a program to do it: we only need a program for a complete resolution proof procedure and the specification A.

Is this an interesting alternative to the writing of a sorting program? I believe it is not. I know of no proof procedure that does any better with this specification than to generate a permutation of α_1 until it is found to violate order, then to generate the next, and so on until an ordered permutation is encountered.

Although it will be universally agreed that this application of simulation is a computational disaster, not everybody will agree on the cause. Most work in automatic resolution theorem-proving prior to about 1970 seems to be based on the assumption that the causes of such a disappointing result can be cured by changes to the proof-procedure. The subsequent lack of success caused most workers in automatic problem-solving to discard resolution logic altogether, thus throwing away the baby with the bath water. The more immediately successful _ad hoc_ methods advocated by Minsky and Papert (see their 'Uniform Proof Procedures versus Heuristic Knowledge' in [30]) carried the day.

In the problem of sorting, however, there is no need to take recourse to such methods: the cause of the disappointing result is in the specification. A good proof procedure is a necessary, not a sufficient, condition for computational success.

There are often different ways to specify the same relation, and there are among equivalent, correct specifications computationally good ones and computationally bad ones (with respect to a given type of proof procedure), just as there are among equivalent, correct programs efficient ones and inefficient ones (with respect to a given machine).

I will exhibit a specification of the sortedness relation for which the simulation method does give efficient computations. The specification will turn out to have a very direct relationship to the 'quicksort' algorithm [17], In order to make the discovery of the specification as understandable as possible and also to introduce the idea of resolution logic as a program language, I will devote the next section to a transformation of an Algol program for quicksort into a specification in resolution logic for sortedness giving efficient computations according to the simulation method with a generally useful proof procedure.

4.5 A specification according to the quicksort principle

In section 2 I have contrasted descriptive against imperative specifications of algorithms. Descriptive specifications are mainly concerned with what an algorithm does and they are characteristic of high-level program languages. Imperative specifications are more concerned with how an algorithm does whatever it does. Such specifications are characteristic of low-level program languages. A language like Algol 60 is in principle completely imperative: every statement is a command to be executed. Yet sometimes such a program has descriptive value as well as an imperative effect: a well-written program can often be understood by a human reader without explicit comments as to what is to be done by the algorithm.

I start out with an Algol program for the quicksort program written as descriptively as I can. I then eliminate some of the remaining imperative features by writing successive versions in hypothetical generalizations of Algol 60. The end result reads (almost) like a specification in logic of the sortedness relation, which I will then also give. In the next section we will study this specification as a program in resolution logic. This interpretation will be facilitated by the Algol programs of this section.

In the original formulation of the quicksort algorithm [17] the sequence to be sorted is represented as an array. This makes an efficient program possible. Here it is preferable to disregard questions of efficiency at this level and to choose a data representation that requires a minimal distraction from the algorithm; hence the choice of lists as representation for the sequences to be sorted and for the results of sorting. However, in Algol 60 lists do not exist and I start therefore with a hypothetical extension, 'AlgolX', of Algol 60 that has as data types atom and list, a constant nil and standard one-place functions 'head' and 'tail' applicable to a non-empty list with the usual result, and the two-place function'.' in infix notation, applicable to an atom and a list giving the usual list as result.

```
procedure Sort(x,y);
begin if x≠nil
      then begin atom x1; list x2,u1,u2,v1,v2
           ;    x1:=head(x); x2:=tail(x)
           ;    Part(x1,x2,u1,u2)
           ;    Sort(u1,v1); Sort(u2,v2)
           ;    Cat(v1,x1.v2,y)
           end
;     if x=nil then y:=nil
end
```

figure 4.1 : Quicksort in AlgolX

In figure 4.1 the main components are calls to procedures without side effects, passing their results by means of output parameters. A non-empty list x is sorted by decomposing it into the first atom x1 (its head) and the remaining list x2 (its tail), possibly empty. 'Part' partitions x2 into two lists u1 and u2. The union of the sets of their elements is the set of the elements of x2; u1 contains all (if any) of the elements <x1, u2 contains all (if any) of the elements >x1. Of the elements =x1 it does not matter in which of u1 or u2 they are contained, as long as it is in exactly one of these. The lists u1 and u2 are sorted by recursive calls to 'Sort', giving v1 and v2. The sorted version y of x is obtained by catenating (by a call to 'Cat') v1 and v2 with x1 in between.

The identity x=head(x).tail(x) for any non-empty list x suggests that the selectors 'head' and 'tail' are superfluous in the presence of the constructor'.'. Consider for instance the declaration of 'Sort' as given in figure 4.2.

```
procedure Sort(x,y);
begin atom x1; list x2,u1,u2,v1,v2
;      if x=x1.x2
       then begin Part(x1,x2,u1,u2)
            ;     Sort(u1,v1); Sort(u2,v2)
            ;     Cat(v1,x1.v2,y)
            end
;      if x=nil then y:=nil
;
end
```

figure 4.2: Quicksort in AlgolY
(AlgolX plus pattern matching)

For the program in figure 4.2 the usual Algol interpreter will give an error message when attempting to evaluate the Boolean expression x=x1.x2 because the variables x1 and x2 are without value. Suppose, however, that the interpreter would not be neutral with respect to truth and falsity but would prefer true expressions to false ones. Then it might notice that even though the expression is not actually true, it can be made true without violating any existing values: the as yet undefined variables x1 and x2 can be assigned the values head(x) and tail(x) respectively and then the Boolean expression is true. We then say that x has been 'matched' to x1.x2.

But this can only be done if x≠nil. So why not combine the explicit assignation and the test x≠nil into one matching operation? The advantage is that in this way a low-level, machine-oriented operation like assignation is avoided. Note that matching is a generalization of equality in Algol 60 because when there are no undefined variables, matching reduces to the usual evaluation of Boolean equalities in Algol 60. Matching is known as a feature of several program languages for AI research [6].

Note that in the program in figure 4.2 the input parameter x seems to be quite unnecessary for communicating the algorithm. If we want to know whether the actual parameter for x is of the form x1.x2, why not have that form itself as a formal parameter? This suggests having two procedure declarations for 'Sort', applicable according to whether the input list is empty. Another extension of Algol is required, say AlgolZ. The proposal is to extend AlgolY to allow more than one procedure call with the same name. A declaration responds to a call when the actual parameters of the call match the formal parameters of the declaration. If the match is successful, the name is replaced by the body where undefined variables have received the values necessitated by the match. If more than one declaration is applicable, it is not determined which is actually applied: the program is indeterminate. This proposal follows the PLANNER language for AI research where the feature is called "procedure invocation by pattern matching".

```
procedure Sort(x1.x2,y);
begin list u1,u2,v1,v2
;     Part(x1,x2,u1,u2)
;     Sort(u1,v1); Sort(u2,v2)
;     Cat(v1,x1.v2,y)
end
; procedure Sort(nil,nil);
;
```

figure 4.3: Quicksort in AlgolZ

I stated that the program in figure 4.1 was as descriptive as I could make it in AlgolX. This is not quite true: the program requires Sort(u1,v1) to be executed before Sort(u2,v2). This is an example of what E.W. Dijkstra calls "sequential overspecification" because in fact the order is irrelevant and should therefore remain unspecified. All three programs in figures 4.1, 4.2, and 4.3 suffer from this defect, which is remedied in figure 4.4. Note the disappointing property of Algol that one sometimes has to write more when one wants to specify less.

```
Boolean procedure Sort(x1.x2,y);
begin list u1,u2,v1,v2; Boolean success
;      success:=Part(x1,x2,u1,u2)
;      success:=success∧Sort(u1,v1)∧Sort(u2,v2)
;      Sort:=success∧Cat(v1,x1.v2,y)
end
; Boolean procedure Sort(nil,nil); Sort:=true
; ...
```

figure 4.4: A better Quicksort in AlgolZ

In figure 4.4 I suppose that Part and Cat have become Boolean procedures as well; with the same effect on their parameters as before, but assuming the Boolean value true if their execution completes successfully.

Note that the effect of the body of Sort(x1.x2,y) in figure 4.4 is not necessarily the same as when it would have been

```
Sort:=
Part(x1,x2,u1,u2)∧Sort(u1,v1)∧Sort(u2,v2)∧Cat(v1,x1.v2,y)
```

This body does not specify any order between the calls, at least not according to the definition [31] of Algol 60. This body would be an example of sequential underspecification: I would have gone too far in eliminating imperative features. Note that a declaration with this body can be read as:

Sort(x1.x2,y) is true if
Part(x1,x2,u1,u2), Sort(u1,v1), Sort(u2,v2), and Cat(v1,x1.v2,y) are true

This suggests as specification of sorting in first-order predicate logic:

```
(∀x1,x2,y.
    (∃u1,u2,v1,v2.  Part(x1,x2,u1,u2)∧Sort(u1,v1)∧Sort(u2,v2)
                    ∧Cat(v1,x1.v2,y)
     ) ⊃ Sort(x1.x2,y)
)∧Sort(nil,nil)
```

The same sentence in resolution logic, together with specifications of Part and Cat can now be given as follows:

```
SORT1 = {+Sort(x1.x2,y) -Part(x1,x2,u1,u2)
                        -Sort(u1,v1) -Sort(u2,v2)
                        -Cat(v1,x1.v2,y)
        ,+Sort(nil,nil)
        ,+Part(x1,z.x2,z.u1,u2) -Inf(z,x1) -Part(x1,x2,u1,u2)
        ,+Part(x1,z.x2,u1,z.u2) -Inf(x1,z) -Part(x1,x2,u1,u2)
        ,+Part(x,nil,nil,nil)
        ,+Cat(u.x,y,u.z) -Cat(x,y,z)
        ,+Cat(nil,y,y)
        }
```

In subsection 4.4 I exhibited a sentence SORT as specification for a sorting algorithm giving a very disappointing behaviour when used for automatic programming in Green's simulation mode. I claimed that the cause was in the specification rather than inherent in the method. SORT1 is a specification of the sortedness relation in the same sense:

$$\text{INF} \cup \text{SORT1} \cup \{-\text{Sort}(\alpha_1,y)\}$$

is unsatisfiable for any linear list α_1 made up of atoms for which the order is specified in INF. A complete resolution procedure will construct as substitution for y the sorted version of α_1 in the course of proving unsatisfiability. In the next section I will discuss Kowalski's proof procedure that will perform the construction with the same efficiency as an AlgolZ interpreter would execute the program in figure 4.3.

5. Kowalski's procedural interpretation of resolution logic

5.1 The LUSH system of inference rule and proof procedure

A clause containing one positive literal is called a regular clause. A sentence containing only regular clauses is called a regular sentence. A clause containing no positive literal is called a goal statement. A goal statement containing no negative literal is called a halt statement and is written, as before, as □.

Up till now we have met with a syntax, a semantics, and a pragmatics for resolution logic. The procedural interpretation can be regarded as a supplement to the pragmatics given before. For specifications of relations we have learned the importance of unsatisfiability of a sentence. It is now time to consider a procedure for constructing a proof of unsatisfiability for an unsatisfiable sentence. For this we will only consider sentences of a special form: containing, apart from one goal statement, only regular clauses.

There are two interrelated reasons for dealing only with sentences with regular clauses and one goal statement. The first is that the LUSH rule of inference and proof procedure are designed for such sentences. The other reason is that for regular clauses and for goal statements there exists a pragmatics complementary to the one discussed in 4.3, namely the relevant part of Kowalski's procedural interpretation of resolution logic.

As can be learned from [23], the procedural interpretation is not restricted to regular clauses and goal statements.

The LUSH system is due to Kowalski [21,23]. In these papers the system has not received a name. The name LUSH was coined by Hill [16], the excuse being given as: '<u>L</u>inear resolution with <u>U</u>nrestricted <u>S</u>election for <u>H</u>orn clauses'. Clauses containing at most one literal are usually called Horn clauses honouring the poineering investigation [18] of A. Horn of some of their properties. I prefer to follow A. Colmerauer (unpublished work) and use the term regular clause, as defined above, to contain exactly one positive literal.

The LUSH rule of inference infers a new goal statement

$$(-A_1 \dots -A_{i-1} -B_1 \dots -B_m -A_{i+1} \dots -A_n)\theta$$

from a goal statement

$$-A_1 \dots -A_{i-1} -A_i -A_{i+1} \dots -A_n$$

with $-A_i$ as <u>selected literal</u> and a regular clause (if one exists)

$$+A \quad -B_1 \dots -B_m$$

that matches the goal statement in the sense that there exists a "most general" substitution θ of terms for variables (see [32]) that makes A_i and A identical. The LUSH rule of inference is a resolution with a goal statement and a regular clause as parents.

The proof procedure of LUSH determines how the rule of inference is used to construct a proof. One component of it is the <u>selection rule</u>; the other component is the <u>search strategy</u>. The rule of inference and the selection rule determine together with a sentence $S \cup \{G\}$ (S a regular sentence, G a goal statement) a tree of (which the nodes are) goal statements (also called the <u>search space</u>) in the following way. The root of the tree is G. The descendants of a node N in the tree are determined by first applying the selection rule to N, thus obtaining a literal L. There is just one descendant of N for each different way in which the rule of inference can be applied to a regular clause in S and the goal statement N with L as selected literal.

A path in the tree from the root to a halt statement represents a proof of the unsatisfiability of $S \cup \{G\}$. Suppose the path is the sequence of goal statements $G = G_o, G_1, \ldots, G_n = \square$. The rule of inference is such that, for $i = n-1, \ldots, 0$, if $S \cup \{G_{i+1}\}$ is unsatisfiable then so is $S \cup \{G_i\}$. $S \cup \{\square\}$ is unsatisfiable, therefore $S \cup \{G\}$ is.

For theoretical investigations an important property of LUSH is its completeness [16] which holds when S is regular and G is a goal statement: if $S \cup \{G\}$ is unsatisfiable, then the tree must contain a halt statement, whatever the selection rule. Completeness in this sense obviously does not mean that any path from the root ends in a halt statement. It does not preclude the possibility of infinite paths in the tree or of finite paths not ending in a halt statement. Completeness is sometimes asserted of a combination of search space and search strategy meaning that the search space always contains a halt statement and that the search strategy always finds a path to it.

Predicate-logic programming as discussed in this paper is restricted to application of LUSH to proving the unsatisfiability of a sentence $S \cup \{G\}$, S a regular sentence, G a goal statement. Neither Kowalski's procedural interpretation of predicate-logic nor the PROLOG system is thus restricted. Yet LUSH represents, as it were, the backbone of the PROLOG system. Because PROLOG has been shown to be a program language of considerable heuristic power, it is interesting to know what are the possible selection rules and what are the possible search strategies in that part of PROLOG that corresponds to LUSH.

The selection rule of PROLOG is the one that always selects the leftmost literal. Only those search spaces can therefore be implemented which can be obtained by ordering, once and for all in a given program, the negative literals of a clause. The search strategy is a depth-first, leftmost-descendant-first search of the tree determined by the selection rule just given. To further define the search strategy an ordering has to be given for the descendants. The ordering is determined by the ordering of the regular clauses in S: if clauses C_1 and C_2 occur in that order and both match the selected literal in a goal statement G then the descendant of G, obtained by matching with C_1, is generated first. Again, note that the search strategy is fixed once and for all by the order of the clauses in S.

5.2 The procedural interpretation

In order to use resolution logic for goal-directed computations another pragmatics is useful in addition to the one discussed before. According to Kowalski's procedural interpretation [21] a regular clause

$$+A\ -B_1\ \ldots\ -B_m \qquad\qquad m = 0,1,\ldots$$

is interpreted as a procedure definition. The positive literal +A is interpreted as the procedure name. The negative literals $-B_1,\ldots, -B_m$ are interpreted as procedure calls constituting the procedure body of the definition. The body may be empty (when m=0); such a procedure definition is interpreted as an assertion of fact.

A goal statement, that is a clause containing no positive literal, is interpreted as a set of procedure calls to be executed. When a goal statement is empty there are no more procedures to be executed; it is therefore called a halt statement.

The procedural interpetation is based on an analogy between the inference rule of LUSH and the computation rule for procedure-oriented program languages that executes a procedure call by replacing the call by the body of a declaration of which the name matches the call. This is just what happens in an application of the LUSH rule of inference as described in the previous section: $-A_i$ is the procedure call selected for execution from the goal statement, A is the name of a matching declaration, and the body replacing the call $-A_i$ is $-B_1,\ldots,-B_m$. The substitution θ modifies the body in a way that corresponds to replacing formal by actual parameters. The fact that the other procedure calls in the original goal statement may also be modified by θ as a result of executing $-A_i$ is an interesting generalization of the usual computation rule for procedure-oriented languages.

Suppose the selection rule of LUSH is the one that always selects the leftmost literal of the goal statement for execution. Then the goal statement acts like the stack of procedures called but not yet executed as used in a typical implementation of the computation rule for procedure-oriented languages. The leftmost literal corresponds to the top of the stack. Other selection rules correspond to departures from the stack-disciplined implementation, such as, for instance, the use of co-routines.

It may be verified that LUSH (with the selection rule that always selects the leftmost literal) acting on the sentence.

$$\text{SORT1} \cup \{-\text{Sort}\ (\alpha_1, y)\}$$

produces the sorted version of α_1 with approximately the same number of procedure calls as an Algol Z interpreter would require for sorting α_1 with the program in figure 4.3. This comparison assumes that Part and Cat would also be programmed in Algol Z using procedure calls only. For this to be achieved it is necessary that the tree of goal statements contains essentially only one path and, of course, that this path terminates and that it terminates in the halt statement. A selection rule that causes the tree to have this form is the one that always selects the leftmost literal with the literals ordered as in the listing of SORT.

Example.

The list $\underline{c}.\underline{a}.\underline{b}.\underline{nil}$ may be sorted by demonstrating the unsatisfiability of the sentence SORT1 $\cup$ INF $\cup$ $\{-\text{Sort}\ (\underline{c}.\underline{a}.\underline{b}.\underline{nil}, y)\}$. Given that the selection rule always selects the leftmost literal, an initial part of the search space is:

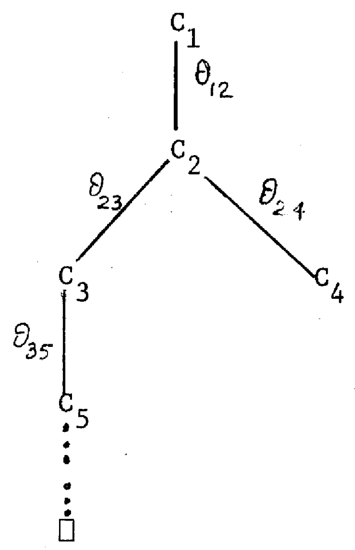


figure 5.1

In figure 5.1 there are the following goal statements:

$C_1 = \{-\text{Sort }(\underline{c}.\underline{a}.\underline{b}.\underline{\text{nil}},y)\}$

$C_2 = \{-\text{Part }(\underline{c},\underline{a}.\underline{b}.\underline{\text{nil}},p1,p2)\} \cup C$

$C_3 = \{-\text{Inf}(\underline{a},\underline{c})\ -\text{Part}(\underline{c},\underline{b}.\underline{\text{nil}},u1,u2)\} \cup C\ \theta_{23}$

$C_4 = \{-\text{Inf}(\underline{c},\underline{a})\ -\text{Part}(\underline{c},\underline{b}.\underline{\text{nil}},u1,u2)\} \cup C\ \theta_{24}$

$C_5 = \{-\text{Part}(\underline{c},\underline{b}.\underline{\text{nil}},u1,u2)\} \cup C\ \theta_{23}\ \ \theta_{35}$

where

$C = \{-\text{Sort}(p1,q1)\ -\text{Sort}(p2,q2)\ -\text{Cat}(q1,\underline{c}.q2,y)\}$

In figure 5.1 there are the follwoing substitutions:

$\theta_{12} = (x1,x2{:=}\underline{c},\underline{a}.\underline{b}.\underline{\text{nil}})$

$\theta_{23} = (x1,z,x2,p1,p2{:=}\underline{c},\underline{a},\underline{b}.\underline{\text{nil}},\underline{a}.u1,u2)$

$\theta_{24} = (x1,z,x2,p1,p2{:=}\underline{c},\underline{a},\underline{b}.\underline{\text{nil}},u1,\underline{a}.u2)$

$\theta_{35} = \text{identity}$

In figure 5.1 the tree of goal statements has more than one path. Wherever a branch occurs (when Part is the predicate of the selected literal), it has two arms, one of which ends at the next goal statement. Notice the important rôle played by the selection rule. Suppose that in the goal statement C_4 any literal would have been selected other than $-\text{Inf}(\underline{c},\underline{a})$, the leftmost. Then the goal statement would have had at least one descendant, and that one perhaps again, and so on, whereas the presence of $-\text{Inf}(\underline{c},\underline{a})$ makes it impossible for the halt statement to be an ultimate descendant. The selection $-\text{Inf}(\underline{c},\underline{a})$ ensures that the fact, that the wrong turn has been taken at the branch, is detected before time is wasted on other goals.

That this happens depends on the fact that the search space for $\text{INF} \cup \{-\text{Inf}(\underline{c},\underline{a})\}$ is finite. A terminating search reports the absence of a halt statement. The search space for $\text{INF1} \cup \{-\text{Inf}(\underline{c},\underline{a})\}$ is infinite and does not contain a halt statement either. A search for a halt statement will not terminate. Therefore a LUSH proof procedure selecting the leftmost literal and using a depth-first search strategy will not in general succeed in sorting by proving the unsatisfiability of

$$\text{SORT1} \cup \text{INF1} \cup \{-\text{Sort}(\alpha_1,y)\}$$

where α_1 is a linear list containg only atoms occurring in INF 1. This in spite of the fact that INF and INF 1 are equivalent in the sense that

$$\text{INF} \cup \{-\text{Inf}(\alpha_1,\alpha_2)\}$$

is unsatisfiable if and only if

$$\text{INF1} \cup \{-\text{Inf}(\alpha_1,\alpha_2)\}$$

is unsatisfiable, for any variable-free terms α_1 and α_2.

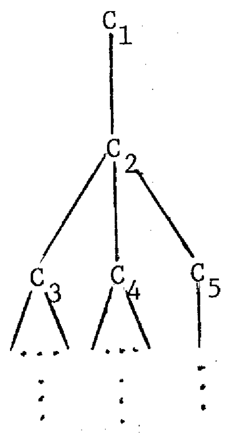
What happens at branches in the tree of figure 5.1 illustrates how "conditionals" can be handled in predicate-logic programming. Where an Algol-like language would have a conditional statement, a specification in logic will have two declarations with the same name, say +P. In general, with a selected literal -P, the goal statement has two descendants, one for each declaration with name +P. Sometimes the call matches only one name, as in the goal statement C_1 of figure 5.1: procedure call by pattern matching has carried out the test whether the input list is empty. Sometimes the call matches both names, as in the goal statement C_2. The test is then deferred one step because in the descendant goal statements the selected literals represent mutually disjoint contingencies. One of the descendants must remain without issue. When both descendants continue on to a halt statement, we have the analogue of an indeterminate algorithm.

The specification SORT 1 is not only a specification for quicksort, because, for instance, the sentence

$$\text{SORT 1} \cup \text{INF} \cup \{-\text{Sort}(x,\underline{a}.\underline{b}.\underline{c}.\underline{nil})\}$$

is unsatisfiable. LUSH is able to prove it is, and will substitute for x any of the lists $\underline{a}.\underline{b}.\underline{c}.\underline{nil}$,$\underline{a}.\underline{c}.\underline{b}.\underline{nil}$,$\underline{b}.\underline{a}.\underline{c}.\underline{nil}$,$\underline{b}.\underline{c}.\underline{a}.\underline{nil}$, $\underline{c}.\underline{a}.\underline{b}.\underline{nil}$, and $\underline{c}.\underline{b}.\underline{a}.\underline{nil}$ of which the sorted version is $\underline{a}.\underline{b}.\underline{c}.\underline{nil}$. The specification SORT 1 can apparently also be used as a permutation algorithm, provided that LUSH used a different selection rule.

Let us see what happens if the selection rule is not changed, that is, if the leftmost literal is always selected, which worked so well for sorting. The tree of goal statements would begin as shown in figure 5.2.



In figure 5.2 the clauses are

$C_1 = \{-\text{Sort}(x,\underline{a}.\underline{b}.\underline{c}.\underline{\text{nil}})\}$

$C_2 = \{-\text{Part}(x1,w2,p1,p2)\} \cup C$

$C_3 = \{-\text{Inf}(z,x1)\ -\text{Part}(x1,x2,u1,u2)\} \cup C\ \theta_{23}$

$C_4 = \{-\text{Inf}(x1,z)\ -\text{Part}(x1,x2,u1,u2)\} \cup C\ \theta_{24}$

$C_5 = \{-\text{Sort}(\underline{\text{nil}},v1)\ -\text{Sort}(\underline{\text{nil}},v2)\ -\text{Cat}(v1,x\ .v2,\underline{a}.\underline{b}.\underline{c}.\underline{\text{nil}})\}$

where

$C = \{-\text{Sort}(p1,v1)\ -\text{Sort}(p2,v2)\ -\text{Cat}(v1,x1.v2,\underline{a}.\underline{b}.\underline{c}.\underline{\text{nil}})\}$

In figure 5.2 the substitutions are

$\theta_{12} = (x,y{:=}x1.w2,\underline{a}.\underline{b}.\underline{c}.\underline{\text{nil}})$

$\theta_{23} = (w2,p1,\ p2{:=}z.x2,z.u1,u2)$

$\theta_{24} = (w2,p1,p2{:=}z.x2,u1,z.u2)$

$\theta_{25} = (x1,w2,p1,p2{:=}x,\underline{\text{nil}},\underline{\text{nil}},\underline{\text{nil}})$

We now have a very large search space. C_5 has one descendant clause only, but no halt statement as ultimate descendant. C_3 and C_4 have hundreds of immediate descendants each. Rather than to try and follow in detail the search space, let us try to get an overview of what happesn in it. It will be useful to extend the notion of "descendant" to literals, as follows. A literal L_2 in a clause C_2 is a descendant of a literal L_1 in a clause C_1 if C_2 is a descendant of C_1 and L_1 is the selected literal in C_1 and if L_2 is in the procedure body that replaces L_1 in C_1 to give C_2. Also if L_3 is a descendant of L_2 and if L_2 is a descendant of L_1 then L_3 is a descendant of L_1.

When the leftmost literal is always selected, descendants of -Part(x1,w2,p1,u2) will be selected as long as there are any. By the time there are no more such descendants, x1,w2,p1,u2 will have been substituted by any terms that happen to fit in the partition relation. The input list $\underline{a}.\underline{b}.\underline{c}.\underline{nil}$ has not been taken into account at all. We see that with a wrong selection rule LUSH behaves as "blindly" as has often been observed with earlier, faulty applications of resolution theorem-proving.

A good selection rule should select for execution a procedure call containing the input data, in this case

$$-\mathrm{Cat}(v1,x1.v2,\underline{a}.\underline{b}.\underline{c}.\underline{nil})$$

The outputs of this procedure call can act as input to the calls to Sort and after finishing those there are data for executing Part. With a selection rule that imposes this ordering on calls, the tree of goal statements is quite different: it may be verified that in that case all substantial branches lead to success; a different success each time: each path to a halt statement builds a substitution that gives the x in the original goal statement a different permutation as value. We have obtained an adequate permutation generator. We have just seen an example of the possibility in perdicate-logic programming, pointed out by Kowalski [21] , of computing different functions from the same relational specification by using different combinations of arguments for input and output.

The selection rule required to achieve successful use as permutationgenerator is again one that can easily be specified in PROLOG by ordering the literals in such a way that the PROLOG rule of selecting the leftmost literal executes the right one every time. A correct ordering is achieved by interchanging the literals

-Part(x1,x2,u1,u2) and -Cat(v1,x1.v2,y) in **SORT** 1. The interchange has been carried out in PERM, shown below, where also a clause has been added which causes the permutations to be printed. The predicate Print can be defined in such a way that in this case all permutations are printed rather than just the first one found.

As a result of having PROLOG execute the sentence

$$\mathrm{PERM} \cup \mathrm{INF} \cup \{-\mathrm{Perm}(\underline{a}.\underline{b}.\underline{c}.\underline{nil})\}$$

all permutations of the list a.b.c.nil are printed out.

```
PERM = {+Sort(x1.x2,y) -Cat(v1,x1.v2,y)
                       -Sort(u1,v1) -Sort(u2,v2)
                       -Part(x1,x2,u1,u2)
       ,+Sort(nil,nil)
       ,+Part(x1,z.x2,z.u1,u2) -Inf(z,x1) -Part(x1,x2,u1,u2)
       ,+Part(x1,z.x2,u1,z.u2) -Inf(x1,z) -Part(x1,x2,u1,u2)
       ,+Part(x,nil,nil,nil)
       ,+Cat(u.x,y,u.z) -Cat(x,y,z)
       ,+Cat(nil,y,y)
       ,+Perm(y) -Sort(x,y) -Print(x)
       }
```

If one would just be interested in obtaining a permutation generator, INF and the calls -Inf(...) would be deleted. They only restrict the program as a permutation generator by making it necessary to give the list in sorted order. But here I want to keep the connection with sorting as simple as possible.

6. Predicate-logic programs for syntactical analysis

6.1 Left-recursivity, search space, and search strategy

I will first explain the method of Colmerauer and Kowalski [8,23] for representing strings and grammars in predicate logic. Strings are represented by lists. Whenever it has to be asserted that a substring is adjacent to another, the substrings are represented by 'differences' of lists. Two lists x and y are defined to have a difference if there exists a list z such that x is the result of catenating z and y, in that order. The difference is the string of the atoms in z. For instance, a.b.c.nil and c.nil have a difference which is the string of the atoms a and b.

Let us consider a formal grammar $G = (V_N, V_T, P, S)$ where V_N is the set of nonterminals, V_T the set of terminals, P the set of productions and S the start symbol. Take for example a context--free left-recursive grammar for 'unsigned integers'

$$
\begin{aligned}
G = (&V_N=\{Ui,D\}\\
,&V_T=\{0,1,2,3,4,5,6,7,8,9\}\\
,&P=\{Ui\rightarrow D\\
&\quad ,Ui\rightarrow Ui\ D\\
&\quad ,D\rightarrow 0,D\rightarrow 1,D\rightarrow 2,D\rightarrow 3,D\rightarrow 4\\
&\quad ,D\rightarrow 5,D\rightarrow 6,D\rightarrow 7,D\rightarrow 8,D\rightarrow 9\\
&\quad \}\\
,&S = Ui\\
)&
\end{aligned}
$$

According to the method of Colmerauer and Kowalski the production rules are represented by the following sentence of resolution logic:

```
UIL = {+Ui(x,y) -D(x,y)
      ,+Ui(x,z) -Ui(x,y) -D(y,z)
      ,+D(0.y,y),+D(1.y,y),+D(2.y,y),+D(3.y,y),+D(4.y,y)
      ,+D(5.y,y),+D(6.y,y),+D(7.y,y),+D(8.y,y),+D(9.y,y)
      }
```

The formula +Ui(x,z) asserts that the substring represented by the difference of lists x and z is of the syntactic category Ui. The second clause reads: the difference between x and z is a Ui if there exists a y such that the difference between x and y is a Ui and the difference between y and z is a D.

The grammar G is left-recursive. It is interesting to see how this fact is reflected in the search space of LUSH selecting the leftmost literal for the sentence

UIL ∪ {-Ui(3.9.2.nil,nil)}

In showing this sentence to be unsatisfiable, LUSH recognizes in effect the difference between 3.9.2.nil and nil as an unsigned integer.

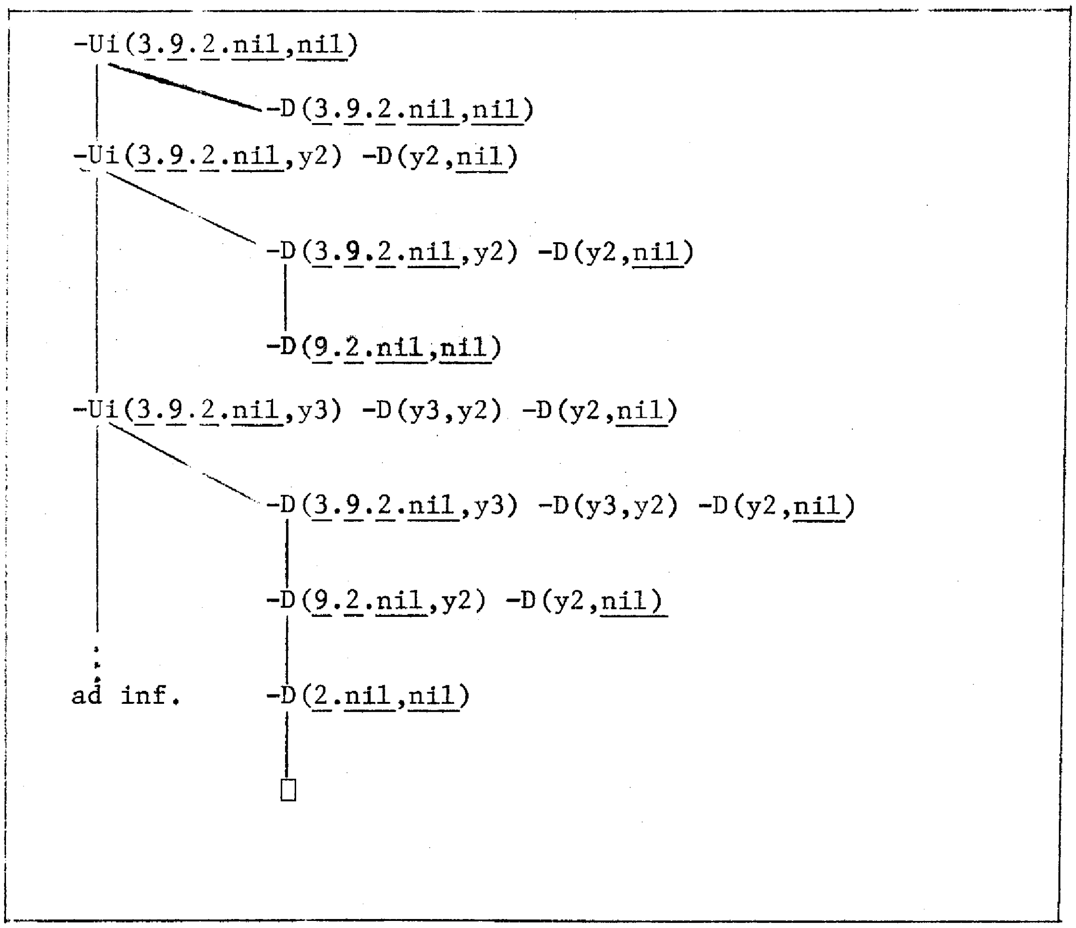

-Ui(3.9.2.nil,nil)
-D(3.9.2.nil,nil)
-Ui(3.9.2.nil,y2) -D(y2,nil)
-D(3.9.2.nil,y2) -D(y2,nil)
-D(9.2.nil,nil)
-Ui(3.9.2.nil,y3) -D(y3,y2) -D(y2,nil)
-D(3.9.2.nil,y3) -D(y3,y2) -D(y2,nil)
-D(9.2.nil,y2) -D(y2,nil)
ad inf.
-D(2.nil,nil)
□

figure 6.1:

Search space for a left-recursive grammar

The combination of left-recursivity and the rule selecting the leftmost literal for the LUSH rule of inference results in the search space of figure 6.1 which has an infinite branch. Whether LUSH will find the parse depends on the search strategy. For instance, PROLOG's depth-first search strategy will find the parse when the first two clauses are ordered as shown in UIL, but not when they would have been ordered in the other way.

The right-recursive version of the grammar UIL is represented by the following sentence:

```
UIR = {+Ui(x,y) -D(x,y)
      ,+Ui(x,z) -D(x,y) -Ui(y,z)
      ,+D(0.y,y),+D(1.y,y),+D(2.y,y),+D(3.y,y),+D(4.y,y)
      ,+D(5.y,y),+D(6.y,y),+D(7.y,y),+D(8.y,y),+D(9.y,y)
      }
```

In figure 6.2 we find the search space for LUSH selecting the leftmost literal for the sentence

$\mathrm{UIR} \cup \{-\mathrm{Ui}(3.9.2.\mathrm{nil},\mathrm{nil})\}$

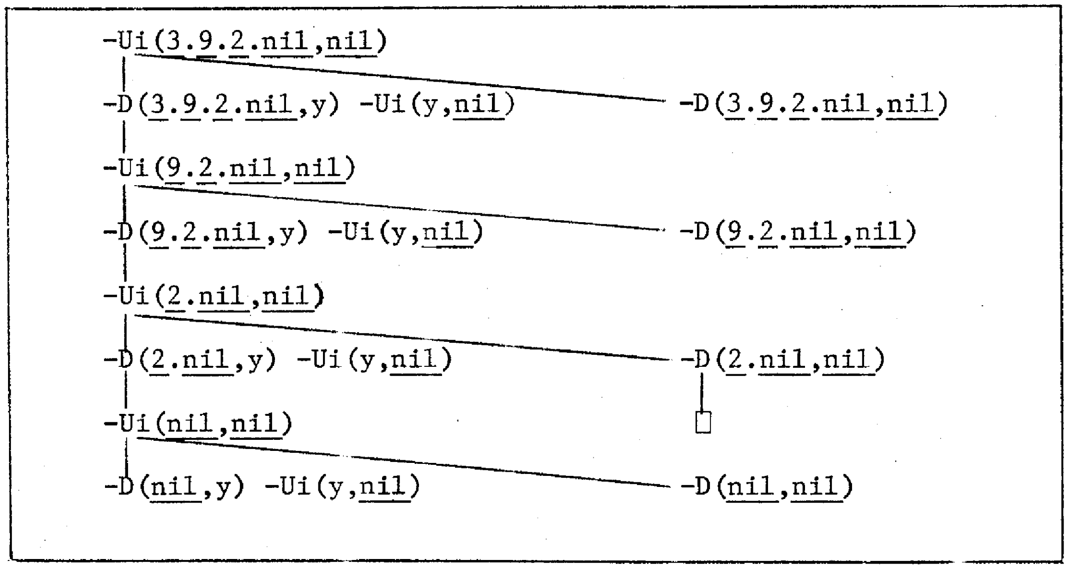


figure 6.2

Search space for a right-recursive grammar

Here the search space is finite: any search strategy will find the parse. In PROLOG one can handle the left-recursive grammar, but only by choosing the fight ordering of clauses. The right-recursive grammar is easier in the sense that ordering of clauses does not matter a great deal.

A. Colmerauer (unpublished work) has given a formal definition of formal grammars and of the parsing problem in resolution logic according to the method of Colmerauer and Kowalski and has proved that there exists a LUSH derivation if and only if a parse exists. The parse can be reconstructed from the successive resolutions performed during the parse. A 'safe' search strategy, like breadth-first search of the tree of goal statements is guaranteed to find a parse if one exists; this cannot be said in general of PROLOG.

6.2 The regular sentences of resolution logic as generalized grammars

For the syntactical analysis of natural language context-free grammars are not suitable. For example, the necessity to express agreement in number (singular or plural) between a noun phrase and the corresponding verb phrase gives rise to an unmanageable proliferation of productions. The problem is discussed by Winograd [38], who goes on to argue that context-sensitive grammars, although an improvement in this respect, are not satisfactory either. He prefers to define a language not by a grammar, but as the set of strings successfully analysed by a program in a program language named PROGRAMMAR.

I do not think one should leave it at this: besides having a program, it is also useful to have a machine-independent description of the language and an easy method to convince oneself of the correctness of the program, that is, that the program indeed performs according to the description. According to the method of Colmerauer and Kowalski one can describe in resolution logic a language in a machine-independent way. With respect to a given (suitable) proof procedure, the machine-independent description becomes a (useful) program.

I will illustrate some of the distinctive features of predicate-logic programming by an application to syntactic analysis and especially by comparing it to the PROGRAMMAR approach. In this section I will give descriptions in predicate-logic (that can be run as PROLOG programs) based on two of Winograd's example PROGRAMMAR programs, namely, his 'Grammar 2' and 'Grammar 3'.

In his first example, Winograd gives both an explicit machine-independent description as a context-free grammar ('Grammar 1') and the corresponding PROGRAMMAR program ('Grammar 2'). To obtain a PROLOG program, all we have to do is to transcribe Grammar 1 systematically according to the representation of Colmerauer and Kowalski for strings and productions. The transcription is in fact so systematic that PROLOG has a facility, 'SuperQ', that allows direct input of grammars in a notation similar to the one of Colmerauer's Q-systems [7]. But for just the two examples here it is not worth introducing a new notation. The remaining examples are therefore also in the language of resolution logic as defined in this paper.

The grammar G1 includes Winograd's Grammar 1:

$$
\begin{aligned}
G1 = (&V_N=\{Sent,Np,Vp,Det,Noun,Iverb,Tverb\}\\
&,V_T=\{the,giraffe, apple, dreams,eats\}\\
&,P =\{Sent\rightarrow Np\ Vp\\
&\qquad ,Np\rightarrow Det\ Noun\\
&\qquad ,Vp\rightarrow Iverb,Vp\rightarrow Tverb\ Np\\
&\qquad ,Det\rightarrow the, Noun\rightarrow giraffe,Noun\rightarrow apple\\
&\qquad ,Iverb\rightarrow dreams,Tverb\rightarrow dreams, Tverb\rightarrow eats\\
&\qquad \}\\
&,S = Sent\\
&)
\end{aligned}
$$

According to the method of Colmerauer and Kowalski the productions of this grammar are represented by the following sentence of of resolution logic:

$$
\begin{aligned}
G2 = \{&+Sent(x,y)\ -Np(x,u)\ -Vp(u,y)\\
&,+Np(x,y)\ -Det(x,u)\ -Noun(u,y)\\
&,+Vp(x,y)\ -Iverb(x,y)\\
&,+Vp(x,y)\ -Tverb(x,u)\ -Np(u,y)\\
&,+Det(\underline{the}.y,y),+Noun(\underline{giraffe}.y,y)\\
&,+Noun(\underline{apple}.y,y),+Iverb(\underline{dreams}.y,y)\\
&,+Tverb(\underline{dreams}.y,y),+Tverb(\underline{eats}.y,y)\\
&\}
\end{aligned}
$$

In order to analyze "the giraffe dreams" LUSH is set to work on the sentence

G2 ∪ {-Sent(the.giraffe.dreams.nil,nil)}

With the rule selecting leftmost literals we find the search space of figure 6.3.

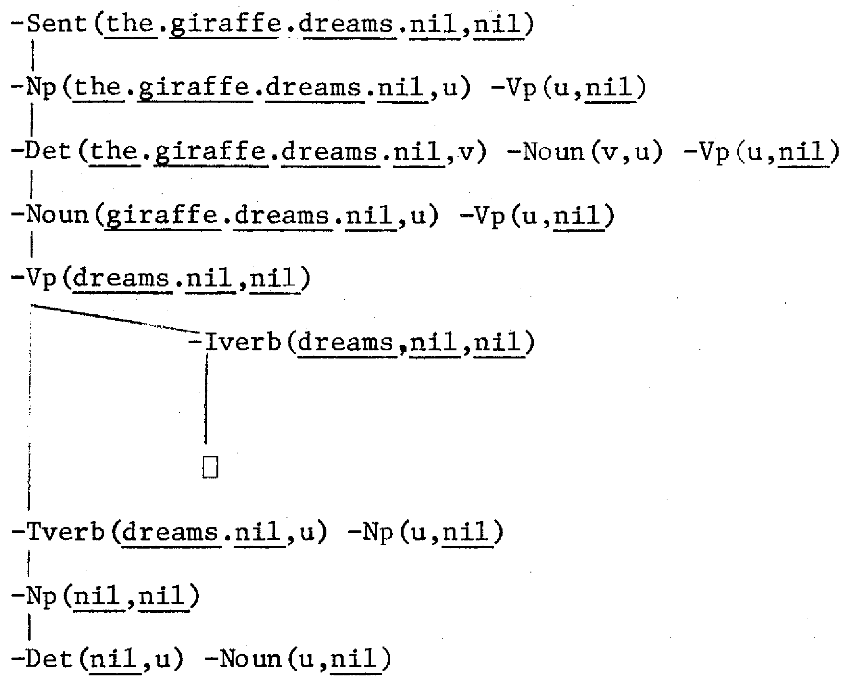

-Sent(the.giraffe.dreams.nil,nil)
-Np(the.giraffe.dreams.nil,u) -Vp(u,nil)
-Det(the.giraffe.dreams.nil,v) -Noun(v,u) -Vp(u,nil)
-Noun(giraffe.dreams.nil,u) -Vp(u,nil)
-Vp(dreams.nil,nil)
-Iverb(dreams,nil,nil)
□
-Tverb(dreams.nil,u) -Np(u,nil)
-Np(nil,nil)
-Det(nil,u) -Noun(u,nil)

figure 6.3:

The search space for 'the giraffe dreams'

The sentence G2 is a direct representation in predicate-logic of the facts expressed in G1. These facts are what has been called in section 3 the logic component of an algorithm for analysing sentences produced by G1. What has been called the control component determines, for instance, that a parser is top-down rather than bottom-up. When the control component is provided by a LUSH proof procedure, the resulting algorithm will be a top-down parser. In the PROGRAMMAR program 'Grammar 2' logic and control are not clearly separated.

The advantage of separating the logic and control components is that programs become more understandable, easier to get right, and to modify. Perhaps a useful way to make more precise the distinction between 'descriptive' versus 'imperative', or 'high-level' versus 'low-level' algorithm specification is to identify both distinctions with whether the logic and control components are separated.

When the logic and control components are separated, it can be recognized easily when two algorithms are the same except for the control component. For example, in the previous section we saw almost identical algorithms for quicksort and for a permutation generator differing only in their control components. For example, two parsers, one top-down and another bottom-up, for the same formal language should, according to the principle of separating logic from control, have a clearly recognizable and easily readable part in common, namely some machine-independent representation of a grammar for the language. This is achieved in predicate-logic programming, but not in PROGRAMMAR.

For parsers, the separation of logic and control is not new. It has been achieved in compiler-generators, which contain a program that takes a grammar as input to 'compile' it into a program that parses according to that grammar. LUSH takes both a grammar and a string (combined in one unsatisfiable sentence) as input to produce a parse, working in 'interpreter' rather than in 'compiler' mode. Another difference is that the program in a compiler-generator that produces a parser will not do anything else. PROLOG not only accepts predicate-logic programs specifying any context-free grammar (and certain nonrestrictive classes of type-0 and type-1 grammars as well, within the framework of method of Colmerauer and Kowalski) but also programs for anything else that fits the paradigm of predicate-logic programming. LUSH may be regarded as a generalized top-down parser and the regular sentences of resolution logic may be regarded as generalized context-free grammars which turn out to be convenient for programming a great variety of problems, like the sorting and permuting algorithms shown in this paper, or the problems arising in the large application programs mentioned earlier [3,8,9,19,33,37].

The sentence G2 can be interpreted as a logic component conveying no control information. In that case the selection rule and the search strategy will have to be stated explicitly when using LUSH. Or one can make use of the fact that one cannot avoid writing down an unordered set of literals or of clauses in some order. In PROLOG a LUSH proof procedure interprets the order of the literals as a selection rule and the order of the clauses as a particular choice of depth-first search strategy. PROLOG makes it possible to provide explicit control information by means of two system-defined predicates. It may be surprising to find how useful such limited possibilities for supplying control information can be.

As far as I can see, there is nothing in resolution logic that makes it more difficult to supply control information than in, say, PLANNER-like languages. There is no need to supply this information in logic or to camouflage it as logic, as is done in PROLOG. A more elaborate control language, as used in MICROPLANNER or CONNIVER, could be applied to direct LUSH, or other proof-procedures, which can combine top-down and bottom-up deductions, or those which, like the connection-graph theorem-prover [22], transcend the distinction between top-down and bottom-up.

Let us now continue with Winograd's examples and see how context-sensitive aspects can be introduced in his Grammar 1. I will use the language of the regular clauses of resolution logic as a generalized grammar of great expressive power, rather than try to fit into the formal grammars of the Chomsky hierarchy. The advantage of separating logic from control will become much more apparent here, because in PROGRAMMAR the context-sensitive aspects are handled by explicit instructions for moving a pointer about in the parse tree as it exists at the moment of execution of the instruction. In order to be able to understand Winograd's Grammar 3 in PROGRAMMAR one has to have a mental picture of the parse tree, and the position of the pointer, as it changes during execution.

It is therefore not easy to discover what the grammar is in Grammar 3. Following Winograd's comments, I assume it is Grammar 1 with the following elaborations:

1) the number of a noun phrase agrees with that of the corresponding verb phrase
2) the number of a noun phrase need not be determined by the noun only (this fish - these fish) and not by the 'th-word' only (the giraffe - the giraffes), but number is a feature of the entire noun phrase
3) a noun phrase need not have a determiner (giraffes dream), presumably provided that the noun phrase is plural

```
G3 = {+Sent(x,y) -Np(n,x,u) -Vp(n,u,y)
     ,+Np(n,x,y) -Th(n,x,u) -Noun(n,u,y)
     ,+Np(pl,x,y) -Noun(pl,x,y)
     ,+Vp(n,x,y) -Verb(i,n,x,y)
     ,+Vp(n,x,y) -Verb(t,n,x,u) -Np(m,u,y)
     }
INTERFACE = {+Noun(sg,u.y,y) -Npr(u,v)
            ,+Noun(pl,v.y,y) -Npr(u,v)
            ,+Verb(tty,sg,u.y,y) -Conj(tty,u,v)
            ,+Verb(tty,pl,v.y,y) -Conj(tty,u,v)
            }
LEXICON = {+Npr(giraffe,giraffes),+Npr(fish,fish),
          ,+Npr(dream,dreams)
          ,+Conj(tty,dreams,dream),+Conj(tty,eats,eat)
          ,+Th(n,the.y,y)
          ,+Th(sg,this.y,y),+Th(pl,these.y,y)
          }
```

Differences between G2 and G3 are the following. The predicates have received extra parameters for transmitting information about number (variables n and m, taking as possible values the constants *sg* for singular or *pl* for plural) or about whether or not a verb is transitive (variabl~~y~~e tty, taking as possible values the constants *i* or *t*). The last two parameters of literals in G3 are, as before, lists indicating where the substring under consideration begins and ends (variables u,x,and y). Lexical information has been removed from G3.

Lexical information has been placed in a separate sentence, LEXICON, giving pairs (Npr) with the singular and plural form of some nouns. At a further level of refinement the description will have to analyse the structure of the words themselves, for instance, to be able to say that, apart from exceptions, the plural is formed by appending an s. As it stands, the words in LEXICON are constants and have therefore no internal structure. Similarly, verbs are given by simply listing (with Conj) various conjugations; in the example these are restricted to the singular and plural of the third person, present, indicative. Finally, singular and plural forms of the 'th-words' <u>the</u> and <u>this</u> are listed. Many, quite different, *ad hoc* or more systematic, representations of the lexicon are usable with the same grammar G3, if a suitable interface is provided.

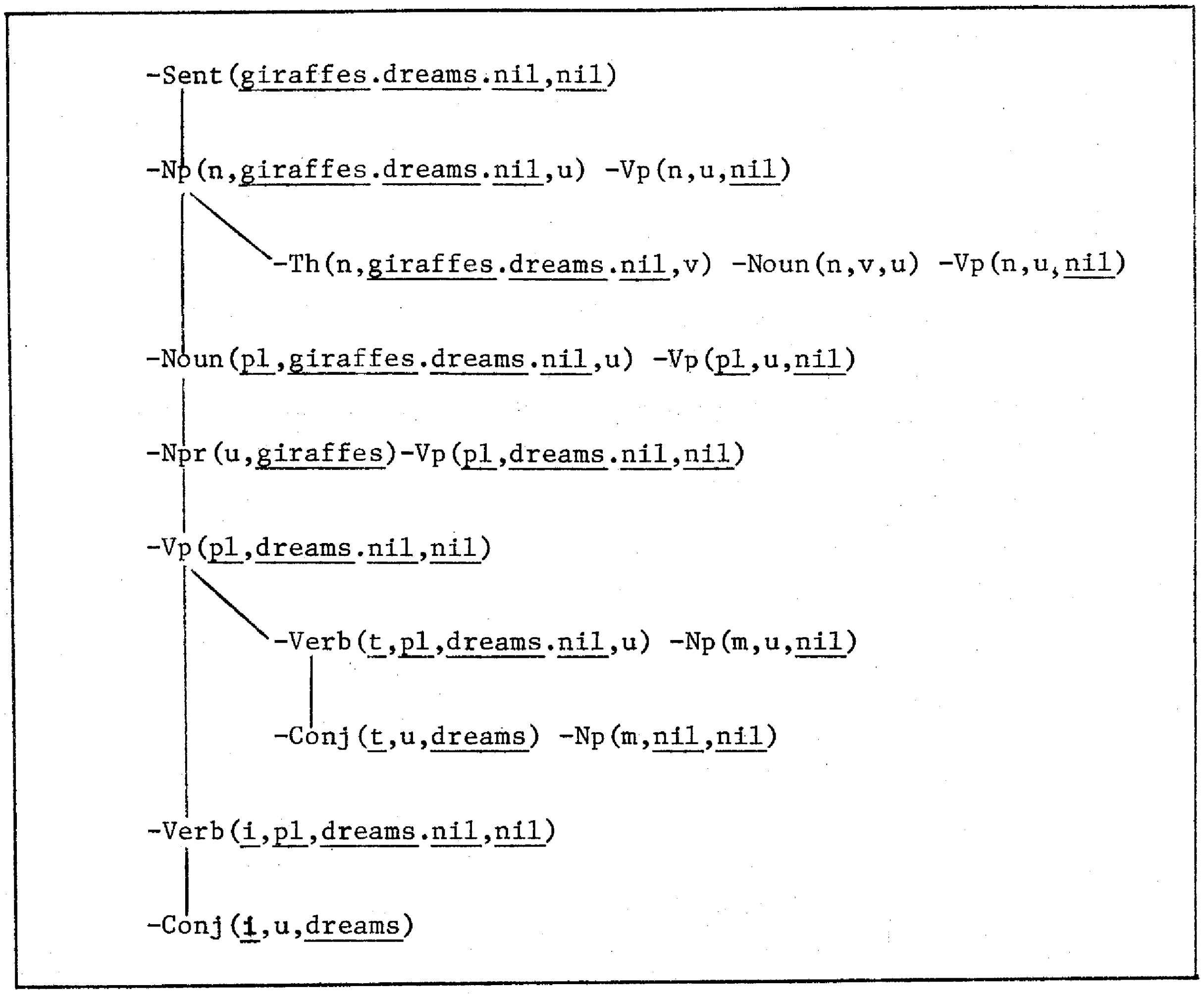
As the final example, I show the search space for LUSH with the rule selecting the leftmost literal for the unsolvable parsing problem represented by the satisfiable sentence

$$\mathrm{G3} \cup \mathrm{INTERFACE} \cup \mathrm{LEXICON} \cup \{-\mathrm{Sent}(\underline{\mathrm{giraffes}}.\underline{\mathrm{dreams}}.\underline{\mathrm{nil}},\underline{\mathrm{nil}})\}$$

The search space is finite; LUSH will terminate, whatever the search strategy, without having found a halt statement.

7 Acknowledgements

This paper grew out of an earlier attempt ('First-order predicate logic as a high-level program language', Report MIP-R-106, Dept. of Machine Intelligence, University of Edinburgh, 1974) at a tutorial exposition of programming with resolution logic. The discussions with Robert Kowalski and the encouragement of Donald Michie have helped me very much with the writing of MIP-R-106, which was supported by the U.K. Science Research Council. Additional support was provided by a research grant from the University of Waterloo.



```
-Sent(giraffes.dreams.nil,nil)
 |
-Np(n,giraffes.dreams.nil,u) -Vp(n,u,nil)
 |  \
 |   -Th(n,giraffes.dreams.nil,v) -Noun(n,v,u) -Vp(n,u,nil)
 |
-Noun(pl,giraffes.dreams.nil,u) -Vp(pl,u,nil)
 |
-Npr(u,giraffes)-Vp(pl,dreams.nil,nil)
 |
-Vp(pl,dreams.nil,nil)
 |  \
 |   -Verb(t,pl,dreams.nil,u) -Np(m,u,nil)
 |     |
 |   -Conj(t,u,dreams) -Np(m,nil,nil)
 |
-Verb(i,pl,dreams.nil,nil)
 |
-Conj(i,u,dreams)
```

figure 6.4:

The search space for "giraffes dreams"

8 References to the literature

1. Annual review in automatic programming. Pergamon Press, 1(1960-6(1970).
2. G. Battani and H. Meloni: Interpreteur du langage de programmation PROLOG. Groupe d'Intellignece Artificielle, U.E.R. de Luminy, Marseille, 1973.
3. M. Bergman and H. Kanoui: Application of mechanical theorem proving to symbolic calculus. Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Marseille, 1973.
4. W. W. Bledsoe: Splitting and reduction heuristics in automatic theorem-proving. Artificial Intelligence 2 (1971), 55-77.
5. W. W. Bledsoe, R. Boyer, W. H. Henneman: Computer proofs of limit theorems. Artificial Intelligence 3 (1972), 27-60.
6. D. Bobrow and B. Raphael: New programming languages for AI research. Computing Surveys 6 (1974), 153-174.
7. A. Colmerauer: Les systèmes-Q, ou un formalisme pour analyser et synthetiser des phrases sur ordinateur. Publication interne no 43, Dépt. d'Informatique, Faculté des Sciences, Université de Montréal.
8. A. Colmerauer: Programmation en langue naturelle. Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Marseille, 1974.
9. A. Colmerauer, H. Kanoui, R. Pasèro, and P. Roussel: Un système de communication homme-machine en français. Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Marseille, 1972.
10. E. W. Elcock: Descriptions. Machine Intelligence 3, B. Meltzer and D. Michie (eds), Edinburgh University Press, 1968. pp. 173-179.
11. M. H. van Emden and R. A. Kowalski: The semantics of predicate-logic as a programming language. Report MIP-R-103, Dept. of Machine Intelligence, University of Edinburgh, 1974.
12. J. M. Foster: Assertions: programs written without specifying unnecessary order. Machine Intelligence 3, B. Meltzer and D. Michie (eds), Edinburgh University Press, 1968, pp 387-391.

References (continued)

13. J. M. Foster and E. W. Elcock: Absys 1: an incremental compiler for assertions. Machine Intelligence 4, B. Meltzer and D. Michie (eds), Edinburgh University Press, 1969, pp 423-429.

14. C. Green: The application of theorem-proving to question-answering systems. Technical note 8, Artificial Intelligence Group, Stanford Research Institute, 1969.

15. P. J. Hayes: Computation and deduction. Proc. 1973 MFCS Conference, Czechoslovakian Academy of Sciences.

16. R. Hill: LUSH resolution and its completeness. DCL Memo 78 (1974), Dept. of Artificial Intelligence, University of Edinburgh.

17. C. A. R. Hoare: Algorithm 64: quicksort. Comm. ACM, 4 (1961), 321-321.

18. A. Horn: On sentences which are true of direct unions of algebras. J. Symbolic Logic, 16 (1951), 14-21.

19. H. Kanoui: Application de la demonstration automatique aux manipulations algébriques et à l'integration formelle sur ordinateur. Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Marseille, 1973.

20. S. C. Kleene: Mathematical Logic. Wiley, 1967.

21. R. A. Kowalski: Predicate logic as programming language. Proc. IFIP74, 569-574, North Holland, 1974.

22. R. A. Kowalski: A proof procedure using connection graphs. DCL Memo 74, University of Edinburgh, 1974. (To appear in the J. ACM.)

23. R. A. Kowalski: Logic for problem-solving. DCL Memo 75, Dept. of Artificial Intelligence, University of Edinburgh, 1974.

24. R. A. Kowalski: Inaugural Lecture, Imperial College, London, May 2, 1975.

25. R. A. Kowalski and D. Kuehner: Linear resolution with selection function. Artificial Intelligence 2 (1971), 227-260.

26. D. W. Loveland: A simplified format for the model-elimination theorem-proving procedure. J. ACM, 16 (1969), 349-363.

References (continued)

27. D. Michie: Memo functions and machine learning. Nature 218 (1968), 19-22.

28. D. Michie, R. Ross, G. J. Shannan: G-deduction. Machine Intelligence 7, B. Meltzer and D. Michie (eds), Edinburgh University Press/Wiley, 1972.

29. D. Michie and E. Sibert: Some binary derivation systems. J. ACM 21 (1974), 175-190.

30. M. Minsky and S. Papert: Progress Report. Memo 252, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1972.

31. P. Naur (ed): Revised Report on the Algorithmic Language Algol60. Comm. ACM 6 (1963), 1-17; Comp. J. 5 (1962/1963) 349-367; Num. Math. 4 (1963), 420-453.

32. N. J. Nilsson: Problem-solving Methods in Artificial Intelligence. McGraw-Hill, 1971.

33. R. Pasero: Representation du français en logique du premier ordre en vue de dialoguer avec un ordinateur. Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Marseille, 1973.

34. R. Reiter: Two results on ordering for resolution with merging and linear format. J. ACM 15 (1968), 630-646.

35. J. A. Robinson: A machine-oriented logic based on the resolution principle. J. ACM 12 (1965), 23-44.

36. J. A. Robinson: Heuristic and complete processes in the mechanization of theorem-proving. Systems and Computer Science, J. F. Hart and S. Takasu (eds), University of Toronto Press, 1967.

37. D. Warren: WARPLAN: a system for generating plans. DCL Memo 76, Dept. of Artificial Intelligence, University of Edinburgh.

38. T. Winograd: Understanding Natural Language. Academic Press and Edinburgh University Press, 1972.