SCHOOL OF ARTIFICIAL INTELLIGENCE
UNIVERSITY OF EDINBURGH

Memorandum: MIP-R-106
Date: May, 1974.

Title: First-order predicate logic as a high-level program language

Author: M. H. van Emden

Abstract

This paper presents an argument in support of the thesis that first-order predicate logic would be a useful next step in the development towards higher-level program languages. The argument is conducted by giving a description of Kowalski's system of logic which is sufficiently detailed to investigate its computational behaviour in the two examples discussed: a version of the "quicksort" algorithm and a top-down parser for context-free languages.

Keywords

Program language, specification language, automatic programming, resolution theorem-proving.

CONTENTS

0. Introduction

Kowalski has argued that predicate logic is a language useful for stating problems and that recent developments in resolution proof procedures make it possible for problems stated thus to be solved automatically. This is of interest to those concerned with computer-aided problem-solving, which encompasses not only conventional programming, but also the new applications being studied in Artificial Intelligence and elsewhere in computer science research.

The present paper is intended as a supplement to Kowalski's arguments [14, 15]. It is useful to conduct an argument from the point of view of the development of program languages as proceeding from the low-level to the high-level. I will first argue that each higher level of program language represents a step towards "automatic programming", a level where programs become more like descriptive specifications of algorithms and less like sequences of commands. Using a formulation of automatic programming due to Green [8], I show that predicate logic, in Kowalski's interpretation as a system for stating and solving problems, is a possible and useful next step in the development of program languages.

The argument depends critically on some details of the proof procedure used. In the early experience of resolution theorem-proving, proofs are usually obtained only after searching a, frequently large, search space. The computations performed by an interpreter for predicate logic programs would be proofs from a logical point of view. It is, therefore, important to investigate the computational behaviour of Kowalski's system. In particular, it is of decisive importance that such an interpreter will not have to search in situations where a conventional version of the same program would not search.

The contributions of this paper supporting the feasibility of first-order predicate logic as a high-level program language are the following:

1. The proof procedure is sufficiently precisely defined to study its computational behaviour.
2. The examples on which computational behaviour is investigated have some interest in programming: a version of the "quicksort" algorithm and a top-down parser for context-free languages.

1. The rôle of specification language in automatic programming.

Automatic programming means the automation of some kind of program writing. To take a specific example, think of writing PL-1 programs, a profitable target indeed 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 machine-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? To do so would be in the same spirit as designing the PL-1 machine to hold a pencil and to write characters on paper.

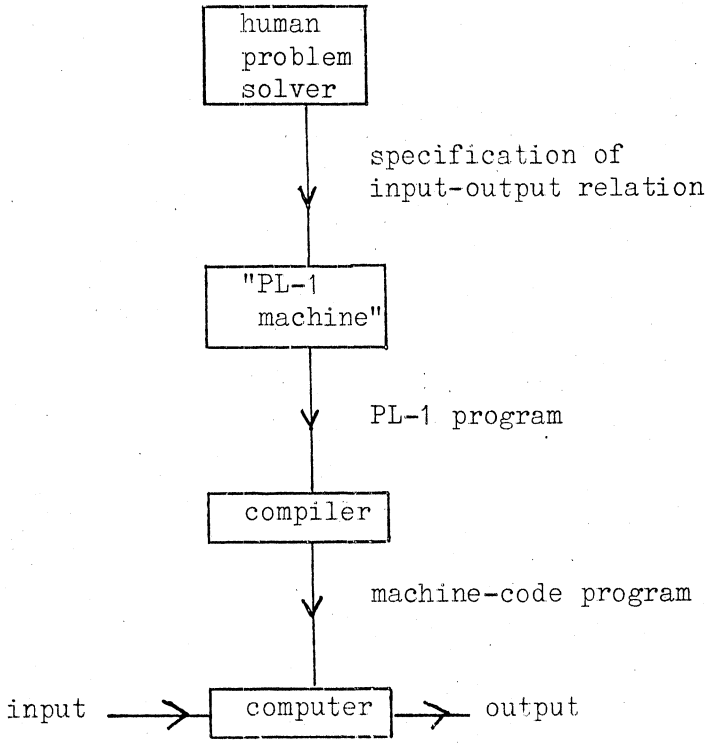

figure 1: Environment of the "PL-1 machine"

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. 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 algorithms specification

The preceding observations suggest that there is no 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, 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 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 imperative value. 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 [4,6,7] is an interesting experiment that allows algorithms to be specified in a more descriptive manner.

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/

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. Contributions of Green and of Kowalski

Green [8] 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 outputs). Even if the program computes a total function, it is advantageous to specify 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 of axioms, not necessarily the same for each task) in the process of proving a theorem of a particular form. Figure 2 shows 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 2: 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 figure 2, 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 do we need the synthesised program if, for any input, we can get by simulation the required output without the program?

This 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 theorem-prover of Kowalski and Kuehner [16], or each of several related proof-procedures [22, 17], can be adapted to act like a program language interpreter for logic axioms in the form of "Horn-clauses". The programming interpretation of Horn clauses is Kowalski's contribution [14] to the pragmatics of predicate logic.

The use of predicate logic discussed in this paper has some features in common with that of Hayes [9], 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 remainder of this paper is devoted to the application of Kowalski's work to the simulation method of automatic programming, which is applied to two problems: sorting a list and parsing a string generated by a context-free grammar. The purpose of the first example is to draw attention to the fact that an autonomous resolution proof-procedure is capable of computationally acceptable behaviour. This may be incompatible with widely-held opinions on this point. To give an example of such an opinion, I quote Hayes [9]:

> "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 the practical evidence mentioned here less abundant. What I can do is to add some evidence for the contrary opinion that autonomous resolution proof-procedures can be computationally useful.

In the example of parsing the use of predicate logic achieves a degree of automatic programming that is beyond that of conventional high-level program languages. In order to be able to observe the computational behaviour of Kowalski's system, it has to be studied in detail./

detail. To make this paper self-contained, the system will be expounded in full.

4. Predicate logic as a language for stating problems

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 are found, for instance, in [11]. J. A. Robinson's syntax for first-order predicate logic [23] is called "machine-oriented" because it has important advantages for automatic deduction. The most significant feature of Robinson's syntax is the inference system, which contains no axioms and one rule of inference: resolution. The language of Robinson's syntax is called the "clausal form" of first-order predicate logic. For a helpful exposition of machine-oriented logic the reader is referred to [21].

As an example of the several variations of language, consider a sentence we shall meet later on. In the usual language it reads as

$$\forall x,y,z.\left[\exists v_1,v_2,v.\ \mathrm{Conc}(v_1,\mathrm{cons}(x,v_2),z)\wedge\mathrm{Conc}(v_1,v_2,v)\wedge\mathrm{Perm}(y,v)\right]\supset\mathrm{Perm}(\mathrm{cons}(x,y),z).$$

In clausal form the same sentence would read as

$$\mathrm{Perm}(\mathrm{cons}(x,y),z)\,\overline{\mathrm{Conc}}(v_1,\mathrm{cons}(x,v_2),z)\,\overline{\mathrm{Conc}}(v_1,v_2,v)\,\overline{\mathrm{Perm}}(y,v).$$

Read disjunction ("or") between formulas; all variables are understood to be universally quantified. In the language of Kowalski's system of first-order predicate logic, to be defined below, the same sentence reads as

$$\mathrm{Perm}(\mathrm{cons}(x,y),z)\leftarrow\mathrm{Conc}(v_1,\mathrm{cons}(x,v_2),z),\mathrm{Conc}(v_1,v_2,v),\mathrm{Perm}(y,v).$$

That this is indeed the same sentence, is explained by the informal semantics in section 4.2.

First-order predicate logic is an important tool in the methodology of the deductive sciences. A more recent application, stimulated/

stimulated by Robinson's machine-oriented syntax and the constructive nature of its deductions, is to automatic problem-solving. The application to automatic programming discussed in section 3 is an example. The different languages of predicate logic are related to its applications. In particular, Kowalski's system of language, solution step, and solution procedure is suitable for the use of predicate logic as a system for goal-oriented problem-solving. It is useful to give, as I will do here, a self-contained account of the system sufficiently detailed to study its computational behaviour on nontrivial examples.

As explained already, the system may be interpreted as first-order predicate logic: the solution step is the resolution rule of inference; solutions are proofs. It is the purpose of this paper to argue that it may equally well be interpreted and used as a high-level program language: the solution step is a generalized subroutine call and the solution procedure is a strategy for sequencing the execution of called subroutines. While I want to be free to use either interpretation for pragmatic purposes, I prefer to remain neutral towards them and to regard Kowalski's system primarily as one for goal-oriented problem-solving.

4.1 Syntax of the language

A sentence is a set of clauses. A clause is an ordered pair of sets of atomic formulas separated by a backward arrow:

$$B_1, \ldots, B_m \leftarrow A_1, \ldots, A_n$$

An atomic formula has the form $P(t_1, \ldots, t_k)$ where P is a k-place predicate symbol and the t_i are terms. A term is either a variable or an expression $f(t_1, \ldots, t_k)$, where f is a k-place function symbol and the t_i are terms. For the sets of all predicate symbols, function symbols, and variables we are free to choose any three mutually disjoint sets of symbols. Constants are 0-place function symbols.

In the examples of this paper, predicate symbols are words starting with a capital letter. Variables are lower-case letters, possibly indexed, near the end of the alphabet. Function symbols are other/

other identifiers consisting of lower case letters.

4.2 Semantics of the language

The meaning of a sentence $\{C_1, C_2, \ldots, C_n\}$ is the conjunction:

C_1 and C_2 and ... and C_n.

The meaning of a clause $B_1, \ldots, B_m \leftarrow A_1, \ldots, A_n$ containing variables $x_1, \ldots, x_k$ 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 where $m = 0$ 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 $m = 0$, write the null clause

$\square$

and read it as denoting contradiction.

Example

In Green's formulation of simulation (see figure 2) there must be a set A of axioms containing a predicate symbol R with the property that

$$A \models R(a,b)$$

if and only if the required program has to give output b for input a. The task of obtaining the output x for given input a by simulation is defined as proving

$$A \models \exists x.\ R(a,x),$$

where the proof constructs the x that exists, which is the required output.

In Kowalski's system (to give an example of its use) this is stated as/

as the task of deriving the null clause $\square$ from the set of axioms

$$A \cup \{\leftarrow R(a,x)\}$$

4.3 The problem of sorting stated in the language

Let us use the language of Kowalski's system to express the specification of a sorting algorithm. From a logical point of view the specification is a logical theory in which some terms denote lists and where the binary relation of "sortedness" is defined between terms denoting lists.

The constants are the atoms 0,1,2, ... and the list nil. There is one function symbol, cons, which is used to construct lists as follows: whenever x is an atom and y is a list, then cons(x,y) is a list again. Thus, for example,

cons(1, cons(8,cons(2,nil)))

is a variable-free term of the theory and it denotes the list of 1 followed by 8 followed by 2. Note that there are terms which are neither atoms nor lists, for instance: cons(1,8) and cons(nil,nil).

Consider the following clauses:

A1. Ord(nil) ←

A2. Ord(cons(x,nil)) ←

A3. Ord(cons(x,cons(y, z))) ← Less(x,y),Ord(cons(y, z))

Here Less(x,y) is true if and only if x and y are atoms and if they are in a given total order among atoms. The meaning of Ord as specified by $\{A_1,A_2,A_3\}$ is defined to be (see [5] for the semantics of such definitions) the set of all variable-free terms l such that the set of clauses

$$\{A_1, A_2, A_3, \leftarrow Ord(l)\}$$

derives the null clause. For instance, nil, cons(nil,nil), cons(1, cons(8,nil)) are in the set (provided that Less(1,8) is true, because the meaning of Ord depends on the meaning of Less). For instance, cons(8,1) and cons(1,8) are not in it. In general, if l is an ordered list, then it is in the meaning of Ord. The converse is not true; for instance, cons(nil,nil) is in the meaning of Ord, although not a list as defined here.

In a similar way a predicate Perm (abbreviation of "permutation") is defined:

A4. $Perm(nil,nil) \leftarrow$

A5. $Perm(cons(x,y),z) \leftarrow Conc(v_1,cons(x,v_2),z),Conc(v_1,v_2,v),$
$Perm(y,v)$

Where $Conc(x,y,z)$ is true if and only if x,y, and z are lists and z is the result of concatenating x an y in that order. With these auxiliary predicates the relation of sortedness is defined as follows:

A6. $Sort(x,y) \leftarrow Perm(x,y),Ord(y)$

For this specification I claim that the set of clauses

$$\{\text{Axioms for Less and Conc}\} \cup \{A_1,\ldots,A6, \leftarrow Sort(l_1,l_2)\}$$

derives the null clause if and only if
whenever l_1 is a list, then so is l_2 and l_2 is the sorted version of l_1.

It is possible to use a complete and correct automatic proof procedure to sort an arbitrary list l by making it derive the null clause from

$$\{\text{Axioms for Less and Conc}\} \cup \{A_1,\ldots,A_6, \leftarrow Sort(l,y)\}$$

The derivation has as side effect the construction of a y that is the sorted version of l. This is automatic programming in the simulation mode. I do not know of a proof procedure that does any better with these axioms than to generate a permutation of l until it is found to violate orderedness, then to generate the next, and so on. Although this is an "algorithm" in the strict sense of the word, it is so extremely inefficient that it is not acceptable under any circumstances.

Although it would be universally agreed that this application of simulation is a useless substitute for an efficient program, not everybody would agree on the cause of its failure. Most work on automatic theorem-proving prior to about 1970 seems to have assumed that such a disappointing result could be cured by improvements in search strategy or by elimination of redundancies in the search space. The subsequent lack of success caused most workers in automatic problem-solving to discard uniform proof-procedures altogether and to emulate the more pragmatically motivated methods advocated by Minsky and Papert. (see, for instance, "Uniform Procedures vs. Heuristic Knowledge" in [20]).

In the example of sorting, however, there is no need to take recourse to such methods: the cause of the disappointing result is the specification. In fact, I suspect that the clause A6 is such that no automatic theorem-prover whatever can elicit an acceptable sorting algorithm from it. In the simulation mode of automatic programming this clause acts as the prescription of a problem-reduction-step: the problem of finding a sorted version of x is reduced to that of finding a permutation that is also ordered.

But in this reduction none of the subproblems is such that it can be solved independently of the other: instead of solving a subproblem once and for all, many trial solutions have to be generated and tested for compatibility with trial solutions for the other subproblem. This seems to be a general characteristic of those reductions, like the one in A6, that fail the criterion:

1) subproblems must be independent.

Furthermore, I suspect that checking whether y is a permutation of x is not computationally less complex than sorting x. This suggests another criterion for effective problem reduction that the axioms $\{A_1,\ldots,A_6\}$ fail to meet:

2) the subproblems must be computationally less complex.

However, there is a well-known way of defining sortedness that does satisfy criteria 1) and 2): it is according to the principle of the quicksort algorithm [10] of C.A.R. Hoare. A specification using this principle is as follows.

B1. $\mathrm{Sort}(\mathrm{nil},\mathrm{nil}) \leftarrow$

B2. $\mathrm{Sort}(\mathrm{cons}(x,y),z) \leftarrow \mathrm{Part}(x,y,u_1,u_2),\mathrm{Sort}(u_1,v_1),\mathrm{Sort}(u_2,v_2),\ \mathrm{Conc}(v_1,\mathrm{cons}(x,v_2),z)$

B3. $\mathrm{Part}(x,\mathrm{nil},\mathrm{nil},\mathrm{nil}) \leftarrow$

B4. $\mathrm{Part}(x,\mathrm{cons}(y,z),\mathrm{cons}(y,v_1),v_2) \leftarrow \mathrm{Less}(y,x),\mathrm{Part}(x,z,v_1,v_2)$

B5. $\mathrm{Part}(x,\mathrm{cons}(y,z),v_1,\mathrm{cons}(y,v_2)) \leftarrow \mathrm{Gr}(y,x),\ \mathrm{Part}(x,z,v_1,v_2)$

The meanings of Gr and Less (specified by axioms not shown here) are complementary: Gr(x,y) true if and only if Less(x,y) is false. As in A3, Less denotes a total order on the atoms. I claim that

$\{\text{Axioms for Conc, Less, and Gr}\} \cup \{B_1,\ldots,B_5,\leftarrow \mathrm{Sort}(l_1,l_2)\}$

derive the null clause if and only if

whenever l_1 is a list, then so is l_2, and l_2 is the sorted version of l_1. In such a derivation, the effect of Part(x,y,u_1,u_2) is to partition a list y into two lists: u_1 containing atoms not greater than the atom x, and u_2 containing atoms greater than x. This effect is determined by the clauses B_3, B_4, and B_5.

The problem reduction in B2 satisfies the criteria 1) and 2). Simulation using a suitably modified SL-resolution theorem-prover sorts an arbitrary input list as efficiently as one can do it in a conventional program language using only procedure calls. This should not be surprising: the specification not only acts like a program; it also reads like one.

Apart from the language discussed in this section, Kowalski's system also has a solution procedure which is based on the SL-resolution method. In order to be able to explain why simulation using the specification $\{B1,\ldots,B5\}$ behaves so differently from what one has become used to in resolution theorem-proving, I have no choice but to expound in full the remaining part of Kowalski's system.

5. The solution procedure of Kowalski's system

5.1 Horn clauses

A Horn clause is a clause

$$B_1,\ldots,B_m \leftarrow A_1,\ldots,A_n$$

where $m \leqslant 1$. The following four kinds of Horn clauses all have a logical interpretation, as explained under the semantics of the language. Here their interpretations are given according to the programming or problem-solving point of view (due to Kowalski [14,15]).

$n \geq 0$ and $m \geq 0$:

$$B \leftarrow A_1,\ldots,A_n$$

is interpreted as a procedure definition. The conclusion B is interpreted as the procedure name. The antecedent $A_1,\ldots,A_n$ is interpreted as the procedure body. It consists of a set of procedure calls.

Alternatively,/

Alternatively, interpret the clause as a prescription for a reduction to subproblems: B is solved if each of the sub-problems in the set $\{A_1,\ldots,A_n\}$ is solved.

n = 0:

$$B \leftarrow$$

is interpreted as an assertion of fact. It can be interpreted as a special kind of procedure which has an empty body. Alternatively, interpret the clause as a problem already solved.

m = 0:

$$\leftarrow A_1,\ldots,A_n$$

is a goal statement; interpret it as a set of procedure calls to be executed or as a set of problems to be solved. If n = 0, then the goal statement is also a halt statement: there are no more procedures to be called, or problems to be solved.

Notice that there are only Horn clauses in both specifications $\{A_1,\ldots,A_6\}$ and $\{B_1,\ldots,B_5\}$. With the procedural interpretation in mind, the latter reads almost like a program in a procedure-oriented language. An unfamiliar feature is the occurrence of a term like cons(x,y) in a procedure name, instead of a variable. Another unfamiliar feature is the occurrence of more than one procedure definition with the same predicate symbol in the name.

It is no coincidence that only Horn clauses occur in the examples A and B. Only such clauses will be used at all. Only such clauses have a pragmatics according to the programming or problem-solving interpretation. Indeed, only for Horn clauses will the solution procedure of Kowalski's system be defined. A specification will mean a set of Horn clauses containing no goal statement. In the examples above $\{A_1,\ldots,A_6\}$ and $\{B_1,\ldots,B_5\}$ are specifications.

The fact that most computer hardware is based on bi-stable switching elements is the basis of the well-known correspondence between computer operations and the logical operations of propositional logic (logic without variables): one of the stable states is interpreted as truth; the other as falsity. Note that such a correspondence plays no rôle in the programming interpretation of Horn clauses, which are a sub-language of predicate logic. The suitability of the language for computer/

computer implementation is not based on some special relation between hardware and predicate logic, but on its similarity with conventional high-level program languages.

The difference between these correspondences between logic and programming is similar to the difference noted by Minsky between two possible ways of looking at a computer. In his introduction to "Semantic Information Processing" [19] he writes:

" ... the dreadfully misleading set of concepts that people get when
" they are told (with the best intentions) that computers are nothing
" but assemblies of flip-flops; that their programs are really nothing
" but sequences of operations on binary numbers, and so on. While this
" is one useful viewpoint, it is equally correct to say that the
" computer is nothing but an assembly of symbol-association and process-
" controlling elements and that programs are nothing but networks of
" interlocking goal-formulating and means-ends evaluating processes.

I find it helpful to be neutral towards the choice between the logic and the program interpretation of the language (though using either when convenient) and to think of it as a language for stating problems to be solved by the solution procedure of the system. In the logic interpretation a solution is a refutation and the solution procedure is a version of the SL-resolution proof procedure. In the program interpretation a solution is a computation and the solution procedure is an interpreter.

5.2 The solution step

The solution procedure is explained by means of a solution step which derives a new goal statement from an existing one in the following way. There is a selection rule which selects a subgoal A_i in a goal statement

$$\leftarrow A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_n$$

If there is a clause

$$A' \leftarrow A'_1, \ldots, A'_m$$

that "matches" the selected subgoal in the sense that there exists a most/

"most general" substitution θ of terms for variables that makes A_i and A' identical (see for instance [21]), then the solution step derives the goal statement

$$\leftarrow (A_1,\ldots,A_{i-1},A'_1,\ldots,A_m',A_{i+1},\ldots,A_n)\theta$$

From a logical point of view this is a resolution step; other interpretations are as a problem reduction step or as a replacement of a procedure call by a procedure body.

It may happen that the predicate in the selected subgoal does not occur in any procedure name. Such a predicate is assumed to be primitive: its meaning is not a relation determined by the specification, but it is computed by hardware or lower-level software. In order to determine the next goal statement when the selected subgoal has a primitive predicate, one may act as if there were added to the specification a set of variable-free assertions containing this predicate such that they constitute a listing of the meaning of the primitive predicate. For instance, in the case of Less the set would include the assertions:

Less(0,0) ←	Less(1,1) ←	Less(2,2) ← ...
Less(0,1) ←	Less(1,2) ←	...
Less(0,2) ←	...	
...		

I shall illustrate the solution step by means of the specification $\{B_1,\ldots,B_5\}$. Suppose we want to sort the list cons(1,cons(8,cons(2,nil))); application of the solution step to the goal statement (containing only one subgoal for selection):

B_0. ← Sort(cons(1,cons(8,cons(2,nil))),y)

and the only matching procedure

B_2. Sort(cons(x,y),z) ← Part(x,y,u_1,u_2), Sort(u_1,v_1)Sort(u_2,v_2), Conc(v_1,cons(x,v_2),z)

derives the goal statement

B6. ← Part(1,cons(8,cons(2,nil)),u_1,u_2),Sort(u_1,v_1),Sort(u_2,v_2), Conc(v_1,cons(1,v_2),y)

5.3 Selection rule and search strategy/

5.3 Selection rule and search strategy

When the halt statement is derived, the problem is solved. For an explanation of how to achieve such a derivation it is useful to define a tree of which the nodes are goal statements: the root is the unique original goal statement (like B_0); a node n has successors n_1, ...,n_k (k = 0,1,...), one for each procedure definition matching the selected subgoal (and no other successors).

A solution is a path in the tree ending in a halt statement. Besides the selection rule and the solution step, the solution procedure requires a search strategy for finding a halt statement in the tree.

Let us now proceed from the goal statement B6 in the example. To select anything but the first subgoal would be disastrous because the other ones can be attained in many ways and, in this example, only one is right, as will now be explained. Which subgoal to select depends on the number of solutions it has without taking the other subgoals into account. For instance, Sort(u_1,v_1) is solved by any of the infinitely many pairs for which the sortedness relation holds. At this stage it is not yet known that only one of these, (nil,nil), is compatible with solutions to the other sub-problems. Later on, more information becomes available transforming Sort(u_1,v_1) into Sort(nil,v_1) which has one solution: v_1:=nil, independently of solutions to other subproblems. Therefore, the selection rule should prefer for selection a subproblem with only one solution. If there is a subproblem with no solution at all, that is a better selection still: the subtree with that problem as root node is empty.

For reasons to be explained below, the first two arguments of Less, Gr, Part, and Conc, and the first argument of Sort, are designated to be i-arguments. This allows the selection rule to be stated as

Select a subgoal of which
the i-arguments are variable-free ... (5.1)

In the programming interpretation this rule corresponds to the calling of a procedure only when its input parameters have received a value.

The/

The selection rule requires that the first subgoal of B6 be replaced. The matching procedures are B4 and B5; the two descendants of B6 in the tree are

$B7^a$: $\leftarrow$ Less(8,1),Part(1,cons(2,nil),u_1,u_2),Sort(cons(8,u_1),v_1),
Sort(u_2,v_2),Conc(v_1,cons(1,v_2),y).

$B7^b$: $\leftarrow$ Gr(8,1),Part(1,cons(2,nil),u_1,u_2),Sort(u_1,v_1),
Sort(cons(8,u_2),v_2),Conc(v_1,cons(1,v_2),y)

In $B7^a$ the selected subgoal is to check Less(8,1). The predicate Less is primitive; its meaning is such that the selected subgoal Less (8,1) is impossible to achieve. Therefore, $B7^a$ can be deleted, because it has no descendants in the tree of goal-statements, and a solution path, if one exists, must pass through $B7^b$.

It seems feasible to go a step further: never generate $B7^a$ or $B7^b$ at all and go directly from B6 to B7: which is $B7^b$ minus the supposedly achieved subgoal Gr(8,1):

B7 $\leftarrow$ Part(1,cons(2,nil),u_1,u2),Sort(u_1,v_1),Sort(cons(8,u_2),v_2),
Conc(v_1,cons(1,v_2),y)

We have the situation where one of the descendants of B6 has Less(8,1) as selected subgoal, the other has Gr(8,1), and only one is achievable. A <u>conditional solution step</u> makes the transition directly from B6 to B7 using the information that just one of Less(y,x) and Gr(y,x) is provable. It corresponds to the execution of a conditional statement in a program language.

If both the solution step and the conditional solution step are available, then the tree of goal statements contains only a single path. According to the programming interpretation the sentence $\{B_1,\ldots,B_5\}$ reads like a program for the quicksort algorithm written entirely with procedures in a program language. The successive goal statements read like the successive states of the stack of procedure calls to be executed.

The tree of goal statements contains one path; there is no opportunity for search. For this example, an autonomous resolution proof/

proof procedure achieves the same level of efficiency as a conventional program language. This makes the behaviour of Kowalski's system a notable exception to earlier experience with resolution proof procedures. Still, the specification $\{B_1,\ldots,B_5\}$ is not efficient by programming standards, because it is like a program in an Algol-like language constrained to do everything by procedure call. This and other sources of inefficiency are within reach of optimization techniques like those investigated by Darlington and Burstall [5].

This favourable result was obtained by the use of a good proof procedure and by some care in writing the specification. It is useful to formulate a condition sufficient to guarantee that the tree of goal statements contains only one path with the use of a computationally simple selection rule: select always the left-most subgoal. This is analogous to the behaviour of an interpreter for conventional program languages, which always executes the procedure call on top of the stack of calls to procedures awaiting execution.

To establish a sufficient condition for the tree of goal statements to have one path when the leftmost subgoal is always selected, effect a partition in the set of arguments of each predicate P into a set of <u>input-arguments</u> and a set of <u>output-arguments</u> which satisfies the following three conditions.

1) Whenever the selected subgoal has P as predicate and all input arguments are variable-free, then the goal-statement has only one descendant.

To verify whether this is true for a given partition, distinguish two cases. Suppose P is a <u>nonprimitive</u> predicate. In this case, ascertain that the subgoal matches one procedure name only, or else that a conditional solution step is available to ensure that there be only one descendant. Note that it depends indeed on input-arguments being variable-free whether a selected subgoal matches only one procedure name. For instance, if the first argument of a subgoal Sort(...,...) is variable-free, then it matches at most one of Sort(nil, nil) (the name in B_1) and Sort(cons(x,y),z) (the name in B_2). If it is a variable, then it matches both.

Suppose/

Suppose P is a primitive predicate. The meaning of the predicate is a relation between variable-free terms. It depends on this relation whether variable-free input-arguments uniquely determine the output-arguments, which is necessary for such a goal statement to have one descendant. For instance, if the first two arguments of a subgoal Conc(...,...,...) are variable-free, then the relation uniquely determines the third argument. If the first two arguments are variables, then this is in general not the case, and there would be more than one descendant.

Note that the i-arguments of the selection rule (5.1) are input-arguments in this sense. For the tree of goal statements to have one path it is sufficient to ensure that the input arguments of the selected subgoal are always variable-free. It remains to formulate conditions under which the leftmost subgoal is always in this state.

2) The calls of each procedure body can be ordered in such a way as to be sequential, that is, for every procedure call of the body, each variable in an input-argument occurs in an argument of a preceding procedure call or in an input argument of the procedure name.

If each procedure body is sequential, and if the input-arguments of the initial goal statement are variable-free, then the leftmost subgoal of every goal statement has its input arguments variable-free, provided that

3) Every subgoal is such that, when it is achieved, the corresponding substitution causes variable-free terms to be substituted for a variable occurring in an output argument of the subgoal.

This condition is not difficult to verify when, as in $\{B_1,\ldots,B_5\}$ there is no mutual recursion. For instance, to verify that a subgoal containing Sort has the property 3), assume that all arguments in the body of B_2 are variable-free and verify that this also holds for z in the name of B_2. To conclude that Sort has property 3) is an example of circular reasoning because of the recursive nature of B_2.

However,/

However, the circularity is not vicious because the output arguments of the calls to Sort in B_2 represent lists that are at least one atom shorter than the one in the procedure name.

6. A top-down parser for unrestricted context-free languages

The problem of parsing can be formulated in such a way that parse trees are represented by terms and then the problem becomes one of simulation with output a parse tree. However, in this example I will view parsing as a task where the required answer is only a "yes" or a "no", indicating whether the parsed string is grammatical. For the application of automatic programming to such a task I will use Green's mode of "checking" (figure 2) and obtain the parse from the path in the tree of goal statements from the root to the halt statement. The example will show that the required specification has to contain only a straightforward transcription of a context-free grammar and a representation of the string to allow Kowalski's system to simulate a reasonable top-down parsing algorithm.

This is a level of automatic programming beyond that provided by a conventional language. Of course, programs exist that accept an arbitrary context-free grammar and use it to parse strings, but such programs will not do anything else. Here the program that accepts the grammar and parses strings is the solution procedure of Kowalski's system, which is generally applicable as an interpreter for a procedure-oriented program language.

The specification of the parsing problem has as individual constants a number of "markers" which identify the positions in the input string between successive terminal symbols. Suppose the input string is

$$\{a = \{b + a\}\}$$

where markers $0,1,\ldots,9$ are inserted as follows

$$0\{_1\ a_2 =_3 \{_4\ b_5 +_6\ a_7\}_8\}_9 \qquad \ldots(6.1)$$

The/

The following clauses specify the input string

C1. {(0,1) ←
C2. a(1,2) ←
C3. =(2,3) ←
C4. {(3,4) ←
C5. b(4,5) ←
C6. +(5,6) ←
C7. a(6,7) ←
C8. }(7,8) ←
C9. }(8,9) ←

These are all procedures with a name but no body. The terminal symbols of the string (6.1) are used as predicate symbols. The meaning of any of these predicates is that the markers constituting their arguments are connected in the specified way: C4, for example, states that "{" connects 3 and 4.

The string is to be parsed with the following grammar (written in conventional notation):

$$B \rightarrow R \mid \{ B \}$$
$$R \rightarrow E = E \qquad \ldots(6.2)$$
$$E \rightarrow a \mid b \mid \{ E + E \}$$

This example is taken from Knuth's tutorial article [12] on top-down parsing. The grammar generates a simple form of "Boolean expression".

The following clauses are a systematic transcription of the grammar:

C10. $B(x,y) \leftarrow R(x,y)$
C11. $B(x,y) \leftarrow \{(x,x_1), B(x_1,x_2), \}(x_2,y)$
C12. $R(x,y) \leftarrow E(x,x_1), =(x_1,x_2), E(x_2,y)$
C13. $E(x,y) \leftarrow a(x,y)$
C14. $E(x,y) \leftarrow b(x,y)$
C15. $E(x,y) \leftarrow \{(x,x_1), E(x_1,x_2), +(x_2,x_3), E(x_3,x_4), \}(x_4,y)$

Note that alternative productions from the same non-terminal symbol transcribe to different procedures with the same name.

The task of parsing the string (b,1) is expressed by the goal statement

C0. ← B(0,9)

which becomes the root of the tree of goal statements. The string is in the language defined by the grammar iff the tree contains a halt statement. The/

The parse is obtained by tracing the path from the root to the halt statement and by noting at each step which procedure was applied.

A good selection rule for problems like this is the following. If any subgoal is present with a predicate denoting a terminal symbol with at least one argument a constant, then select this. (Such a subgoal is merely to check whether a terminal symbol occurs in a specified place). Otherwise, select any subgoal, preferring one with a greater number of constant arguments.

This selection rule, together with the method for representing production rules and strings as clauses, is all that is required to apply Kowalski's system to the task of parsing strings from context-free languages from the top down. It does not seem too ambitious to envisage automatic generation of the selection rule from the general principle that subgoals that are seen to have fewer solution possibilities should be attempted first. The preference for terminal symbols in the above selection rule then follows from inspection of $\{C1,\ldots,C15\}$. The preference for subgoals having more variable-free arguments is a generally useful heuristic indicating fewer solution possibilities.

In the quicksort example I did not discuss a search strategy for the tree of goal statements, because it consists of a single path. In the parsing examples the search strategy matters: the tree has two substantial branches. Of course, if a solution exists, then there exists a finite path ending in a halt statement, and this path is found by a breadth-first search of the tree. What makes Kowalski's system work so well is that the tree is drastically pruned by deleting all subgoals which can be shown to be unattainable. For instance, in this example a parser must ultimately discover that a successful parse cannot begin with an application of the production rule B→ R. The subtree of figure 4 represents the attempt to find such a parse. It turns out to be necessary to prove that $+(2,v_{31})$, which is impossible simply because the string has a "=" in that place and not a "+".

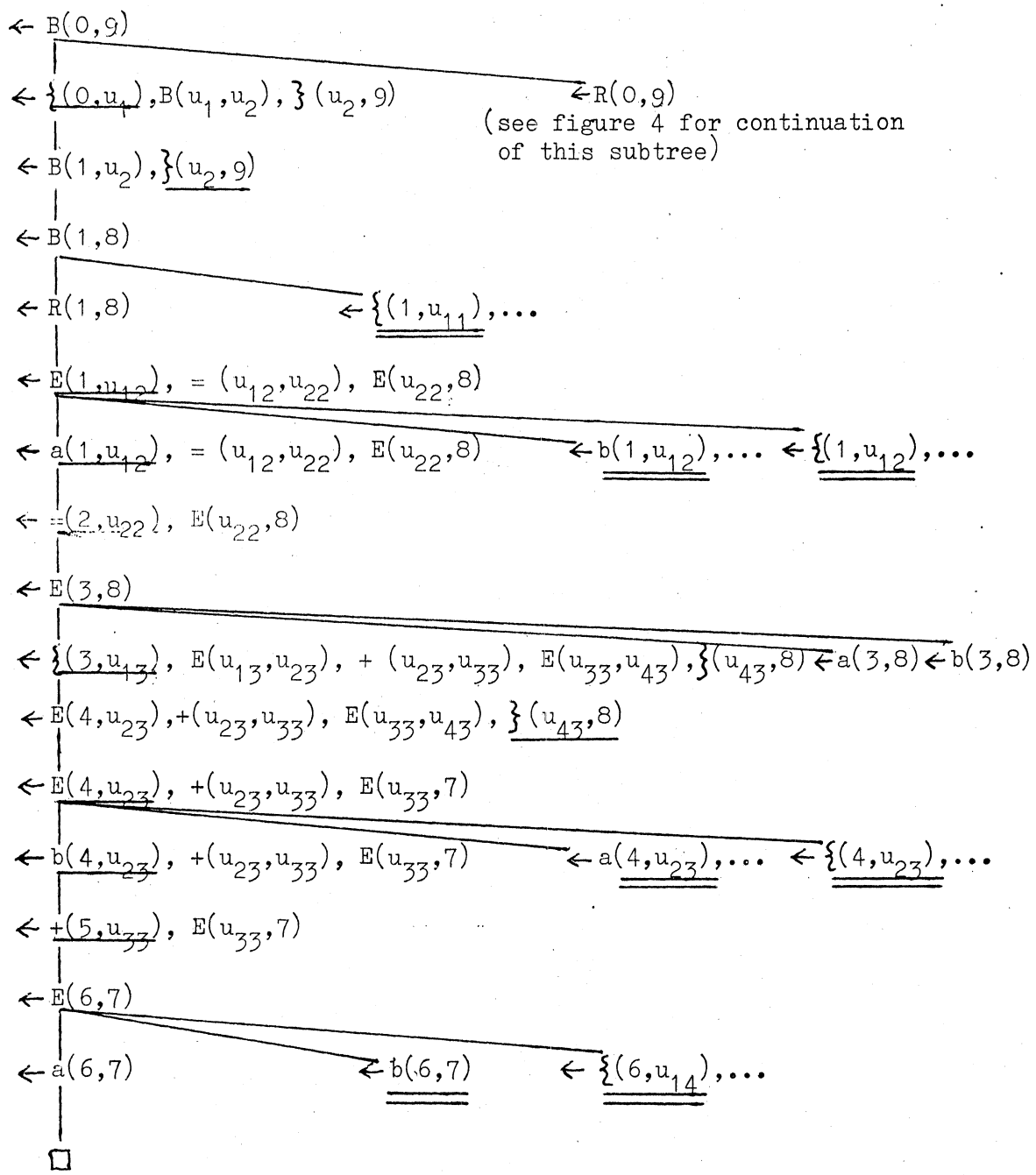

Figure 3. The tree of goal statements generated when parsing the string (6.1) with the grammar (6.2). Goal statements with a doubly underlined subgoal have no successor. Singly underlined subgoals are those selected.

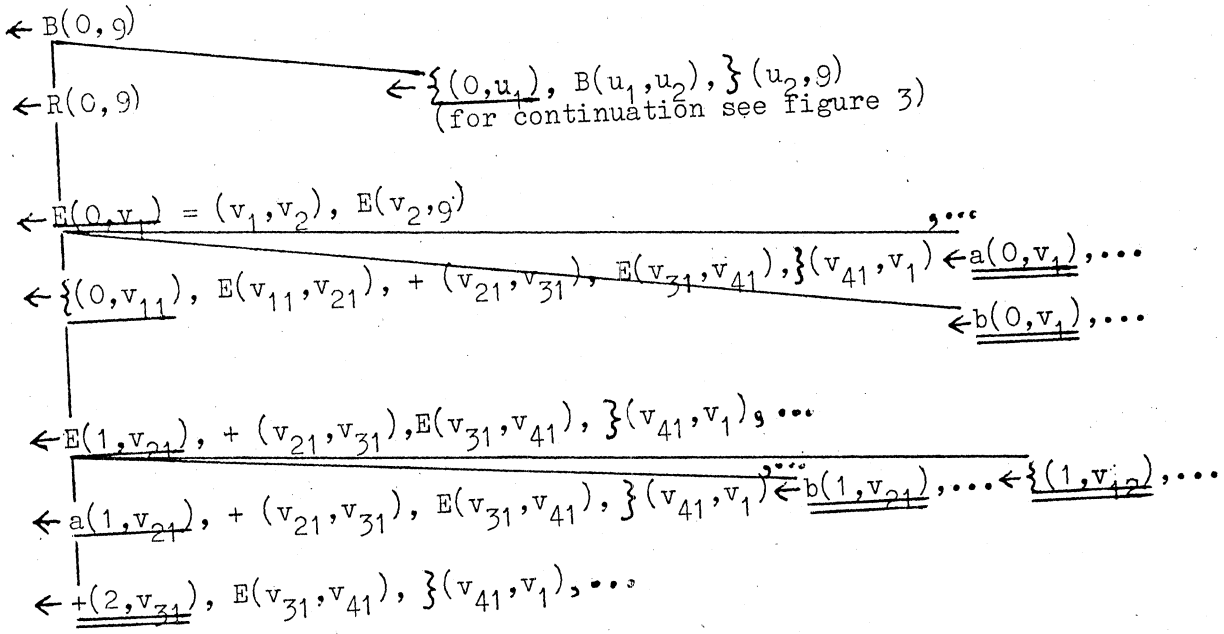

Figure 4. The unsuccessful branch of the tree of goal statements shown in figure 3.

In figures 3 and 4 failures are shown by a double underlining. It is apparent that these failures severely reduce the size of the tree of goal statements. If even a single subgoal is unattainable the entire goal statement can be deleted because all subgoals must be successful. Moreover, if all descendants of a goal statement have been deleted, then it may itself be deleted. Because of these two rules, the failure of the subgoal $+(2,v_{31})$ triggers a progressive collapse which erases the entire subtree of figure 4 except for its root B(0,9).

This representation of the parsing problem was invented by Colmerauer and Kowalski in unpublished work done at the University of Marseille in the summer of 1971. It was suggested by Colmerauer's parsing method using "Q-systems" [2]. Subsequent work by Kowalski on "connection graphs" [13] lifts the restriction to top-down parsing inherent in the goal-oriented nature of the solution procedure described here. When using connection graphs, the solution step would have no bias towards top-down or bottom-up and it could alternate between the two strategies in an advantageous way.

A more elaborate investigation than the one in this example has been reported by Minker and Vanderbrugh [18]. They described two methods for representing formal grammars in first-order predicate logic using Horn clauses only and in each case used as a parser a resolution proof procedure, exploiting the computational advantages of Horn clauses. One of their methods, the "derivation sequence representation" leads to the simulation of a top-down parser, whereas the other simulates bottom-up parsing. Compared with the derivation sequence representation the method of Colmerauer and Kowalski has the advantage that the notions of derivation in logic and in the formal grammar almost/

almost coincide. As a result, the solution step in Kowalski's system is a derivation step for the formal grammar, as shown in the example. In the derivation sequence representation there is a clause for each possible form of derivation step.

7. Acknowledgment

I am indebted to Robert Kowalski for useful discussions and for making available essential material not yet published. The research was supported by a Science Research Council grant to Professor D. Michie, whom I wish to thank for his encouragement and for help with earlier versions of this paper. Many suggestions for improvement were made by Luis Pereira.

8. References to the literature

1 Annual review in automatic programming
R. Goodman (ed.); Pergamon Press

2 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

3 J. Darlington and R.M. Burstall
A system which automatically improves programs
Proc. 3rd International Joint Conf. on Artificial Intelligence, 1973

4 E.W. Elcock
Descriptions
Machine Intelligence 3, B. Meltzer and D. Michie (eds.), 173-179
Edinburgh University Press, 1968

5 M.H. van Emden and R.A. Kowalski
The semantics of predicate logic as a programming language
Report MIP-R-103,/

Report MIP-R-103,
Dept. of Machine Intelligence, University of Edinburgh, 1974

6 J.M. Foster
Assertions: programs written without specifying unnecessary order
Machine Intelligence 3, B. Meltzer and D. Michie (eds.), 387-391
Edinburgh University Press, 1968

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

8 C. Green
The application of theorem-proving to question-answering systems
Technical note no. 8, Artificial Intelligence Group,
Stanford Research Institute, 1969

9 P.J. Hayes
Computation and deduction
Proc. 1973 MFCS Conference
Czechoslovakian Academy of Sciences

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

11 S.C. Kleene
Mathematical Logic
Wiley, New York, 1967

12 D.E. Knuth
Top-down syntax analysis
Acta Informatica 1 (1971), 79-110

13/

13 R.A. Kowalski
An improved theorem-proving system for first-order logic
Memo no. 65, Dept. of Computational Logic,
University of Edinburgh, 1973

14 R.A. Kowalski
Predicate logic as programming language
Memo no. 70, Dept. of Computational Logic,
University of Edinburgh, 1973

15 R.A. Kowalski
Logic for problem-solving
Memo no. 75, Dept. of Computational Logic,
University of Edinburgh, 1974

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

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

18 J. Minker and G.J. Vanderbrugh
Representations of the language recognition problem for a theorem-prover
Technical Report TR-199, Computer Science Center,
University of Maryland, 1972

19 M. Minsky
Semantic Information Processing
MIT Press, Cambridge, Mass., 1968

20/

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

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

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

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

9. Epilogue

Kowalski's system described in this paper is in fact similar to PROLOG, a program language implemented in the University of Marseille by Colmerauer and his colleagues [1,2]. Like Kowalski's system, PROLOG admits only Horn clauses. Unlike Kowalski's system, PROLOG always attempts subgoals from left to right in the program text, thus giving a depth-first backtracking search of the tree of goal statements.

Several ambitious programming tasks have been accomplished in PROLOG. These include a natural language understanding system [1], a formula-manipulation system [3], and a STRIPS-style problem-solver [4]. The first version of PROLOG was implemented in ALGOL-W; the next version has been coded in FORTRAN, resulting in a programming system which may well be competitive in use of machine time for tasks of the type mentioned above. This is suggested by the fact that for the examples tried, Warren's problem-solver is faster than the original STRIPS system. A more important advantage of first-order predicate logic as a high-level program language is suggested by the fact that Warren's problem-solver required about one man-week of programming time.

1) A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel
Un système de communication homme-machine en français
Groupe d'Intelligence Artificielle
Université d'Aix - Marseille, 1972

2) G. Battani and H. Meloni
Interpreteur du language de programmation PROLOG
Groupe d'Intelligence Artificielle
Université d'Aix - Marseille, 1973

3) M. Bergman and H. Kanoui
Application of mechanical theorem-proving to symbolic calculus
Groupe/

Groupe d'Intelligence Artificielle
Université d'Aix-Marseille, 1973

4) D. Warren
WARPLAN: a system for generating plans
Report in preparation, Dept. of Computational Logic
University of Edinburgh