DESCRIBING PROLOG BY ITS INTERPRETATION AND COMPILATION

Since its conception, Prolog has followed a developmental course similar to the early evolution of LISP. Although the version of Prolog described here typifies that currently in use, it should be considered within the framework of language evolution.

JACQUES COHEN

Prolog is a language developed about 10 years ago by Alain Colmerauer and his coworkers at the Artificial Intelligence Group (GIA—Groupe d'Intelligence Artificielle) in Marseilles, France. It is a logic programming language in the sense that its statements are interpreted as sentences of a logic. During the last decade, Prolog has attracted the attention of a fairly large number of European computer scientists who formed research groups actively engaged in refining the language and developing new applications. The adoption of Prolog as the core language for the Japanese Fifth Generation project has increased interest in the language by American computer scientists from both universities and industry.

There is a growing literature on the subject of Prolog, and presently several journals regularly publish articles in this area [34]. Since 1981, European groups have also held yearly symposia on logic programming [32, 40, 41]. More recently, annual conferences have also been held in the United States [20, 21]. A brief review of the available literature appears in the references.

In addition to Colmerauer's article that appears in this issue of *CACM*, there are a few excellent articles that are highly recommended as introductions to Prolog. Kowalski's articles [23, 25] stress the relationship between Prolog-like languages and classical logic. Two introductory articles by Genesereth and Davis have appeared more recently [13, 16]. The theoretical foundations of logic programming are covered in Lloyd's book [27], and a historical account of the evolution of Prolog is described by Robinson [37].

In an effort to teach Prolog to persons who were already fluent in a programming language (say, Pascal or LISP), we found that learning the language could be accelerated by drawing parallels between Prolog programs and programs written in other languages. We also found that explaining the programming machinery needed to interpret Prolog programs was an incentive to learning more about the new language. It was this experience that compelled us to write this article, which is directed to programmers who are at ease with recursion and linked data structures (such as lists). We believe that the most interesting examples of the use of Prolog are in symbolic processing, and we will illustrate the advantages of the language in that context.

This article combines the features of a tutorial and a survey. More specifically, its objectives are

1. to describe the nucleus of an interpreter that can be used in running simple programs,
2. to present examples that make Prolog unique among existing languages,
3. to show how Prolog programs can be compiled,
4. to discuss recent extensions, and
5. to provide a guide to the current literature.

If this self-contained presentation fulfills the above objectives, it should enable the reader to follow the state-of-the-art literature on the subject. The reader is referred to the bibliography in [34] for references to applications using Prolog.

A common misconception about Prolog is that its main application is to prove theorems in predicate calculus. Recall that LISP is a general-purpose programming language based on Church's lambda calculus. By the same token, Prolog is a programming language based on predicate calculus. Its foundations rest indeed on Robinson's automatic theorem-proving theory [35, 36]. But, since its conception, Prolog has been evolving in a manner not unlike the early evolution of LISP. Actually, both languages are still evolving! We ask the reader to consider this article within the framework of language evolution. The version of Prolog described in this article is typical of that in current use. (See the box illustrating the main syntactic differences between the Marseilles Prolog in Colmerauer's article (p. 1296) and the Edinburgh Prolog used in this article.)

This work was partly supported by NSF Grant DCR-85 00881.

© 1985 ACM 0001-0782/85/1200-1311 75¢

The following illustrates through examples the main syntactic differences between the Marseilles (M) Prolog in Colmerauer's article (p. 1296) and the Edinburgh (E) Prolog used in this article.

Variables	(M)	x	a	x'	$x1$[a]
	(E)	X	A	*Xprime*	$X1$[b]
Constants	(M)	123 *abc*[c]			
	(E)	123 *abc*			
Rules	(M)	$a \rightarrow b\ c;$		$a \rightarrow ;$	
	(E)	a :- $b, c.$		$a.$	
Lists	(M)	*a.b.x*.nil		*x.y*	
	(E)	[*a*,*b*,X]		[X|Y]	

[a] Single letters followed by a prime or by digits.
[b] Identifiers starting with an uppercase letter.
[c] Integers or a sequence having more than two letters.

DEFINITION

One concrete syntax for Prolog rules is given by

⟨*rule*⟩ ::= ⟨*clause*⟩.|⟨*unit clause*⟩.
⟨*clause*⟩ ::= ⟨*head*⟩ :- ⟨*tail*⟩
⟨*head*⟩ ::= ⟨*literal*⟩
⟨*tail*⟩ ::= ⟨*literal*⟩ {,⟨*literal*⟩}
⟨*unit clause*⟩ ::= ⟨*literal*⟩

where the curly braces denote any number of repetitions (including none) of the sequence enclosed by the brackets. A Prolog program is a sequence of rules that can be viewed initially as parameterless procedures that call other procedures. For the time being, let us consider the simplest case, where a ⟨*literal*⟩ is a single letter. For example, consider the following Prolog program in which rules are numbered for future reference:

1. *a* :- *b*, *c*, *d*.
2. *a* :- *e*, *f*.
3. *b* :- *f*.
4. *e*.
5. *f*.
6. *a* :- *f*.

In the first rule, *a* is the ⟨*head*⟩, and *b*, *c*, *d* is the ⟨*tail*⟩. The fourth and fifth rules are unit clauses. We assume the rules are stored in a database so that they can be accessed efficiently.

The execution of a Prolog program is triggered by a query, which is syntactically equivalent to a ⟨*tail*⟩. For example, *a*, *e*. is a query. The result of querying the program is a *yes* or *no* answer indicating the success or failure of the query.

There are three ways of interpreting the semantics of Prolog rules and queries. The first is based on logic, in this particular case, on Boolean algebra. The literals are Boolean variables, and the rules express formulas. For example, the first rule is interpreted as

a is true if b and c and d are true

or

$b \wedge c \wedge d \rightarrow a.$

A unit clause such as *e* means $e \equiv true$. The Prolog program is viewed as the conjunction of formulas it defines. The query succeeds if it and the program are simultaneously satisfiable.

In a second interpretation of a Prolog rule, we assume that a ⟨*literal*⟩ is a goal to be satisfied. For example, the first rule states that

goal a can be satisfied if goals b, c, and d can be satisfied.

The unit clause states that the defined goal can be satisfied. As before, the program defines a conjunction of goals to be satisfied. The query succeeds if the goals can be satisfied using the rules of the program.

Finally, in a third interpretation we invoke the similarity between Prolog rules and context-free grammar rules. A Prolog program is associated with a context-free grammar in which a ⟨*literal*⟩ is a nonterminal and a ⟨*rule*⟩ corresponds to a grammar rule in which the ⟨*head*⟩ rewrites into the ⟨*tail*⟩; a unit clause is viewed as a grammar rule in which a nonterminal rewrites into the empty symbol ϵ. Under this interpretation, a query succeeds if it can be rewritten into the empty string, or, equivalently, can be erased.

Although the above three interpretations are all helpful in explaining the semantics of this simplified version of Prolog, the logic interpretation will be used in the remainder of this article.

INTERPRETATION

We will first show how an interpreter can be implemented for the simplified Prolog of the previous section. The rules will be stored sequentially in a database implemented as a one-dimensional array *Rule*[1 .. *n*] and containing pointers to a special type of linear list. Such a list is a record with two fields, the first storing a letter, and the second being either *nil* or a pointer to a linear list. Let the function *cons* be the constructor of a list element, and assume that its fields are accessible via the functions *head* and *tail*. The first rule is stored in the database by

Rule[1] := *cons*('*a*', *cons*('*b*', *cons*('*c*', *cons*('*d*', *nil*)))).

The fifth rule defining a unit clause is stored as

Rule[5] := *cons*('*e*', *nil*).

Similar assignments are used to store the remaining rules.

We are now ready to present a procedure *solve* that has as a parameter a pointer to a linear list and is capable of determining whether or not a query is successful. The query itself is the list with which *solve* is first called. The procedure uses two auxiliary procedures *match* and *append*; *match*(*A*, *B*) simply tests if the alphanumeric *A* equals the alphanumeric *B*; *append*(*L1*, *L2*) produces the list representing the concatenation of *L1* with *L2* (this is equivalent to the familiar *append* function in LISP; it basically copies *L1* and makes its last element point to *L2*).

The procedure *solve*, written in a Pascal-like language, appears in Figure 1. The procedure performs a

```
procedure solve(L: pLIST);
  begin local i: integer;
    if L ≠ nil
      then
        for i := 1 to n do
          if match(head(Rule[i]), head(L))then
            solve(append(tail(Rule[i]), tail(L)));
      else write('yes')
  end;
```

FIGURE 1. An Initial Version of the Interpreter

depth first search of the problem space where the local variable is used for continuing the search in case of a failure.[1] The head of the list of goals *L* is matched with the head of each rule. If a match is found, the procedure is called recursively with a new list of goals formed by adding (through a call of *append*) the elements of the tail of the matching rule to the goals that remain to be satisfied. When the list of goals is *nil*, all goals have been satisfied, and a success message is issued. If the attempts to match fail, the search is continued in the previous recursion level until the zeroth level is reached in which case no more solutions are possible.

For example the query *a*, *e*. is expressed by

solve(cons('a', cons('e', nil)))

and yields two solutions. If we were to print the successful sequence of the list of goals, we would obtain

solution 1: $a, e \Rightarrow e, f, e \Rightarrow f, e \Rightarrow e \Rightarrow nil$;
solution 2: $a, e \Rightarrow f, e \Rightarrow e \Rightarrow nil$.

The entire search space is shown in Figure 2 (p. 1314) in the form of a tree. In nondeterministic algorithms, that tree is called the tree of choices [7]. Its leaves are nodes representing failures or successes. The internal nodes are labeled with the list of goals that remain to be satisfied. Note that, if the tree of choices is finite, the order of the goals in the list of goals is irrelevant insofar as the presence and number of solutions are concerned. Thus, the order of the parameters of *append* in Figure 1 could be switched, and the two existing solutions would still be found. Note that if the last rule were replaced by

a :- *f*, *a*.

the tree of choices would be infinite and solutions similar to the first solution would be found repeatedly. The procedure *solve* in Figure 1 can handle these situations by generating an infinite sequence of solutions. However, had the above rule appeared as the first one, the procedure *solve* would also loop, but without yielding any solutions. This last example shows how important the ordering of the rules is to the outcome of a query.

Let us now temporarily interrupt our description of Prolog interpretation to show how parameters are introduced into Prolog rules. These parameters allow a much more general pattern matching mechanism to take place, extending the *match* function used in this section. It is the combination of the control mechanism using backtracking and the powerful pattern matching capabilities that allows the language to reach its full expressive power.

The introduction of parameters in Prolog rules corresponds to extending the Boolean algebra interpretation to cover a special form of predicate calculus called Horn clauses. Basically, *P*(X, Y, . . .) is true if there are values for X, Y, . . . that render *P* true. The task of the Prolog interpreter is to attempt to find these values.

In the next section, we will discuss one of the most useful procedures in Prolog, namely, the *append* procedure. Following that we will again return to Prolog interpretation in the more general case of procedures containing parameters.

[1] Alain Colmerauer's clock variable corresponds to the recursion level in our procedure *solve*.

EXAMPLES

This section demonstrates the transformation of a functional specification of a procedure into its Prolog counterpart. Consider the LISP-like function *append* that concatenates two lists, *L1* and *L2*:

```
function append(L1, L2: pLIST): pLIST;
  if L1 = nil then append := L2
              else append := cons(head(L1), append(tail(L1), L2));
```

In the above *L1* and *L2* are pointers to lists; a list is a record containing the two fields *head* and *tail*, which themselves contain pointers to other lists or to atoms. The *tail* must point to a list or to the special atom *nil*. The function *cons(H, T)* creates the list whose *head* and *tail* are, respectively, *H* and *T*.

Let us transform the function *append* into a procedure having an explicit third parameter *L3* that will contain the desired result. The local variable *T* is used to store intermediate results.

```
procedure append(L1, L2: pLIST; var L3: pLIST);
  begin local T: pLIST;
  if L1 = nil then L3 := L2
            else
              begin
                append(tail(L1), L2, T);
                L3 := cons(head(L1), T)
              end
  end;
```

The former procedure can be transformed into a Boolean function that, in addition to building *L3*, checks if *append* produces the correct result.

```
function append(L1, L2: pLIST; var L3: pLIST): Boolean;
  begin local H1, T1, T: pLIST;
    if L1 = nil then
      begin
        L3 := L2;
        append := true
      end
    else
    if {There exists an H1 and a T1 such that
        H1 = head(L1) and T1 = tail(L1)}
```

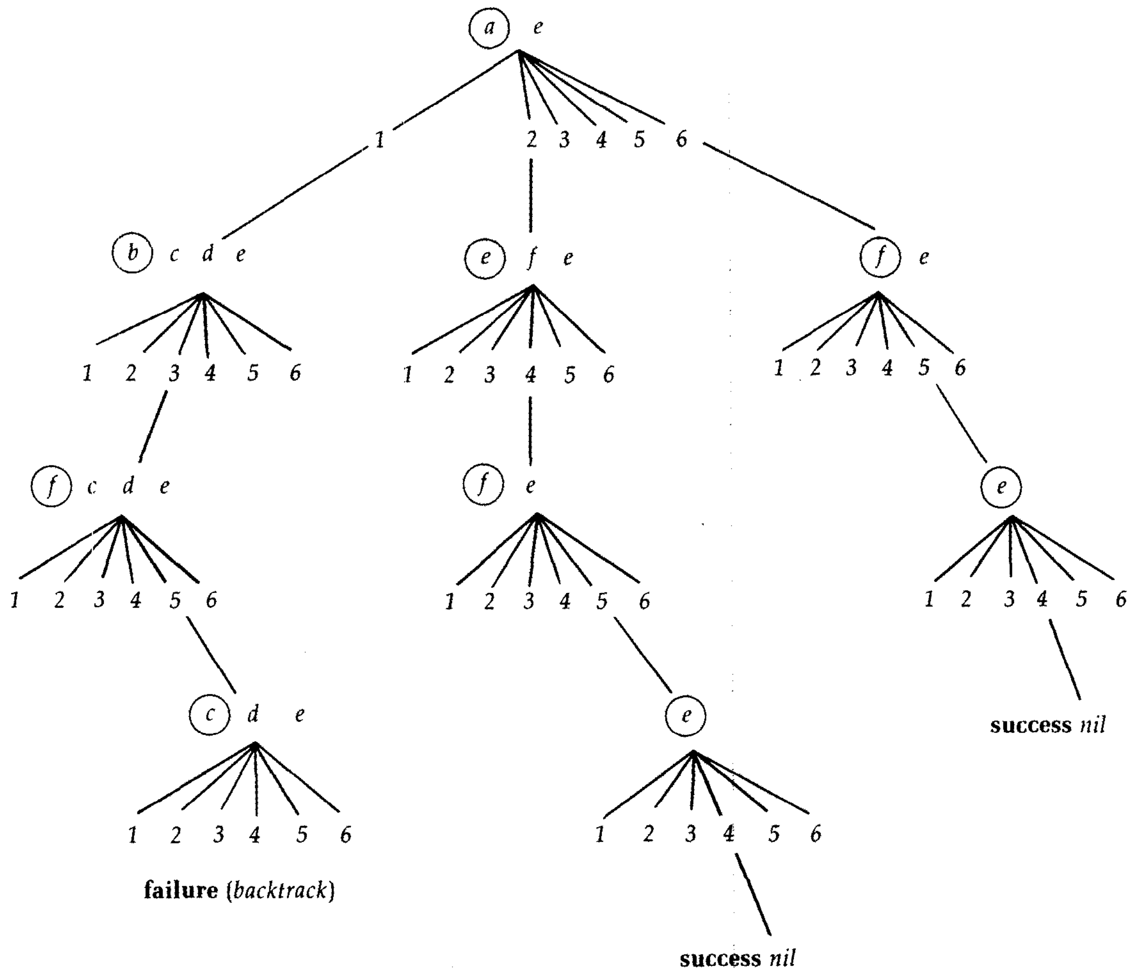


FIGURE 2. The Search Space (also called the tree of choices)

```
      then
        begin
          append := append(T1, L2, T);
          L3 := cons(H1, T)
        end
      else append := false
  end;
```

The Boolean in the second conditional has been presented informally, but it could actually have been programmed in detail. Note that the assignments in the above program are executed at most once for each recursive call. The function returns *false* if *L1* is not a list (e.g., if $L1 = cons(a, b)$ for some atom $b \neq nil$).

We are now ready to transform the last function into a Prolog-like counterpart in which rules of assignments and conditionals are subsumed by a general pattern matching operation called unification and specified by the equality sign. The statement $E1 = E2$ succeeds if *E1* and *E2* can be matched. In addition, some of the variables in *E1* and *E2* may be bound if necessary. We obtain

```
append(L1, L2, L3) is true if  L1 = nil
                               and
                               L3 = L2
                    otherwise
append(L1, L2, L3) is true if  L1 = cons(H1, T1)
                               and
                               append(T1, L2, T)
```

and
$L3 = cons(H1, T)$
otherwise *append* **is false.**

The reader can now compare the above results with the previous description of the subset of Prolog. This comparison yields the Prolog program

```
append(L1, L2, L3) :- L1 = nil, L3 = L2.
append(L1, L2, L3) :- L1 = cons(H1, T1),
                      append(T1, L2, T),
                      L3 = cons(H1, T).
```

In this case, the equality sign is an operator that commands a unification between its left and right operands. This could also be done using the unit clause *unify*(X, X) and substituting *L1* = *nil* by *unify*(*L1*, *nil*), and so on.

Replacing *L1* and *L3* with their respective values in the right-hand side of a clause, we obtain

```
append(nil, L2, L2).
append(cons(H1, T1), L2, cons(H1, T)) :- append(T1, L2, T).
```

The explicit calls to *unify* have now been replaced by implicit calls that will be triggered by the Prolog interpreter when it tries to match a goal with the head of a rule. Notice that a ⟨*literal*⟩ now becomes a function name followed by a list of parameters each of which is syntactically similar to a ⟨*literal*⟩.

The Edinburgh Prolog representation of *cons*(*H*, *T*) is [*H*|*T*], and *nil* is [ ]. The Marseilles counterparts are *H.T* and *nil*. In the Edinburgh dialect, *append* is presented by

```
append([ ], L2, L2).
append([H1 | T1], L2, [H1|T]) :- append(T1, L2, T).
```

The transformations above have been presented in an informal manner with the primary goal of showing the relationship between well-known languages (Pascal, LISP) and Prolog. The potential Prolog user is urged to attempt programming directly in Prolog instead of following the above steps.

The query *append*(*cons*(*a*, *cons*(*b*, *nil*)), *cons*(*c*, *nil*), *Z*) yields

$$Z = cons(a, cons(b, cons(c, nil))).$$

In Edinburgh Prolog, the above query is stated as *append*([*a*, *b*], [*c*], *Z*), and the result becomes

$$Z = [a, b, c]$$

A remarkable difference between the original Pascal-like version and the Prolog version of *append* is the ability of the latter to determine (unknown) lists that, when appended, yield a given list as a result. For example, the query *append*(X, Y, [*a*]) yields

$$X = [\,] \quad Y = [a]$$
$$X = [a] \quad Y = [\,]$$

The above capability is due to the generality of the search and pattern matching mechanism of Prolog. An often asked question is, *Is the generality useful*? The answer is *definitely yes*! A few examples will provide supporting evidence.

The first appears in Warren [45] and is a procedure for determining a list *LLL*, which is the concatenation of *L* with *L* the result itself being again concatenated with *L*.

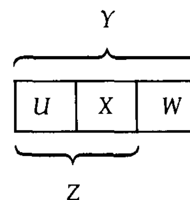
```
triple(L, LLL) :- append(L, LL, LLL), append(L, L, LL).
```

Note that the first *append* is executed even though *LL* has not yet been bound. This amounts to copying the list *L* and having the variable *LL* as its last element. After the second *append* is finished, *LL* is bound, and the list *LLL* becomes fully known. This property of postponing binding times can be very useful. For example, a dictionary may contain entries whose values are unknown. Identical entries will have values that are bound among themselves. If and when a value is actually determined, all communally unbound variables are bound to that value. This situation occurs, for example, in compiling, when several *gotos* are directed to the same forward label that has not yet been reached at a given stage of the processing.

Another interesting example is *sublist*(X, Y), which is true when X is a sublist of Y. Let *U* and *W* be the lists at the left and right of the sublist X. Then the program becomes

```
sublist(X, Y) :- append(Z, W, Y), append(U, X, Z),
```

where the variables represent the sublists indicated below:



An additional example is the bubblesort program credited to van Emden in [6]. The specification of two adjacent elements *A* and *B* in a list *L* is done by a call:

```
append(_, [A, B|_], L)
```

The underscore stands for a variable whose name is irrelevant to the computation, and the notation [*A*, *B*|C] stands for *cons*(*A*, *cons*(*B*, C)). (Note that the underscores correspond to different variables.) The rules to bubblesort then become

```
bsort(L, S) :- append(U, [A, B|X], L),
               B < A,
               append(U, [B, A|X], M),
               bsort(M, S),
bsort(L, L).
```

The first *append* generates all pairs of adjacent elements in *L*. The literal $B < A$ is a built-in predicate that tests whether *B* is lexicographically smaller than *A*. (Note that both *A* and *B* must be bound, otherwise a failure occurs! There will be more discussion of this limitation later in this article.) The second *append* reconstructs the modified list, which becomes the argu-

ment in a recursive call to *bsort*. If the first clause is no longer applicable, then all pairs of adjacent elements are in order, and the second clause then provides the desired result. This version of bubblesort is space and time inefficient since *U*, the initial segment of the list, is copied twice at each level of recursion.[2] However, the brevity of the program is indicative of the savings that can be accrued in programming and debugging.

In a final example also credited to van Emden [6], the program plays the classical game of Nim. A list such as

[*s*(*s*(*0*)), *s*(*0*), *s*(*s*(*s*(*0*)))]

represents a current list of piles of matches: *0* indicates no matches, *s*(*0*) one match, and so on. The objective of the game is to make a move—remove one or more matches from one of the piles—such that, after alternate moves the opponent is unable to remove any matches. The program is presented in Figure 3.

The clauses *us* and *move* state that a move should produce the list Y by selecting a pile *X1* from X, taking some matches from *X1* to form *X2*, and replacing *X1* with *X2* to form Y. The number of matches removed should be such that the opponent will be unable to win the game by removing matches from list Y. The procedure *not* can be considered as a built-in predicate that succeeds if its argument fails and vice versa. Each call of the procedure *takesome* attempts to remove one or more matches from the original stack of matches.

The program could be further simplified noting that

1. the procedure *them* is identical to *us* and can therefore be eliminated;
2. by using lists of lists such as [[*1*, . . . , *1*], [*1*, . . . , *1*], . . . , [*1*, . . . , *1*]], the procedure *takesome* could be replaced by a call to *append*(X2, [_|_], *X1*) that removes one or more *1*s from a given sublist.[3]

We urge readers to estimate the size of a corresponding program written in the language of their choice.

The generality and succinctness achieved in Prolog programs vis-à-vis those written in conventional languages is due to two factors:

1. the central role played by unification that combines the effect of both conditionals and assignments, and
2. the backtracking capabilities that allow the generation of multiple solutions to a given problem.

[2] Bsort could be modified to avoid this copying.
[3] We thank Randy Goebel for suggesting this variant.

```
us(X, Y):-          move(X, Y), not(them(Y, Z)).
them(X, Y):-        move(X, Y), not(us(Y, Z)).
move(X, Y):-        append(U,[X1|V], X),
                    takesome(X1, X2),
                    append(U,[X2|V], Y).

takesome(s(X), X).
takesome(s(X), Y):- takesome(X, Y).
```

FIGURE 3. A Program for Playing the Game of Nim

The price to be paid for having these advantages is the added time and space needed to perform unification and to simulate nondeterministic situations. It is conceivable that optimization methods can be developed that will scrutinize the unifications of a given program and will replace them by the simpler conditionals and assignments used in conventional languages. If this is achieved, the speeds of such Prolog programs will approach those of conventional languages. This is particularly true in the case of deterministic Prolog programs (see [30]).

UNIFICATION

Our previous definition of a ⟨*literal*⟩ is generalized to encompass labeled tree structures.

```
⟨literal⟩    ::= ⟨composite⟩
⟨composite⟩  ::= ⟨functor⟩ (⟨term⟩ {, ⟨term⟩})|
                 ⟨functor⟩
⟨functor⟩    ::= ⟨lower case identifiers⟩
⟨term⟩       ::= ⟨constant⟩|⟨variable⟩|⟨composite⟩
⟨constant⟩   ::= ⟨integers and lower case identifiers⟩
⟨variable⟩   ::= ⟨identifiers starting with
                 an upper case letter or _⟩
```

Examples of terms are *constant*, *Var*, *319*, *line*(*point*(X, 3), *point*(4, 3)). It is usual to refer to a single rule in a Prolog program as a clause. A Prolog procedure (or predicate) is defined as the set of rules whose head has the same ⟨*functor*⟩ and arity.

The objective now is to extend the capabilities of the function *match* in the procedure *solve* to handle these more general terms. This corresponds to the unification algorithm proposed by Robinson [28, 35, 36]. Terms are trees whose internal nodes are functors and whose children are either constants, variables, or terms. Unification tests whether two terms *T1* and *T2* can be matched by binding some of the variables in *T1* and *T2*. The simplest algorithm uses the rules summarized in Table I to match the terms. If both terms are composite and have the same functor, it recurses on their children. This algorithm only binds variables if it is *absolutely necessary*, so there may be variables that remain unbound.

It is not difficult to write a recursive function *unify*, which, given two terms, tests for the result of unification using the contents of Table I. For this purpose, one has to select a suitable data structure. In a sophisticated version, terms are represented by variable size records containing pointers to other records, to constants, or to variables.

A simpler data structure uses linked lists and the so-called Cambridge Polish notation. For example, the term *f*(X, *g*(Y, *c*)) is represented by (*f*(*var x*) (*g*(*var y*) (*const c*))), which can be constructed with *cons*'s. The unification of the two terms

f(X, *g*(Y), *T*)   and   *f*(*f*(*a*, *b*), *g*(*g*(*a*, *c*), *Z*))

succeeds with the following bindings:

TABLE I.

Terms 1↓, 2→	⟨constant⟩ C2	⟨variable⟩ X2	⟨composite⟩ T2
⟨constant⟩ C1	*succeed if* C1 = C2	*succeed with* X2 := C1	*fail*
⟨variable⟩ X1	*succeed with* X1 := C2	*succeed with* X1 := X2	*succeed with* X1 := T2
⟨composite⟩ T1	*fail*	*succeed with* X2 := T1	*succeed if* (1) T1 and T2 have the same functor and arity (2) the matching of corresponding children succeeds

$$X := f(a, b)$$
$$Y := g(a, c)$$
$$T := Z$$

Note that, if we had $g(Y, c)$ instead of $g(a, c)$ in the second term, the binding would have been

$$Y := g(Y, c).$$

Prolog interpreters would usually carry out this circular binding, but would soon get into difficulties since most implementations of the described unification algorithm cannot handle circular structures. Manipulating these structures (e.g., printing, copying) would result in an infinite loop unless the so-called *occur check* were incorporated to test for circularity. This is an expensive test: Unification is linear with the size of the terms unified, and incorporation of the *occur check* renders unification quadratic. More comments on circular structures will be made later.

Further machinery is needed to incorporate unification into the procedure *solve* of Figure 1. The more general procedure *solve* is presented in Figure 4. The function *unify* that replaces *match* has three parameters: the two terms to be tested for unification and the list *env* that contains the current bindings of the variables. The result of *unify* is either *nil*, if the unification fails, or, in the case of success, the new environment *newenv* obtained by appending the new bindings to the previous environment *env*. The variable *env* is a pointer to a linked list of pairs (*variable-binding*) corresponding to the sequence of bindings performed when unification succeeds. Since *newenv* is appended to the front of *env*, the proper environment is always available when backtracking occurs and a new call of *solve* is made.

The function *unify* is called with the following arguments: the head of a rule in the database, the head of the list of goals, and the current environment. Since Prolog rules are usually recursive, it is important to create new copies of the clauses in the database before the unification actually takes place. This copying can be expensive, but it may be avoided by using shared structures in which only the new variables are created, without having to copy the entire skeleton of the clause. A suitable numbering scheme for the newly created variables is to combine the variable's name with the current recursion level of the procedure *solve*. This explains why *copy* is called with the parameter *level*.

When success is reached, the environment is printed to reveal the current bindings. At that point, the user should have the ability to either abort the computation or request an attempt to find another solution.

It is our hope that the reader will have no difficulty in reconstructing this simple interpreter. A few additional hints are helpful. It is necessary to write a *lookup* function that traverses the environment list and finds the bindings of the variables. For example, if the environment list contains

$$((X_{123}Z_{110}) \ldots (Y_{112}f(a, X_{109})) \ldots (Z_{110}g(Z_{100})) \ldots (Z_{100}c))$$

the binding of X_{123} is $g(c)$. This lookup is performed within the unification algorithm and the *printenv* routine that prints the environment when success is reached.

To facilitate the reading and storing of rules, it is convenient to implement a syntax directed recursive descent translator based on the syntactic rules given in

```
procedure solve(L, env: pLIST; level: integer);
  begin local i: integer; newenv: pLIST;
    if L ≠ nil
      then
        for i := 1 to n do
        begin
          newenv := unify(copy(head(Rule[i]), level + 1),
                          head(L), env);
          if newenv ≠ nil then
            solve(append(copy(tail(Rule[i]), level + 1), tail(L)),
              newenv, level + 1)
        end
      else printenv(env)
end;
```

FIGURE 4. A Final Version of the Interpreter

this article. Other papers [2] present different descriptions of interpreters, written in LISP, and other languages.

The extensive updating of linked data structures inevitably leads to unreferenced structures that can be recovered by a garbage collection. Already in the primitive interpreter of Figure 4, the successive *appends* in the recursive call of *solve* indirectly produce unreferenced cells that can be explicitly returned to the allocator by the programmer. Some of the existing interpreters do perform garbage collection with the classical phases of marking, collecting, and compacting. Nevertheless, the recovery of cells in the environment list can become more complex, especially when "structure sharing" is used [1, 44].

COMPILATION

This section will show the reader how Prolog programs can be compiled into a Pascal-like language. Each Prolog procedure (i.e., a sequence of clauses having the same functor and arity) will be compiled into a Pascal-like procedure. The availability of a language featuring *continuations* considerably eases the task of writing Prolog interpreters and compilers. The compilation of Prolog programs will be illustrated by describing the result of compiling the *append* program presented in the "Examples" section (page 1313). This presentation is based on a (hand-compiled) program by Michel van Caneghem of Marseilles, France.

The compiled procedures will have one more parameter than the original procedures. This parameter, *cont*, is a procedure call whose role is to specify a continuation (i.e., a sequence of commands to be executed just before exiting the basic procedure). It should be noted that most Pascal compilers do not allow procedure calls as parameters. This difficulty may be circumvented by selecting another language (see [29]) or by simulating the desired mechanism.

The procedure *unify(X, Y: pLIST;* **procedure call** *cont)* is similar to the one used by the interpreter of Figure 4; its role is to check whether the two terms X and Y are unifiable and, if so, to modify the current environment that is represented by a list pointed to by a global variable. The environment consists of a list of variables, their type (*bound*, *free*, or *reference*), and a pointer to a term if the variable is not free. The parameters X and Y of the procedure *unify* are one of three types:

1. an unbound variable, denoted by *free*;
2. a variable bound to a composite or constant term, denoted by *bound*;
3. a variable bound to another variable, denoted by *ref*.

A more detailed description of the steps performed by *unify* is

1. if X and Y are unifiable according to the specifications in Table I, then
   a. update the environment by incorporating the new list of bindings,
   b. call *cont*, and
   c. unbind the bindings done in (a) (this step is only performed in the backtrack mode);
2. if X and Y are not unifiable, simply exit.

Once *unify* is available, the compiled code for *append* follows readily from the version preceding the final one in the "Examples" section. The equal sign is made to correspond to an explicit call to *unify*. The programs for *append* and *unify* appear in Figure 5. These procedures call on several simple auxiliary procedures, which will now be described.

The auxiliary procedure *newvar(V)* creates a new variable *V* of type *free* in the current environment (say,

```
procedure append(L1, L2, L3: pLIST; procedure call cont);
 begin
   A: unify(L1, nil, unify(L2, L3, cont));
        {let L1 = cons(H1, T1) and L3 = cons(H1, T)}
   B: newvar(H1); newvar(T1); newvar(T);
        {These variables are created with type free}
      unify(L1, cons(H1, T1),
        unify(L3, cons(H1, T), append(T1, L2, T, cont)))
 end;
procedure unify(X, Y: pLIST; procedure call cont);
 begin var U, V: pLIST;
   U := lookup(X);
   V := lookup(Y);
   if constant(U) and constant(V) then
   begin
     if U = V then cont
   end
   else
   if unbound(U) and unbound(V) then
   begin {bind(U, V)}
     assign(U, V, ref);
     cont;
     {control reaches this point after backtracking}
     {and binding has to be undone}
     unbind(U)
   end
   else
   if unbound(U) then
   begin{V is a composite term}
     assign(U, V, bound);
     cont;
     {control reaches this point after backtracking}
     {and binding has to be undone}
     unbind(U)
   end
   else
   if unbound(V) then
   {U is a composite term}
     {do as above but substituting U by V and vice versa}
     ...
   else {Both U and V are composite terms}
   unify(head(U), head(V), unify(tail(U), tail(V), cont))
end;
```

FIGURE 5. A Compiled Version of *append* and *unify*

by considering a next integer of a sequence). This corresponds to the copying done in the procedure *solve* of Figure 4. However, only the variables are copied. The function *lookup* follows the chain of linked variables having the attribute *ref* that had been previously set by the procedure *assign*. Once a link to a *bound* composite term (or to a constant) is found, *lookup* takes its value. The procedure *assign* binds two variables with the link type specified as the last parameter; *unbind*(*V*) sets the link type of *V* back to *free*; *unbound*(*U*) tests if *U* is of the type *free*; similarly, *constant*(*U*) tests whether *U* is a constant. Note that the environment list is now manipulated by the procedures *assign*, *unbind*, and *newvar*.

The program in Figure 5 can be compiled into a P-code-like sequence of statements. This is basically the approach used by Warren [44, 46]. Several optimizations are possible; for example, Warren's compiler for the PDP-10 produces optimized code by requiring the user to specify the nature of the parameters (input or output or both). In the case of *append*, if the lists *L1* and *L2* are input parameters and *L3* is an output parameter, the efficiency of the generated code approaches that of the LISP-like version presented earlier. Note that a simple optimization could speed up the execution of *append* in Figure 5. A case statement could transfer control to the label *A* or *B* upon a test of the type of parameter *L1* (either *nil* or a list).

Another optimization consists of avoiding the main **for**-loop of *solve* in Figure 4 by using a switch that directs the search to the clause or clauses for which *unify* has a chance of succeeding.

Symbolic execution (also called macroexpansion) uses *unify* at compile time to increase the efficiency of run-time computations. For example, the clauses

```
p(X, Y) :- q(X), z(Y).
q(a).
```

can be replaced by[4]

```
p(a, Y) :- q(a), z(Y).
q(a).
```

Prolog compilers can be written in Prolog and are therefore prime candidates for self-compilation and optimization. Warren's Prolog compiler for the PDP-10 was written in Prolog and so is the one developed by Robert Kong and the author at Brandeis University. It produces a C program similar to the one presented in Figure 5 except that it simulates the mechanism of continuation.

The speed of a compiler or interpreter is often measured in LIPS (logical inferences per second) where one logical inference corresponds to a successful unification. This is a meaningful measure only if all unifications require a bounded amount of time, which is often the case. The present VAX interpreters operate at 300–1000 LIPS. Recently announced compilers for workstations claim to produce programs operating at around 20,000 LIPS.

[4] Care should be exercised when performing this optimization especially in programs containing the cuts explained in the following section.

"IMPURE" FEATURES

In the previous sections, the so-called "pure" features of Prolog were described—namely, those that conform with the logic interpretation. These features include any relation defined by a (possibly infinite) list of assertions. A few "impure" features have been added to the language to make its use more practical. (This situation parallels the introduction of *setq*, *rplaca*, and other impure LISP functions). However, a word of warning is in order. Some of these impure features vary from one implementation to another.

The most prominent of these is the *cut*, represented by an exclamation point. Its purpose is to let the programmer change the search control mechanism embodied by the procedure *solve* in Figure 4. Let us reconsider the example on page 1312. Assuming that

```
a :- b, !, c, d.
```

then in the forward mode, *b*, *c*, and *d* would, as before, be placed in the list of goals and matched with heads of clauses in the database. If, however, a goal following the *cut* fails (e.g., *c*, or *d*), then no further matching of clauses defining *a* or *b* would take place.

The *cut* can be readily incorporated into the nucleus of the interpreter in Figure 4. One way to implement it is to introduce an additional variable parameter to indicate the level at which backtracking should be resumed when a *cut* is encountered. Recall that the parameter *level* was used to obtain different variable names when recopying rules. Assume now that the list of goals contains pairs (*goal-level*) instead of simply *goals*. If the goal is a *cut*, then *solve* is recursively called with the elements following the *cut* in the goal list. However, upon returning from the recursive call, that is, after a failure, the elements of the goal list are successively bypassed until the desired level is encountered.

The *cut* is often used to increase the efficiency of programs and to prevent the consideration of alternate solutions. Prolog purists avoid the use of *cuts*. The predicate *dif* explained on page 1321, can be used to minimize the number of *cuts*.

Another useful built-in predicate is *fail*, which automatically triggers a failure. To implement it, one simply forbids its presence in the database. Other built-in predicates that can be easily introduced are the input-output commands *read* and *write*.

Once *cut* and *fail* are implemented, negation by failure is accomplished by the clauses

```
not(X) :- X, !, fail.
not(X).
```

which fails if X succeeds and vice versa.

The built-in predicates *assert* and *retract* are used to add or remove a clause from the database. These built-in predicates are not backtrackable. They can be incorporated without difficulty into the interpreter of Figure 4.

The assignment can be introduced using a binary infix operator such as *is*. For example, *Y is X + 1* is only valid if X has been bound to an integer in which case

the right-hand side is evaluated and unified with Y; otherwise the *is* fails. To have a fully backtrackable addition, one would have to use the successor procedure with the database containing

succ(0, 1). succ(1, 2). etc.

In that case, sums become backtrackable by using the predicates embodying Peano's axioms.

```
sum(0, X, X).
sum(X1, Y, Z1) :- succ(X, X1),
                  succ(Z, Z1),
                  sum(X, Y, Z).
```

The above can be useful when dealing with small integers.

EXTENSIONS

In this section, some of the extensions that have been and are presently being proposed to increase the capabilities or the efficiency of Prolog programs are summarized. The DCG section considers a special type of Prolog rule that has been incorporated into most Prolog interpreters. These rules have a proven record of usefulness in compiling and in natural-language processing. The other extensions can be classified according to the changes they introduce in our version of the procedure *solve* of Figure 4. The extension to handle infinite trees generalizes the *unify* function to consider circular structures as well as trees. Similarly, the use of disequalities, in addition to equalities, can also be viewed as an extension aimed at improving *unify*. The goal-freezing technique extends Prolog by changing the order in which goals are placed in the list of goals prior to recursively calling *solve*. The potential for parallel execution of Prolog programs, and extensions based on the use of more general clauses, for example, those containing more than one literal in the head of a clause, are discussed in the last two sections.

DCGs

We alluded earlier to the similarity between Prolog rules and context-free grammar rules. In 1975, Colmerauer showed that parsers written in Prolog could be generated automatically from a set of grammar rules. More important, the grammar rules could have attributes that would also be readily translated into Prolog clauses performing syntax directed translation in a manner akin to attribute grammars. Colmerauer called these attribute grammar rules "Metamorphosis Grammars" (or M-Grammars) [8]. Although his main interests were in natural-language processing, he presented, as an example, the use of M-Grammars in describing a compiler for a minilanguage.

A few years later, Pereira and Warren reintroduced a form of M-Grammar called DCGs (for Definite Clause Grammars), which specialize to context-free grammars [31]. Warren's article on compiling using Prolog [45], which has become required reading for anyone interested in compilers and Prolog, is based on Colmerauer's work on M-Grammars.

The ability to process DCGs has been incorporated in several Prolog interpreters. The rules have a syntax that is similar to that of Prolog rules defined earlier in this article.

⟨dcg rule⟩ ::= ⟨head⟩ --> ⟨tail⟩

where the rewriting arrow "-->" replaces the sign ":-". Prolog interpreters translate DCGs into Prolog rules (containing additional parameters) that are used in parsing the strings generated by the context-free rule defining the nonterminal *⟨head⟩* and having *⟨tail⟩* as the right-hand side. In the case of DCGs, [*⟨terminal⟩*] is a valid term appearing in a tail. It denotes an element of the terminal vocabulary. An example will illustrate the use of DCGs. A Prolog parser using the grammar rules

$$S \rightarrow a\ S\ b$$
$$S \rightarrow c$$

is given by the DCG

```
s--> [a], s, [b].
s --> [c].
```

If we wished to determine the value of n for an input string of the form $a^n c b^n$, we would introduce an attribute that counts n using the term *succ* (for successor). The DCG becomes

```
s(succ(X)) --> [a], s(X), [b].
s(0) --> [c].
```

This yields *succ(succ(0))* when parsing the string *aacbb*. The ability of Prolog to have parameters that are both input and output enables us to generate *aacbb* when n is given as *succ(succ(0))*.

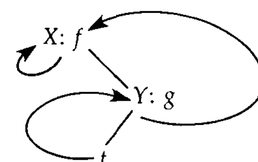
An additional feature of DCGs is that Prolog literals (enclosed within curly braces) can be interspersed in the right-hand side of grammar rules thus allowing actions to be executed while parsing is being done.

Infinite Trees

At the beginning of this article, we mentioned that most Prolog interpreters do not prevent the construction of circular structures; since the cost of this prevention is high (see *occur check*), Colmerauer has advocated the use of more general unification capable of dealing with terms containing variables that can be bound to subterms of the given term. These circular structures are called infinite trees and have an interesting property: They can be decomposed into a finite set of (infinite) subtrees. Assuming that the unit clause *eq*(Z, Z) is in the database, the query

eq(X, *f*(X, Y)), *eq*(Y, *g*(*t*(Y), X)).

produces the infinite tree



Colmerauer's view is that the unification algorithm overhead needed to handle these structures is smaller than that required by the *occur* check and, more important, infinite trees are natural representations for graphs, grammars, and flowcharts and therefore should not be avoided. The theoretical foundations for his generalized unification appear in [9]. His paper also presents examples of the use of infinite trees in the synthesis and minimization of finite state automatons. Giannesini and the author have utilized infinite trees in parser generation [17]. The use of the extended unification algorithm requires that a special printing program be available to output infinite trees [33]. An interesting characteristic of this program is that it can eliminate common subtrees thus producing a minimal infinite tree. For example, the minimal tree corresponding to *eq*(X, *f*(*f*(*f*(X)))) is X = *f*(X). This capability of the printing routine is useful in determining minimal finite state automatons [9] and in optimizing code [17].

Goal Freezing

Another very useful extension often incorporated into interpreters is the notion of coroutining, or lazy evaluation. The built-in procedure *freeze*(X, *P*) tests whether the variable X has been bound. If so, *P* is executed; otherwise the pair (X, *P*) is placed in a "freezer." As soon as X becomes bound, *P* is placed at the head of the list of goals for immediate execution. We now show how *freeze* can be easily implemented by expressing the procedure *solve* in Prolog itself! Although this metalevel programming will of course considerably slow down the execution, this capability can and has been used for fast prototyping extensions to the language. The Prolog procedure *solve* uses the built-in predicate *clause*(*Goal*, *Tail*), which determines the (first) head of a Prolog rule that unifies with *Goal* and binds *Tail* to the tail of that rule. In the case of unit clauses, *Tail* is bound to the constant *true*. (It would not be difficult to incorporate this prédicate into the interpreter discussed earlier.) The metalevel interpreter becomes

```
solve(true).
solve([Goal|Restgoal]) :- solve(Goal), solve(Restgoal).
solve(Goal) :- clause(Goal, Tail), solve(Tail).
```

To incorporate *freeze*, two additional parameters are needed: the list representing a current *Freezer* and another list representing its modified counterpart, the *NewFreezer*. Both lists contain pairs (*variable-goal*) in which *variable* is an unbound (or frozen) variable and the *goal* is a term to be activated as a procedure as soon as the variable becomes bound (or unfrozen). Immediately after *clause* matches a *Goal* in the database (i.e., after unification occurs), the *defrost* predicate is used to check whether any variable has thawed, in which case the corresponding procedure is immediately *solve*d after updating the *Freezer*. Details are given in Figure 6. The built-in predicates *var* and *nonvar* are used to check whether variables are bound.

In an efficient implementation of *freeze*, variables have an additional field that contains *nil* if the variable is bound; otherwise it contains a pointer to the goal that has to be executed when the variable becomes bound. The unification algorithm inspects the variable's field and, if applicable, triggers the goal execution. Also notice that, in the backtracking mode, thawed procedures should be refrozen. The predicate *freeze* has applications in tree and graph manipulation.

```
solve(true, Freezer, Freezer).
solve([Goal|Restgoal], Freezer, NewFreezer):-
                    solve(Goal, Freezer, TempFreezer),
                    solve(Restgoal, TempFreezer,
                      NewFreezer).
solve(Goal, Freezer, NewFreezer):-
                    clause(Goal, Tail),
                    defrost(Freezer, TempFreezer),
                    solve(Tail, TempFreezer, NewFreezer).
solve(freeze(X, Goal), Freezer, [[X|Goal]|Freezer]):-
                    var(X).
solve(freeze(X, Goal), Freezer, NewFreezer):-
                    nonvar(X),
                    solve(Goal, Freezer, NewFreezer).
defrost([ ], [ ]).
defrost([[X|Goal]|Freezer], [[X|Goal]|NewFreezer]):-
                    var(X),
                    defrost(Freezer, New Freezer).
defrost([[X|Goal]|Freezer], NewFreezer):-
                    nonvar(X),
                    defrost(Freezer, TempFreezer),
                    solve(Goal, TempFreezer, NewFreezer).
```

FIGURE 6. Steps in the Unification Algorithm

Disequalities

We begin here by showing an example of the use of *freeze* in "simulating" another useful Prolog extension: *dif*(X, Y) or X ≠ Y. This apparently innocuous built-in predicate is available in most interpreters; however, with the exception of the Prolog II interpreter [11], the predicate is applicable only when both X and Y are bound! If one or both of the variables are unbound, the interpreters precipitously enter the backtrack mode by assuming a failure. This is, of course, logically unsound. The more general predicate *dif* could be programmed using the clause

```
dif(X, Y) :- freeze(X, freeze(Y, different(X, Y))).
```

in which the built-in predicate *different*(X, Y) would test whether or not the bound variable X is different from the bound variable Y. Actually, the procedure *different* would have to be much more complex to achieve some of the generality of *dif* in Prolog II. Consider, for example, the query

```
dif(X, Y), X = f(a, B), Y = f(A, b).
```

The predicate *different* may prematurely fail if it does not check that X and Y are bound to terms containing new variables. The Prolog interpreter would have to ensure that *dif*(*A*, *a*) **or** *dif*(*B*, *b*). Furthermore, it should trigger a failure when processing the query

dif(X, Y), X = Y

Although the above could be done with special *freezes* [42], Colmerauer's approach is cleaner in the sense that its general unification incorporates lists of equalities and disequalities that are kept in the environment.

The availability of a general *dif* can render programs more efficient. For example, in coloring maps we may use the rules

```
color(red).    color(white).    color(blue).
validcolors(node(N1, C1), node(N2, C2)) :- dif(C1, C2),
                                           color(C1),
                                           color(C2).
```

The procedure *validcolors* asserts that two adjacent "nodes" must be colored differently. Note that *dif* is called when *C1* and *C2* are unbound and it is invoked as soon as *C1* and *C2* become bound.

In Colmerauer's approach, it is the generalized unification algorithm, capable of handling infinite trees, that is also capable of deciding whether the set of constraints is satisfiable or not. The backtracking mode is triggered only in the case of unsatisfiability. A logical extension of this philosophy is presently being pursued by the Marseilles' group. Why not make similar backtracking decisions for other inequalities (i.e., ≤ ≥ . . .)? Indeed, the simplex method is now being incorporated into a Prolog interpreter dealing with rational numbers and that is also able to decide the unsatisfiability of a system of linear inequations.

Parallelism

Whereas for most languages it is fairly difficult to write programs that automatically take advantage of operations and instructions that can be executed in parallel, Prolog offers an abundance of opportunities for parallelization. There are at least three possibilities for performing Prolog operations in parallel:

1. *Unification.* Since this is one of the most frequent operations in running Prolog programs, it would seem worthwhile to search for efficient parallel unification algorithms. Some work has already been done in this area [14, 47]. However, the results have not been encouraging.
2. *And-parallelism.* This consists of simultaneously executing each procedure in the tail of a clause. For example, in

   a(X, Y, U) :- *b*(X, Z), *c*(X, Y), *d*(T, U).

   an attempt is made to continue the execution in parallel for the clauses defining *b*, *c*, and *d*. The first two share the common variable *X*; therefore, if unification fails in one but not in the other, or if the unification yields different bindings, then some of the labor done in parallel is lost. However, the last clause in the tail can be executed independently since it does not share variables with the other two.
3. *Or-parallelism.* When a given predicate is defined by several rules, it is possible to attempt to apply the rules simultaneously [12].

A language called *Concurrent Prolog* has been proposed by Shapiro [39] and is being used to develop parallel algorithms in systems programming and in graph manipulations. It assumes that both And- and Or-parallelism are available.

Concurrent Prolog uses the special punctuation marks "?" and " | ". The question mark is a shorthand notation for *freezes*. For example, the literal *p*(X?, Y) can be viewed as a form of *freeze*(X, *p*(X, Y)).

The vertical bar is called *commit* and usually appears once in the tail of clauses defining a given procedure.[5] Consider, for example,

a :- *b*, *c*|*d*, *e*.
a :- *p*|*q*.

The literals *b*, *c*, and *d*, *e* are executed using And-parallelism. However, the computation using Or-parallelism for the two clauses defining *a* continues only with the clause that first reaches the *commit* sign. For example, if the computation of *b*, *c* proceeds faster than *p*, then the second clause is abandoned, and execution continues with *d*, *e* only.

Other Extensions

Many other extensions are constantly being proposed, and they range from polymorphic type checking to extending the generality of Prolog rules. In the logic interpretation of a Prolog program, we saw that

a :- *b*, *c*, *d*.

could be interpreted as the Boolean formula

$$b \land c \land d \rightarrow a$$

or

$$a \lor \bar{b} \lor \bar{c} \lor \bar{d}$$

The above formulas have just one positive literal and correspond to clauses having just one head. These are called Horn clauses, and Prolog's foundations rest on proving theorems that are expressible by Horn clauses [43]. Several researchers have suggested extensions for dealing with more general clauses [21] and for developing semantics for negation that are more general than that of "negation by failure" [27]. Experience has shown that the most general extension, that is, to the general predicate calculus, poses difficult combinatorial search problems. It is conceivable that other restricted forms of the predicate calculus may prove to be practical.

FINAL REMARKS

Prolog has not yet had time to fully mature. Yet, its youth should be viewed positively since it encourages experimentation. For this reason, efforts toward standardization have been considered premature by a fair number of its practitioners.

Prolog programs are usually significantly shorter than programs written in other languages (typically 5–10 times shorter). However, there have not yet been ex-

[5] If a commit sign does not appear in a rule, its implicit presence is assumed as the first element of the tail.

tremely large programs written in Prolog. To develop these large programs, one would need a special environment with adequate compilers, debuggers, type checkers, editors, profilers, etc. Prolog does not yet have such an environment, but the direction taken in developing LISP workstations should serve as a guide in producing Prolog counterparts.

The nonexistence of block structure, scoping, and type checking of variables may deter potential users from writing very large Prolog programs. Also lacking is a methodology for documentation, development of suitable notation for avoiding an ever-increasing number of parameters and for specifying parallelism. These, we believe, can only be achieved through experimentation. We also believe that there are a considerable number of people who are adept in the language, who, in due time, will provide solutions to these problems.

A unique property of some Prolog programs is their ability to perform inverse computations. For example, if $p(X, Y)$ defines a procedure p taking an input X and producing an output Y it can (in certain cases) determine which input X produces Y. Therefore, if p is a differentiation program it can also perform integration. This is easier said than done, since the use of impure features may result in operations that are not correctly backtrackable.

The inverse computation of parsing is string generation, and parsers are now available to perform both operations. This generation is only applicable when dealing with grammars for natural languages for which the generated sentences are small.

A compiler carefully written in Prolog can also be useful in decompiling. The difficulties encountered in doing the inverse operations are frequently due to the use of the *cut* and of assignments. It is likely that by using *dif* and by making certain assignments backtrackable, decompilation and other inverse computations would be attainable. Problems akin to inverse computation have also been programmed in Prolog. Among them is the synthesis of automatons given samples of input strings that they should accept or refuse [9, 38].

A fairly common classification of programming languages divides them into imperative, functional, and declarative. Most existing programming languages are imperative, pure LISP is functional, and pure Prolog is declarative. Since these types of languages have useful features, it is tempting to graft features of one of these types of language onto another. However, to achieve orthogonality this blending task may require considerable effort.

Summarized below are some of the features whose combination render Prolog unique among other languages.

1. Procedures may contain parameters that are both input and output;
2. procedures may return results containing unbound variables;
3. backtracking is built in, therefore allowing the determination of multiple solutions to a given problem;
4. general pattern matching capabilities operate in conjunction with a goal-seeking search mechanism;
5. program and data are presented in similar forms.

We believe, however, that the above listing of the features of Prolog does not fully convey the subjective advantages of the language. In our view, there are at least three such advantages:

1. Having its foundations in logic, Prolog encourages the programmer to describe problems in a logical manner that facilitates checking for correctness and consequently reduces the debugging effort.
2. The algorithms needed to interpret Prolog programs are particularly amenable to parallel processing.
3. The conciseness of Prolog programs, with the resulting decrease in development time, makes it an ideal language for prototyping.

Another important characteristic of Prolog that deserves extension, and is now being extended, is the ability to postpone variable bindings as much as is deemed necessary. Failure and backtracking should be triggered only when the interpreter is confronted with a logically unsatisfiable set of constraints. In this respect, Prolog's notion of variables will approach that used in mathematics.

The price to be paid for the advantages offered by the language amounts to the increasing demands for larger memories and faster CPUs. The history of programming-language evolution has demonstrated that, with the consistent trend toward less expensive and faster computers with larger memories, this price becomes not only acceptable but also advantageous since the savings achieved by program conciseness and by a reduced programming effort largely compensate for the space and execution time overheads. Furthermore, the quest for increased efficiency of Prolog programs will encourage new and important research in the areas of optimization and parallelism.

Acknowledgments. I owe my Prolog education to the members of GIA (Groupe d'Intelligence Artificielle) of Marseilles, France. It took a few summers there to convince me that the elegance and simplicity of Prolog could make a dramatic change in the craft of programming. The revelation I had learning Prolog was comparable to my initial encounter with recursion in LISP and also with my first attempts to code digital computers back in the late 1950s.

With minimal equipment, the members of GIA demonstrated the importance of the concepts they had created. My thanks go to Alain Colmerauer, Henri Kanoui, Francis Giannesini, Bob Pasero, and Michel van Caneghem for sharing their enthusiasm and knowledge of the new language. Thanks are also due to many GIA graduate students with whom I interacted. Among these I owe special thanks to Robert Kong, now at Brandeis University, who provided substantial help in the preparation of this article. Also at Brandeis University, I am fortunate to have a receptive group of co-

workers and students with whom I continue to learn and experiment with new features of Prolog. In particular, I wish to thank Mitchell Wand for his valuable comments and Tim Hickey for developing new techniques for using and implementing the predicate *freeze* and for participating actively in the multiple revisions of the article. Finally, I wish to express my gratitude to Randy Goebel of the University of Waterloo. He was an outstandingly helpful referee who provided numerous detailed suggestions for improving the original manuscript.

GUIDE TO CURRENT LITERATURE

Journals: *Journal of Logic Programming*, North-Holland, Amsterdam, (first issue—1984). *New Generation Computing*, Springer-Verlag, New York (first issue—1983).

Newsletter: Logic Programming Newsletter, Dept. of Informatics, Univ. of Lisbon, Portugal (first issue—1981).

Proceedings: See references [20], [21], [32], [40], and [41].

Books: See references [2]–[6], [15], [18], [19], [22], [24], [26], and [27].

Bibliography: See reference [34].

REFERENCES

References [3]–[5], [10], [15], [18], [19], [22], [24], and [26] are not cited in text.

1. Bruynooghe, M. A note on garbage-collection in Prolog interpreters. In *Implementation of Prolog*, Ellis Horwood Series in Artificial Intelligence. J.A. Campbell, Ed. Wiley, New York, 1984, pp. 258–267.
2. Campbell, J.A., Ed. *Implementation of Prolog*. Ellis Horwood Series in Artificial Intelligence. Wiley, New York, 1984.
3. Clark, K.L., and McCabe, F.G. *Micro-Prolog: Programming in Logic*. Prentice-Hall, Englewood Cliffs, N.J., 1984.
4. Clark, K.L., and Tarnlund, S.A., Eds. *Logic Programming*. Academic Press, New York, 1982.
5. Clocksin, W.F., and Mellish, C.S. *Programming in Prolog*. 2nd ed. Springer-Verlag, New York, 1984.
6. Coelho, H., Cotta, J.C., and Pereira, L.M. How to solve it in Prolog. Laboratorio Nacional de Engenharia Civil, Lisbon, Portugal, 1980.
7. Cohen, J. Nondeterministic algorithms. *Comput. Surv. 11*, 2 (June 1979), 79–94.
8. Colmerauer, A. Les grammaires de metamorphose. Groupe d'Intelligence Artificielle, Univ. of Marseilles-Luminy, France, 1975. (Appears as Metamorphosis grammars. In *Natural Language Communication with Computers*, L. Balc, Ed. Springer-Verlag, New York, 1978, pp. 133–189.)
9. Colmerauer, A. Prolog and infinite trees. In *Logic Programming*, K.L. Clark and S.A. Tarnlund, Eds. Academic Press, New York, 1982, pp. 231–251.
10. Colmerauer, A. Equations and inequalities on finite and infinite trees. Groupe d'Intelligence Artificielle, Université Aix-Marseille II, France, 1984.
11. Colmerauer, A., Kanoui, H., and van Caneghem, M. Prolog II. Groupe d'Intelligence Artificielle, University of Marseilles-Luminy, France, 1982.
12. Conery, J.S. The and/or process model for parallel interpretation of logic programs. Tech. Rep., Univ. of California, Irvine, May 1980.
13. Davis, R.E. Logic programming and Prolog: A tutorial. *IEEE Softw. 2*, 5 (Sept. 1985), 53–62.
14. Dwork, C., Kanellaris, P.C., and Mitchell, J.C. On the sequential nature of unification. *J. Logic Program. 1* (1984), 35–50.
15. Gallaire, H., and Minker, J., Eds. *Logic and Data Bases*. Plenum, New York, 1977.
16. Genesereth, M.R., and Ginsberg, M.L. Logic programming. *Commun. ACM 28*, 9 (Sept. 1985), 933–941.
17. Giannesini, F., and Cohen, J. Parser generation and grammar manipulations using Prolog's infinite trees. *J. Logic Program.* (Oct. 1984), 253–265.
18. Giannesini, F., Kanoui, H., Pasero, R., and van Caneghem, M. *Prolog* (in French). InterEditions, Paris, 1985 (a forthcoming English version will be published by Addison-Wesley, Reading, Mass.).
19. Hogger, C.J. *Introduction to Logic Programming*. Academic Press, New York, 1984.
20. IEEE. *Proceedings of the International Symposium on Logic Programming*. IEEE, Atlantic City, N.J., Feb. 1984.
21. IEEE. *Proceedings of the Symposium on Logic Programming*. IEEE, Boston, Mass., July 1985.
22. Kluzniak, F., and Szpakowicz, S. *Prolog for Programmers*. Academic Press, New York, 1985.
23. Kowalski, R.A. Algorithm = logic + control. *Commun. ACM 22*, 7 (July 1979), 424–436.
24. Kowalski, R.A. *Logic for Problem Solving*. Elsevier North-Holland, New York, 1979.
25. Kowalski, R.A. Logic as a computer language. In *Logic Programming*, K.L. Clark and S.A. Tarnlund, Eds. Academic Press, New York, 1982, pp. 3–16.
26. Li, D. *A Prolog Data Base System*. Research Studies Press, Wiley, New York, 1984.
27. Lloyd, J.W. *Foundations of Logic Programming*. Springer-Verlag, New York, 1984.
28. Martelli, A., and Montanari, U. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst. 4*, 2 (Apr. 1982), 258–282.
29. Mellish, C., and Hardy, S. Integrating Prolog in the poplog environment. In *Implementation of Prolog*, Ellis Horwood Series in Artificial Intelligence, J.A. Campbell, Ed. Wiley, New York, 1984.
30. Mellish, C.S. Some global optimizations for a Prolog compiler. *J. Logic Program. 2*, 1 (Apr. 1985), 43–66.
31. Pereira, F.C., and Warren, D.H. Definite clause grammars for language analysis. *Artif. Intell. 13* (1980), 231–278.
32. Pereira, L.M., et al. Eds. *Proceedings Logic Programming Workshop*. Algarve, Portugal, July 1983.
33. Pique, J.F. Drawing trees and their equations in Prolog. In *Proceedings of the Second International Logic Programming Conference, UPMAIL*. Univ. of Uppsala, Sweden, July 1984, pp. 23–33.
34. Poe, M.D., Nasr, R., Potter, J., and Slinn, J. A Kwic bibliography on Prolog and logic programming. *J. Logic Program. 1* (1984), 81–142.
35. Robinson, J.A. A machine-oriented logic based on the resolution principle. *J. ACM 12*, 1 (Jan. 1965), 23–41.
36. Robinson, J.A. Computational logic: The unification computation. *Mach. Intell. 6* (1971), 63–72.
37. Robinson, J.A. Logic programming—Past, present and future. *New Generation Comput. 1* (1983), 107–124.
38. Shapiro, E. Algorithmic program debugging. Ph.D. dissertation, Yale Univ., MIT Press, Cambridge, Mass., 1982.
39. Shapiro, E. Systems programming in concurrent Prolog. ICOT Tech. Rep., Nov. 1983.
40. Univ. of Uppsala. *Proceedings of the 2nd International Logic Programming Conference, UP-MAIL*. Univ. of Uppsala, Sweden, July 1984.
41. van Caneghem, M., Ed. *Proceedings of the 1st International Logic Programming Conference*. Faculté des Sciences de Luminy, Marseilles, France, Sept. 1982.
42. van Caneghem, M. L'anatomie de Prolog II (in French). Ph.D. dissertation, Université d'Aix-Marseille II, France, Oct. 1984.
43. van Emden, M.H., and Kowalski, R.A. The semantics of predicate logic as a programming language. *J. ACM 23*, 4 (Oct. 1976), 733–742.
44. Warren, D.H.D. Applied logic—Its use and implementation as a programming tool. Ph.D. dissertation, Univ. of Edinburgh, 1977 (also appeared as Tech. Note 290, SRI International, Menlo Park, Calif., 1983).
45. Warren, D.H.D. Logic programming and compiler writing. *Softw. Pract. Exper. 10* (Feb. 1980), 97–125.
46. Warren, D.H.D. An abstract Prolog instruction set. Tech. Note 309, SRI International, Menlo Park, Calif., Oct. 1983.
47. Yasuura, H. On the parallel computational complexity of unification. In *Proceedings of the International Conference on Fifth Generation Computing Systems* (Tokyo, Japan, Nov. 6–9), 1984, pp. 235–243.

CR Categories and Subject Descriptors: D.3.1; [**Programming Languages**]: Formal Definitions and Theory—*semantics, syntax*; D.3.2 [**Programming Languages**]: Language Classification—*very high-level languages, Prolog*; D.3.4 [**Programming Languages**]: Processors—*compilers, interpreters*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*logic programming, metatheory*; I.2.5 [**Artificial Intelligence**]: Programming Languages and Software

General Terms: Languages

Additional Key Words and Phrases: nondeterminism, unification

Author's Present Address: Jacques Cohen, Computer Science Dept., Ford Hall, Brandeis University, Waltham, MA 02254; CSNET address: jc.brandeis@csnet-relay.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.