Parsing and Compiling Using Prolog

JACQUES COHEN and TIMOTHY J. HICKEY
Brandeis University

This paper presents the material needed for exposing the reader to the advantages of using Prolog as a language for describing succinctly most of the algorithms needed in prototyping and implementing compilers or producing tools that facilitate this task. The available published material on the subject describes one particular approach in implementing compilers using Prolog. It consists of coupling actions to recursive descent parsers to produce syntax-trees which are subsequently utilized in guiding the generation of assembly language code. Although this remains a worthwhile approach, there is a host of possibilities for Prolog usage in compiler construction. The primary aim of this paper is to demonstrate the use of Prolog in parsing and compiling. A second, but equally important, goal of this paper is to show that Prolog is a labor-saving tool in prototyping and implementing many non-numerical algorithms which arise in compiling, and whose description using Prolog is not available in the literature. The paper discusses the use of unification and nondeterminism in compiler writing as well as means to bypass these (costly) features when they are deemed unnecessary. Topics covered include bottom-up and top-down parsers, syntax-directed translation, grammar properties, parser generation, code generation, and optimizations. Newly proposed features that are useful in compiler construction are also discussed. A knowledge of Prolog is assumed.

Categories and Subject Descriptors: D.1.0 [**Programming Techniques**]: General; D.2.m [**Software Engineering**]: Miscellaneous—*rapid prototyping*; D.3.4 [**Programming Languages**]: Processors; F.4.1. [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*logic programming* I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*logic programming*

General Terms: Algorithms, Languages, Theory, Verification

Additional Key Words and Phrases: Code generation, grammar properties, optimization, parsing

1. INTRODUCTION

The seminal paper by Alain Colmerauer on Metamorphosis Grammars first appeared in 1975 [9]. That paper spawned most of the developments in compiler writing using Prolog, a great many of them due to David H. D. Warren. Warren's thesis [30], the paper summarizing it [31], and the related work on Definite Clause Grammars [25] are practically the sole sources of reference on the subject.[1]

The available published material on the subject describes *one* particular approach in implementing compilers using Prolog. It consists of coupling actions

[1] A recent book edited by Campbell [3] mostly covers the implementation of Prolog itself.

This work was supported by the NSF under grant DCR 85-00881.
Authors' address: Computer Science Department, Ford Hall, Brandeis University, Waltham, MA 02254.
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.
© 1987 ACM 0164-0925/87/0400-0125 $00.75

to recursive descent parsers to produce syntax-trees which are subsequently utilized in guiding the generation of assembly language code. Although this remains a worthwhile approach, there is a host of possibilities for Prolog usage in compiler construction. The primary aim of this paper is to present the material needed for exposing the reader to the advantages of using Prolog in parsing and compiling. A second, but equally important, goal of this paper is to show that Prolog is a labor-saving tool in prototyping and implementing many non-numerical algorithms which arise in compiling, and whose description using Prolog is not available in the literature. Finally, a third goal is to present new approaches to compiler design which use proposed extensions to Prolog.

This paper is directed to compiler designers moderately familiar with Prolog, who wish to explore the advantages and present drawbacks of using this language for implementing language processors. The advantages of Prolog stem from two important features of the language.

(1) The use of unification as a general pattern-matching operation allowing procedure parameters (logical variables) to be both input and output or to remain unbound. Unification replaces the conditionals and assignments which exist in most languages.
(2) The ability to cope with nondeterministic situations, and therefore allow the determination of multiple solutions to a given problem.

From a subjective point of view, the main advantage of Prolog is that the language has its foundations in logic, and it therefore encourages the user to describe problems in a logical manner which facilitates the checking for correctness, enhances program readability, and reduces the debugging effort. It will be seen that unification and nondeterminism play an important role in compiler design; however, using their full generality is often costly and unnecessary. These issues are discussed throughout the paper whenever they become relevant. Remarks are made in the last section about the efficiency of Prolog-written compilers and the means to improve their performance.

The Prolog proficiency assumed in this paper can be acquired by reading the first few chapters of either Kowalski's [20] or Clocksin and Mellish's [6] books. In particular, the reader should be at ease with elementary list processing and with the predicate *append.* The concrete syntax used in this paper is that of Edinburgh Prolog [6]. It is also assumed that the reader is familiar with compiler design topics such as parsing, lexical analysis, code generation, optimizations, and so on. These topics are covered in standard texts [1, 17, 29].

2. PARSING

In this section we present parsers belonging to two main classes of parsing algorithms, namely, bottom-up and top-down. Due to the backtracking capabilities of Prolog, these parsers can in general handle nondeterministic and ambiguous languages. An early paper by Griffiths and Petrick [18] describes various parsing algorithms and their simulation by automata. There the amount of nondeterminism is roughly specified by a selectivity matrix which guides the parser in avoiding states leading to backtracking. A similar situation occurs in the Prolog parsers described here. In compilers, interest is commonly restricted to deterministic languages. Backtracking may be prevented by a judicious use of cuts(!) and/or by introducing assertions in the database that guide the parser in avoiding dead-ends.

A word about notation is in order. The grammar conventions are those in [1]. Edinburgh Prolog uses capital letters as variables, and therefore capitals cannot be used to represent nonterminals unless they are quoted. In this paper, the terms *t*( ) and *n*( ) denote, respectively, terminals and nonterminals. Quoting may be necessary for specifying certain terminals (e.g., parentheses). For example, the right-hand side (rhs) of the rule

$$F \rightarrow (E)$$

is described by the list

$$[t(`('), n(e), t(`)')].$$

Whenever stacks are used, they are also represented by lists whose leftmost element is the top of the stack.

This section does not pretend to make an exhaustive treatment of parsers. We describe bottom-up and top-down parsers for both nondeterministic and deterministic languages. A nondeterministic shift-reduce and a deterministic weak-precedence parser are the bottom-up representatives. Their top-down counterparts are, respectively, a predictive and an LL(1) parser. A recursive descent version of the latter is also considered. Besides those described herein, we have programmed and tested Earley's algorithm [13] and a parser generator that produces the necessary tables for parsing SLR(1) grammars [1].

2.1 Bottom-Up

A very simple (albeit inefficient) shift-reduce parser can be readily programmed in Prolog. Its action consists of attempting to reduce whenever possible; otherwise the window is shifted on to a stack and repeated reductions (followed by shifts) take place until the main nonterminal appears by itself in the top of the stack. Note that a reduction may be immediately followed by other reductions. A reduction corresponds to the recognition of a grammar rule; for instance, the reduction for the rule $E \rightarrow E + T$ occurs when $E + T$ lies on the top of the stack. It is then replaced by an E. This action is expressed by the unit clause

reduce$([n(t), t(+), n(e) \mid X], [n(e) \mid X])$.

Let us consider the classical grammar describing arithmetic expressions:

$$\begin{aligned} G_1\colon\ & E \rightarrow E + T \\ & E \rightarrow T \\ & T \rightarrow T^*F \\ & T \rightarrow F \\ & F \rightarrow (E) \\ & F \rightarrow \langle letter \rangle \end{aligned}$$

The appropriate sequence of *reduce* clauses follows immediately from the above rules. To decrease the amount of backtracking it is convenient to order these clauses so that rules with longer right-hand sides are tried before those with shorter rhs. We are now ready to present the parser. It has two parameters: (1) a list representing the string being parsed, and (2) the list representing the current stack.

```
% try-reduce %
sr_parse(Input, Stack) :- reduce(Stack, NewStack),
                          sr_parse(Input, NewStack).
% try_shift %
sr_parse([Window | Rest], Stack) :- sr_parse(Rest, [Window | Stack]).
```

Assume that a marker ($) is to be placed at the end of each input string. The following acceptance clause accepts a string only when the marker is in the window and the stack contains just an E.

```
% acceptance %
sr_parse([$], [n(e)]).
```

Consider the input string $a*b$. We assume that a scanner is available to translate it into the suitable list, understandable by the parser. Then the query

```
?- sr_parse([t(a), t(*), t(b), $], [ ]).
```

will succeed.

The above parser is very inefficient, since it relies heavily on backtracking to eventually accept or refuse a string. Note that in parsing the string $a*b$, $t(a)$ is first shifted and successively reduced to an F, T, and (even) an E; the latter being a faulty reduction. The parser is, however, capable of undoing these reductions through backtracking. This inordinate amount of backtracking can be controlled by a careful selection of the reductions and shifts that eliminate possible blind-alleys. This is done in our next bottom-up parser, which is the weak-precedence type [1, 19].

The basic strategy is to consult a table made of unit clauses like

try_reduce(Top_of_stack, Window). and *try_shift(Top_of_stack, Window).*

which command a reduction or a shift, depending on the elements lying on the window and on the top of the stack. The problem of automatically generating the above clauses from the grammar rules is addressed in Section 5. The weak-precedence relations for the grammar G_1 are represented by the clauses

```
try_reduce(n(t), $).
try_reduce(n(f), $).
...
try_reduce(t(')'), t(+)).
...
try_reduce(t(')'), t(')')).,  and
try_shift(t(+), t('(')).
try_shift(n(e), t(+)).
and so on.
```

We now transform the previous sr-parser into a wp-parser which takes advantage of the additional information to avoid backtracking. Using

Griffiths and Petrick's terminology [18], these unit clauses render the algorithms selective.

```
% acceptance %
wp_parse([$], [n(e)]).
% try_reduce %
wp_parse([W | Input], [S | Stack]) :– try_reduce(S, W),
  reduce([S | Stack], NewStack),
  wp-parse([W | Input], NewStack).
% try shift %
wp_parse([W | Input], [S | Stack]) :– try_shift(S, W),
  wp_parse(Input, [W, S | Stack]).
```

Notice that if a grammar is truly a weak-precedence grammar (i.e., there are no precedence conflicts and rules have distinct rhs), then backtracking will only occur when *try_reduce* fails and *try_shift* has to be tried. Thus the query

```
?- wp_parse(Input, [ ]), print(accept).
```

will print "accept", and succeed if the *Input* string is in the language. If the string is not in the language, the query will fail. The time complexity is proportional to the length of *Input*. Error detection and recovery are discussed in Section 9.

A comment about the efficiency of this version of *wp_parse* is in order. Since there will in general be a large number of *try_reduce* and *try_shift* rules, the execution time of the wp-parser could be significantly reduced if a Prolog compiler could branch directly to a clause having the appropriate constant as its first term (for example, by constructing a hashing table at compile time). Recent and planned Prolog optimizing compilers can indeed perform this branching [30]. The reader should also refer to [21] for a discussion of optimizations applicable to deterministic Prolog programs, which render their efficiency closer to those of conventional programs.

Finally, note that it would be straightforward to extend this type of parser to cover the syntactical analysis of bounded-context grammars, that is, those for which a decision to reduce or shift is based on an inspection of m elements in the top of the stack and a look-ahead of n elements in the input string.

2.2 Top-Down

A Prolog implementation of predictive parsers [1] follows readily from the programs described in the previous section. The grammar G_2, below, generates the same language as G_1, but left-recursion has been replaced by right-recursion.

$$\begin{aligned}
G_2:\; E &\rightarrow TE' \\
E' &\rightarrow + TE' \\
E' &\rightarrow \epsilon \\
T &\rightarrow FT' \\
T' &\rightarrow * FT' \\
T' &\rightarrow \epsilon \\
F &\rightarrow (E) \\
F &\rightarrow \langle letter \rangle
\end{aligned}$$

The above rules are placed in the database using the unit clauses

rule(Non_terminal, Rhs).

Examples are

```
rule(n(tprime), [t(*), n(f), n(tprime)]).
rule(n(tprime), [ ]).
```

The parser *predict(Input, Stack)* has the same parameters as its predecessors, namely: (1) the input string and (2) the current stack contents (initially *n(e)*, where *E* is the main nonterminal). The parser succeeds if the *Input* string is in the language, and fails otherwise.

The basic action of *predict* is to replace a nonterminal on the top of the parse stack by the rhs of the rule defining that nonterminal. If a terminal element lies on the top of the stack and if it matches the element *W* in the window, then parsing proceeds by popping *W* and considering the next element of the input string to be in the window. A string is accepted when the stack is empty and the window contains the marker. In Prolog we have

```
% acceptance.
predict([$], [ ]).
% try a possible rule.
predict(Input, [n(N) | Stack]) :-
  rule(n(N), Rhs),
  append(Rhs, Stack, NewStack),
  predict(Input, NewStack).
% match the terminals.
predict([t(W) | Input], [t(W) | Stack]) :- predict(Input, Stack).
```

The above parser can handle nondeterministic or even ambiguous grammars, but may become trapped in an infinite recursion loop if the grammar is left-recursive.

To improve the efficiency when processing deterministic grammars, one could again resort to placing additional information in the database. This is the case for the next parser we consider, which is applicable to LL(1) grammars, and does not rely on backtracking. It will become apparent in Section 5 that it is straightforward to generate tables for LL(1) grammars [1]. These tables have as entries the contents of the window *t(W)* and the nonterminal *n(N)* on the top of the stack, and they specify the appropriate (unique) replacement by the rhs of the rule defining *N*. Entries may be defined by unit clauses of the form

entry(t(W), n(N), Rhs).

for all pairs (*W*, *N*) such that $N \Rightarrow^* W \ldots$. An LL(1) deterministic parser is obtained by replacing the middle clause of *predict* by

```
predict[t(W) | Input], [n(N) | Stack]) :- entry(t(W), n(N), Rhs),
  append(Rhs, Stack, NewStack),
  predict([t(W) | Input], NewStack).
```

By properly selecting one among multiple entries, *predict* can deterministically parse languages defined by ambiguous grammars, as is the case of the **if then else** construct considered in [1, p. 191]. Moreover, the parser does not rely on backtracking to accept a string. The complexity of the LL(1) parser is therefore $O(n)$ where n is the length of the input string.

2.3 Recursive Descent

All of the previously described parsers contain a general nucleus which drives the parsing, the grammar rules being specified by unit clauses in the database. Parser efficiency can be increased by establishing a direct mapping between grammar rules and Prolog clauses. This is accomplished as in recursive descent parsing: each procedure directly corresponds to a given grammar rule. As usual, left-recursion is not allowed and has to be replaced by right-recursion to avoid endless loops.

There are three manners in which these parsers can be implemented in Prolog, depending on the form of the input string. The first and the least efficient of these is the one that uses the predicate *append*. The second uses links to define the input string that appears as unit clauses in the database. Finally, the third, which uses difference lists, is the most efficient, as will be seen by estimates of the various complexities. The implementation of these versions is illustrated using the grammar G_3, generating a^ncb^n, $n \geq 0$. The notation $t(T)$ and $n(N)$ will no longer be needed to differentiate between terminals and nonterminals, since the nonterminals will be transformed into Prolog procedures which manipulate terminal strings.

$$\begin{aligned} G_3\colon\ & S \rightarrow aSb \\ & S \rightarrow c \end{aligned}$$

Every grammar rule is transformed into a clause whose argument is the list of terminals derived from the defined nonterminal. Terminals are similarly handled using unit clauses. We have

```
s(ASB) :- append(A, SB, ASB),
  append(S, B, SB),
  a(A),
  s(S),
  b(B).
s(C) :- c(C).
a([a]).
b([b]).
c([c]).
```

The *append*s are used to partition the list ASB as the concatenation of three sublists A, S, B. Although the only partition for which the parser will succeed is

$$A = a, \quad S = a^{n-1}cb^{n-1}, \quad B = b,$$

this program will generate at least $2n$ incorrect partitions. Hence the number of calls needed to *append* is at least n^2. Note that the *append*s should precede the calls of $a(A)$, $s(S)$, $b(B)$. Otherwise, an infinite loop would occur. The above program can be optimized by symbolic execution: the terms $a(A)$, $b(B)$, and $c(C)$ can be directly replaced by their unit clause counterparts, yielding

```
s(ASB) :- append([a], SB, ASB),
  append(S, [b], SB),
  s(S).
s([c]).
```

The second approach for programming recursive descent parsers in Prolog is the use of links. An input string such as [a, a, c, b, b] is represented by the unit clauses *link*(i, t, $i + 1$), stating that there is a terminal t located between positions i and $i + 1$. In our case the input string *aacbb* becomes

```
link(1, a, 2).
link(2, a, 3).
link(3, c, 4).
link(4, b, 5).
link(5, b, 6).
```

A clause recognizing a nonterminal will now have two parameters denoting the leftmost and rightmost positions in the input string that will parse into the given nonterminal. In our particular example we have

```
s(X1, X4) :- link(X1, a, X2),
             s(X2, X3),
             link(X3, b, X4).
s(X1, X2) :- link(X1, c, X2).
```

The *a*s will be consumed by the n successive calls of the first two literals. Then, only the second clause is applicable and the c is consumed. Finally, the unbound variables $X3$, $X4$ are successively bound to the points separating the remaining *b*s. The algorithm's complexity is therefore linear.

An efficient implementation of recursive descent parsers in Prolog makes use of difference lists. If a nonterminal A generates a terminal string α (i.e., $A \Rightarrow^* \alpha$), that string can be represented by the difference of two lists U and V; V is a sublist of U which has the same tail as U. For example, if U is [a, c, b, b, b] and V is [b, b] the difference $U - V$ defines the list [a, c, b], which for G_3 parses into an S. Warren [31] points out that the use of difference lists corresponds to having the general link-like clause:

```
link([H | T], H, T)
```

which can be read as "the string position labelled by the list with head H and tail T is connected by a symbol H to the string position labelled T." A parser for G_3 using difference lists can be written as follows:

```
s(U, V) :- a(U, V1), s(V1, V2), b(V2, V).
s(W, Z) :- c(W, Z).
```

For the terminals a, b, and c we have

```
a([a | U1], U1).
b([b | U2], U2).
c([c | U3], U3).
```

Symbolic execution allows us to find the values of U and $V1$ in the first clause:

```
U = [a | U1],   V1 = U1
```

Similarly,

```
V2 = [b | U2],   V = U2
 W = [c | U3],   Z = U3
```

Substituting the values of the above variables, we obtain the optimized program

```
s([a | U1], U2) :- s(U1, [b | U2]).
s([c | U3], U3).
```

(The above program could also have been derived using symbolic execution by considering the first version of the parser with *append* and noticing that if $X - Y$ and $Y - Z$ are difference lists, then *append*$(X - Y, Y - Z, X - Z)$ is a fact.)

Let us follow the execution of the call

```
s([a, a, c, b, b], [ ]).
```

Notice that $U1$ becomes $[a, c, b, b]$ and $U2$ is [ ]. The next calls of S are

```
s([a, c, b, b], [b])
s([c, b, b], [b, b])
```

This last call matches only the second clause thus indicating a valid string. An informal English description of the acceptance is as follows: successively remove each a in the head of the first of the difference lists and add a b to the second one. A string is accepted when no more *a*s can be removed, the head of the first list is a c, and the two lists contain the same number of *b*s. Therefore, for this particular grammar, G_3, the parsing is done in linear time with no backtracking. The reader might have already surmised that the use of difference lists and of symbolic executions illustrated in this example could be carried out automatically from the given grammar rules. Clocksin and Mellish ([6, 1st ed., p. 237–238]) present a short Prolog program that does the translation.

3. SYNTAX-DIRECTED TRANSLATION

This type of translation consists of triggering semantic actions specified by the programmer, once selected syntactic constructs are found by a parser. In the case of the bottom-up parsers described in Section 2.1, it suffices to add a third parameter to the *reduce* clauses specifying the rule number and to modify the parser so that a semantic action (specified by the rule number) will take place just after the reduction. For example, in order to translate arithmetic expressions into postfix Polish notation, the corresponding reduce for the first rule of G_1 becomes

```
reduce([n(t), t(+), n(e) | X], [n(e) | X], 1).
```

The modified parser contains two additional parameters: (1) a stack, *Sem*, which will be manipulated by the *action* procedure and (2) a parameter, *Result*, which will be bound to the final result of the semantic actions:

```
% accept and bind Result to the semantic parameter.
wp_translate([$], [n(e)], Result, Result).
% try to perform a reduction and a semantic action.
wp_translate([W | Input], [S | Stack], Sem, Result) :-
  try_reduce(S, W),
  reduce([S | Stack], NewStack, RuleNumber),
  action(RuleNumber, [S | Stack], Sem, NewSem),
  wp_translate([W | Input], NewStack, NewSem, Result).
```

```
% try a shift.
wp_translate([W | Input], [S | Stack], Sem, Result) :-
  try_shift(S, W),
  wp_translate(Input, [W, S | Stack], Sem, Result).
```

The parser can then be equipped with actions by adding rules which specify how the temporary semantic parameter is to be modified for each rule. The following *action* procedure constructs parse trees for the arithmetic expressions defined by grammar G_1:

```
syntax_tree(Input, Tree) :- wp_translate(Input, [ ], [ ], Tree).
action(1, Stack, [X1, X2 | T], [plus(X2, X1) | T]).
action(3, Stack, [X1, X2 | T], [times(X2, X1) | T]).
action(6, [t(Letter) | Stack], Temp, [Letter | Temp]).
action(X, Stack, Temp, Temp) :- X ≠ 1, X ≠ 3, X ≠ 6.
```

The body of the last clause guarantees that no spurious actions are performed should backtracking ever occur. Notice that the *action* procedure must have access to the parsing stack (as is the case for rule 6) so that specific terminals may be incorporated into the actions. A similar strategy is applicable in adding actions to predictive parsers.

All of the above descriptions of semantic actions utilize inherited attributes and are admittedly standard. The main purpose of presenting them here is to point out how succinct the descriptions become when Prolog is used. The truly novel way of performing syntax-directed translation is that pioneered by Colmerauer and widely utilized by Warren. That approach does not strictly separate syntax from semantics as was done in this section. They have added new parameters to the recursive descent parser described in Section 2.3, so that the translation takes full advantage of the unification and goal-seeking features of Prolog. Colmerauer's approach is the subject of the next section.

4. M-GRAMMARS AND DCGs

A metamorphosis (or M-) grammar is a formalism which combines a Chomsky-type language definition with logic programming capabilities for manipulating the semantic attributes needed to perform syntax-directed translations. Colmerauer [9] maps general type-O Chomsky rules into general logic-programming clauses, (i.e., those that may contain more than one predicate in the left-hand side). A very useful subset of M-Grammars are Definite Clause Grammars (DCGs), which are based on Chomsky's context-free grammars. The reader has undoubtedly noticed the similarity between Prolog clauses and context-free grammar rules: they both have one term in the lhs and several (or none, i.e., ϵ) in the rhs. Prolog restricts itself to those special clauses called Horn or Definite clauses, thus explaining the acronym. It will be seen shortly that although DCGs are based on context-free grammars they are able to parse context-sensitive ones as well. (In fact, any recursively enumerable language can be recognized using DCGs with parameters.)

DCGs are translated directly into Prolog clauses which include a recursive descent parser using difference lists. For example, the DCG rules for recognizing strings in G_3 are

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

The syntax of DCGs is close to that of Prolog clauses. The ':–' is replaced by '-->', and terminals appear within square brackets. Most Prolog interpreters automatically translate the above into the clauses:

```
s([c | L0], L0).
s([a | L0], L1) :– s(L0, [b | L1]).
```

which have already been explained in Section 2.3. DCG terms usually contain one or more arguments which are directly copied into their Prolog counterparts, which also contain the difference list parameters. Our first example of usage of DCGs is to determine the value of n for a given input string a^ncb^n (generated by grammar G_3).

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

The added argument specifies that the recognition of a c implies a value of $N = 0$. Each time an s surrounded by an a and a b is recognized, the value of N increases by one (*succ* indicates the successor). The above DCGs are automatically translated into

```
s(0, [c | L0], L0).
s(succ(N), [a | L0], L1) :– s(N, L0, [b | L1]).
```

The call $s(X, [a, a, c, b, b], [\ ])$ yields $X = succ(succ(0))$. The backtracking capabilities of Prolog allow the call $s(succ(succ(0)), X, [\ ])$ which yields $X = [a, a, c, b, b]$.

By employing a technique similar to the one illustrated by the previous example, we can construct a parser s to recognize the language $a^nb^nc^n$. It uses the auxiliary procedure *sequence*(X, N) (defined below) which parses a list of Xs and binds N to the number of Xs found.

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

Let us now consider the use of DCGs for translating arithmetic expressions into their syntax-trees. We start with the simplified right-recursive grammar rules:

$$\begin{aligned} E &\rightarrow T + E \\ E &\rightarrow T \\ T &\rightarrow a \end{aligned}$$

Initially one would be tempted to use the DCG

```
e(plus(X, Y)) --> t(X), [+], e(Y).
e(X) --> t(X).
t(a) --> [a].
```

These rules, however, translate $a + a + a$ into *plus*$(a,$ *plus*$(a, a))$ which is right-associative, and therefore semantically incorrect. Some cunning is needed to circumvent this difficulty. Let us first rewrite the grammar rules as

$$\begin{aligned} E &\rightarrow TR \\ R &\rightarrow +\ TR \\ R &\rightarrow \epsilon \\ T &\rightarrow a \end{aligned}$$

Our goal is to generate *plus*(*plus*(a, a), a) for the input string $a + a + a$. The following DCG will do the proper translation:

```
expr(E) --> term(T1), restexpr(T1, E).
restexpr(T1, E) --> [+], term(T2), restexpr(plus(T1, T2), E).
restexpr(E, E) --> [ ].
term(a) --> [a].
```

When the clause *expr* recognizes the first term $T1$ in the expression, it passes this term to the second clause, *restexpr*. If there is another term $T2$ following $T1$, then the composite term *plus*($T1$, $T2$) is constructed and recursively passed to *restexpr*. The first parameter of *restexpr* is used to build a left-recursive parse tree, which is finally transmitted back to *expr* by the third clause.

Unfortunately, the above "contortions" are needed if one insists on using a recursive descent parser to contruct a left-associative syntax tree. This particular Prolog technique has become a standard idiom among DCG writers. A way out of this predicament is to implement DCGs using bottom-up parsers. (This has been proposed in [28].) At present these capabilities are not generally available in existing Prolog interpreters and compilers.

It is straightforward to generalize the above translation by introducing multilevel grammar rules such as

$$\begin{aligned} E_i &\rightarrow T_iR_i \\ R_i &\rightarrow op_iT_iR_i \\ R_i &\rightarrow E_{i+1} \\ R_i &\rightarrow \epsilon \end{aligned}$$

with $1 \leq i \leq n$ and $E_{n+1} \rightarrow letter \mid (E_1)$, where i denotes the precedence of the operator op_i. The corresponding DCG contains i as a parameter, and could allow for redefining the priorities of the operators, therefore rendering the language extensible. This approach is used in the Edinburgh version of Prolog.

A very useful feature of DCGs is that parts of Prolog programs may appear in their right-hand sides. This is done by surrounding the desired Prolog predicates within curly brackets. Our next example illustrates the use of this feature to perform the translation of arithmetic expressions into postfix notation directly by a DCG that does not construct syntax trees. Our first example of this technique will output the postfix notation.

```
e --> t, r.
r --> [+], t, {write(+)}, r.
r --> [ ].
t --> [a], {write(a)}.
```

This procedure produces the postfix expression using side effects, and this technique can be a drawback. One solution to this problem is to use difference lists to simulate the *write* procedure. To each DCG clause corresponding to a nonterminal N we add difference list parameters representing the list of symbols that are output during the recognition of N. (These difference lists are in addition to those used in syntactic analysis). We have

```
e(F1, F3) --> t(F1, F2), r(F2, F3).
r(F1, F4) --> [+], t(F1, F2), {writefile(+, F2, F3)}, r(F3, F4).
r(F, F) --> [ ].
t(F1, F2) --> [a], {writefile(a, F1, F2)}.
```

The output is simulated by the procedure *writefile(Symbol, Pos, NewPos)* defined by the unit clause

```
writefile(X, [X | B], B).
```

The call *?- e(F, [ ], [a, +, a, +, a], [ ])* produces *F* = [*a*, *a*, +, *a*, +]. This example shows that difference lists can be used both to select parts of a list and to construct a list. It is not hard to write a program that automatically performs the translation from a DCG using *write* to a DCG using *writefile* and additional difference lists. In the remainder of the paper we use the procedure *write*, and leave to the interested reader the task of adding difference lists to avoid side effects.

The BNF of full-fledged programming languages can be readily transcribed into DCGs that translate source programs into syntax-trees which can then be either interpreted or used to generate code. We have tested the DCG needed to process the entire Pascal language by translating input programs into syntax-trees. The following program fragments illustrate this construction for parts of a mini-language. A **while** statement is defined by the DCG:

```
statement(while(Test, Do)) -->
  [while], test(Test), [do], statement(Do).
```

A *test* may be defined by

```
test(test(Op, E1, E2)) --> expr(E1),
  comp(Op),
  expr(E2).
comp(=) --> [=].
comp(⟨ ⟩) --> [⟨ ⟩].
etc . . .
```

The translation of statements into P-code-like instructions is also easily achieved. For example the statement **while** *T* **do** *S* can be directly translated into the sequence

```
L: code for test T
   jif(i.e, jump if false) to Exit
   S
   jump to L
Exit:
```

If labels are represented by terms of the form *label(L)* and the instructions by *instr(jif, L)* or *instr(jump, L)*, the translation is performed by the DCG:

```
statement([label(L), Test, S, instr(jump, L), label(Exit)]) -->
  [while], test(Test, Exit), [do], statement(S).
```

where

```
test([R1, R2, Op, instr(jif, Exit)], Exit) -->
  expr(E1, R1),
  comp(Op),
  expr(E2, R2).
```

This example illustrates the elegant use of Prolog's logical variables and unification in compiling. Each of the variables *L* and *Exit* occur twice in the generated code. When instantiated, each pair will be bound to the same actual value. This instantiation may occur at a later stage when the final program is assembled and storage is allocated. Even when using special compiler-writing tools such as YACC, the implementation of similar constructs requires lengthier programs since one has to keep track of locations that have to be updated when final addresses become known. Prolog's ability to postpone bindings is therefore of great value in compiling.

The advantage of using logical variables and delayed binding is also apparent in managing symbol tables. Consider the procedure *lookup*(*Identifier*, *Property*, *Dictionary*), in which *Dictionary* is a list containing the pairs ⟨identifier-property⟩; *lookup*'s behavior is similar to that of the procedure *member*(*E*, *L*) which tests if an element *E* is present or not in the list *L*. However, *lookup* adds the pair to the *Dictionary* if it has not been previously added. We have

```
lookup(I, P, [[I, P] | T] :- !.
lookup(I, P, [[I1, P1] | T] :- lookup(I, P, T).
```

If *lookup* is initially called with an uninstantiated variable, the first clause will create the new pair [*I*, *P*] as well as a new uninstantiated variable *T*. The cut is needed to prevent backtracking once the desired pair is found or is created. Consider now the sequence of calls to *lookup*:

```
lookup(a, X1, D), lookup(b, X2, D), lookup(a, X3, D).
```

The net effect of the above calls is to store the two pairs [*a*, *X*1] and [*b*, *X*2] in *D* and to bind *X*3 to *X*1. Later on, when *X*1 and *X*2 are instantiated, *X*3 will automatically be bound to the value of *X*1.

A similar approach is used in [31] to implement binary tables. In that paper, table lookup is done in the code-generation phase after constructing the syntax trees (see Section 7). If one wished to perform that operation while parsing, the DCG rule defining a factor could be

```
f(id(I, P), D) --> [ident(I)], {lookup([I, P], D)}.
```

where *ident*(*I*) is constructed in a previous scanning pass and the property *P* is determined while processing declarations. In this case *lookup* should be modified to handle semantic errors such as undeclared identifiers.

5. GRAMMAR PROPERTIES

This section makes extensive use of the built-in predicate *setof* which implicitly relies on the nondeterministic capabilities of Prolog. In our view the use of this and similar predicates in determining grammar properties is perfectly justifiable, since, in this context, efficiency plays a secondary role: grammar properties are usually determined only a few of times when generating the parser and, although it is important that the generated parser itself be efficient (and deterministic), longer generation times are usually tolerable.

We start by pointing out that it is easy to test whether a grammar is strictly weak-precedence or LL(1), provided one knows the sets *first*(N), *follow*(N), and *last*(N). The Prolog procedures for performing these tests follow the declarative definitions closely and appear at the end of this section. We first show how Prolog can be used to calculate these sets in the general case of context-free grammars which may contain left-recursive nonterminals and ϵ-rules.

We assume that the rules for a grammar are stored in the database by assertions like

rule(*RuleNum*, *n*(*A*), *Rhs*)

in which *RuleNum* is an integer number identifying a rule, *Rhs* is the list representing the right-hand side of the rule defining the nonterminal A. Recall that the elements of *Rhs* are identified by the terms of the form $t(T)$ and $n(N)$ representing terminals and nonterminals.

For each nonterminal N in the grammar, *first*(N) is the set of all (terminal or nonterminal) symbols V such that $N \Rightarrow^* V \ldots$. To calculate the set *first*(A) for a nonterminal A, we use the built-in procedure *setof* in conjunction with a procedure *first* which finds a single element of this set. Thus, we make the top-level call:

```
all_first(N, L) :- setof(X, first(N, [ ], X), L).
```

The procedure *first*(*Input*, *Stack*, *V*) has three parameters:

(1) an *Input* list representing a sequence α of terminals and/or nonterminals,
(2) a *Stack* of rule numbers which keeps track of the already considered rules,
(3) a terminal or nonterminal element V such that $\alpha \Rightarrow^* V \ldots$.

There are three ways in which a symbol T can be the first element of a sentential form derived from α: (1) it can be the first element of α, (2) it can be the first element of a sentential form obtained by rewriting the first element in α (which must be a nonterminal in this case), or (3) it can be the first element of a sentential form obtained by rewriting some of the initial nonterminals of α into the empty string ϵ. The following procedure contains a clause for each of these three cases. The middle parameter *Stack* is used to prevent looping by prohibiting the consideration of previously used rules. The third clause uses the procedure *reduces_to_epsilon* (defined below) to determine if a sequence of nonterminals rewrites into ϵ.

```
first([Symbol | Rest], Stack, Symbol).
first([n(N) | Rest], Stack, Symbol) :-
   rule(Number, n(N), Rhs),
   not(member(Number, Stack)),
   first(Rhs, [Number | Stack], Symbol).
first(List, Stack, Symbol) :-
   append(A, B, List),
   A ≠ [ ],
   reduces_to_epsilon(A),
   first(B, Stack, Symbol).
```

The predicate *reduces_to_epsilon*(A) will succeed if A represents a sequence α of nonterminals which rewrite into the empty string. If a sentential form reduces to epsilon, then it must consist entirely of nonterminals that reduce to epsilon. Moreover, if a nonterminal rewrites to epsilon, then there is a parse tree representing this reduction such that no branch of the parse tree contains more than one occurrence of any nonterminal. The translation of these two statements into Prolog is straightforward. The procedure *list_reduces_to_epsilon* asserts that a sequence of *nonterminals List* rewrites into epsilon if each of the nonterminals does, and the procedure *nt_reduces_to_epsilon* asserts that a nonterminal N reduces to epsilon if it rewrites into a sentential form that reduces to epsilon. The stack parameter is used to guarantee that no branch of the parse tree contains multiple occurrences of any nonterminal.

```
reduces_to_epsilon(List) :-
  list_reduces_to_epsilon(List, [ ]).
list_reduces_to_epsilon([ ], Stack).
list_reduces_to_epsilon([n(N) | Rest], Stack) :-
  nt_reduces_to_epsilon(n(N), [n(N) | Stack]),
  list_reduces_to_epsilon(Rest, Stack).
nt_reduces_to_epsilon(n(N), Stack) :-
  rule(Number, n(N), Rhs),
  not(intersect(Rhs, Stack)),
  list_reduces_to_epsilon(Rhs, Stack).
intersect(List1, List2) :- member(X, List1), member(X, List2).
```

In weak-precedence, parsing reductions are called for when $S > W$, where

(1) W is the terminal element in the window,
(2) S is the (terminal or nonterminal) element in the top of the stack,
(3) $S > W$ if there exists a grammar rule

$$Y \rightarrow \cdots X_1 X_2 \ldots,$$

where $X1 \Rightarrow^+ \cdots S$ and $W \in \mathit{first}(X_2)$.

Shifting occurs when $S < W$, that is, if there is a rule

$$Y \rightarrow \cdots S X_2 \ldots, \quad \text{where} \quad W \in \mathit{first}(X_2).$$

To determine whether a language is of the weak-precedence type and to construct the parsing tables, one needs to find for each nonterminal X the set *last*$^+(X)$, consisting of all terminals and nonterminals V such that $X \Rightarrow^+ \cdots V$. This can be done by finding the sets *first*(X) for the grammar that is obtained by reversing the right-hand sides of the rules in the original grammar. It is easy to define a procedure *first_rev* that finds the sets *first*(A) for the reversed grammar by modifying the procedure *first*. The procedure to compute *last*$^+(A)$ is then concisely expressed as follows:

```
last_plus(n(X), Z) :-
  rule(Number, n(X), Rhs),
  reverse(Rhs, RRhs),
  first_rev(RRhs, [ ], Z).
```

As before, the set *last*$^+(A)$ can then be found using the *setof* predicate:

```
all_last_plus(n(A), L) :- setof(X, last_plus(n(A), [ ], X), L).
```

The set *follow*(*N*) is also succinctly expressed in Prolog. There are two ways in which a symbol *V* can be in the set *follow*(*N*): (1) there is a rule $X \rightarrow \alpha N\beta$ such that $V \in$ *first*(β), or (2) there is a rule $X \rightarrow \alpha N\beta$ such that β rewrites into epsilon, and $V \in$ *follow*(*X*). The Prolog procedure for *follow* consists of two clauses closely paralleling these two cases. The middle parameter *Stack* is again used to prevent looping by prohibiting the multiple use of rules:

```
follow(n(N), Stack, Terminal) :-
   rule(Number, n(X), Rhs),
   not(member(Number, Stack)),
   append(A, [n(N) | B], Rhs),
   first(B, [ ], Terminal).
follow(n(N), Stack, Terminal) :-
   rule(Number, n(X), Rhs),
   not(member(Number, Stack)),
   append(A, [n(N) | B], Rhs),
   reduces_to_epsilon(B),
follow(n(X), [Number | Stack], Terminal).
```

The predicate *all_follow* below calculates the list of all *follow* symbols of a nonterminal *N*:

```
all_follow(N, L) :- setof(X, follow(N, [ ], X), L).
```

To assess the gains in program size and readability the reader may want to compare the above programs with the English description of *first* and *follow* in [1, p. 184] and with a Pascal version in [2]. As to efficiency, these programs could be significantly improved by using the *assert* procedure to memorize previously computed *first*s and *follow*s, thereby avoiding recomputation. (This technique, called memoization, has been considered in [24].)

The predicates *first* and *follow* and *last_plus* can be used to test for the LL(1) and weak-precedence grammar properties and to generate the parsing tables for each of these types of grammars. For example, the clauses of the *try_reduce* procedure can be computed by the following procedure:

```
generate_reduces(L) :-
   setof(try_reduce(X, Y), wp_greater(X, Y), L).
wp_greater(X, Y) :-
   rule(RuleNum, n(N), Rhs),
   append(Any1, [A, B | Any2], Rhs),
   last_plus(A, X),
   first([B], Y).
```

and the *try_shift* clauses can be generated in a similar manner. Once these clauses have been computed and stored in the database, the grammar can be tested for weak-precedence by the query:

```
not_weak_precedence :- try_reduce(S, W), try_shift(S, W).
not_weak_precedence :- rule(N, X, Rhs), rule(M, Y, Rhs), N ≠ M
```

The first clause tests for reduce-shift conflicts, which could easily be reported to the user for selecting the desired action. This choice enables the processing of ambiguous grammars. The second clause tests if two grammar rules have identical right-hand sides.

The procedures to generate LL(1) tables and to test whether a grammar is LL(1) can also be written concisely. The procedure that generates the tables consists of a call to *setof*, combined with a procedure to find the *firsts* and *follows* of the right-hand side of a rule:

```
generate_ll1_table(L) :-
   setof(entry(t(W), n(X), Rhs), first_of_rule(t(W), n(X), Rhs), L).
first_of_rule(W, N, Rhs) :-
   rule(Number, N, Rhs),
   first(Rhs, W).
first_of_rule(W, N, Rhs) :-
   rule(Number, N, Rhs),
   reduces_to_epsilon(Rhs),
   follow(N, W).
```

To test whether a language is LL(1) we must show that the table constructed above has no multiple entries. This can be done with a call to the procedure *not_ll1*, defined as follows:

```
not_ll1 :- entry(t(W), n(X), Rhs1), entry(t(W), n(X), Rhs2), Rhs1 ≠ Rhs2.
```

The generation of the unit clauses for weak-precedence and LL(1) parsing actually amounts to prototyping a parser generator. Additional discussion on this topic is given in Section 6. The predicates *first* and *last* can also be used to determine if a grammar contains a nonterminal that is left-recursive and also right-recursive. This is a commonly used test for attempting to detect ambiguity in context-free grammars.

There is a host of grammar properties and transformations that could be succinctly described in Prolog. A few that we have programmed are elimination of ϵ rules, general replacement of left-recursive rules by right-recursive ones, and reduction to Chomsky and standard normal forms. Other properties that seem likely candidates for description in Prolog are an attempt to determine if a grammar is LL(k) or LR(k), and the reduction of an LR(k) grammar to LR(1).

6. LEXICAL SCANNERS AND PARSER GENERATION

We first note that the syntax of regular expressions is quite similar to that of arithmetic expressions. The union (|) replaces the add operator and concatenation (represented by a blank or period) replaces the multiplication operator. The star operation may be represented by surrounding a starred sequence by curly brackets. The translation of a regular expression into its syntax-tree is performed either using DCGs (Section 4) or triggering the semantic actions described in Section 3. For examle, the expression $\{(a \mid b).c\}.d$ is translated into the tree: *conc*(*star*(*conc*(*union*(*a*, *b*), *c*)), *d*). We now present a recognizer accepting strings defined by a regular expression given by its syntax-tree. The first argument of the procedure *rec* is the syntax-tree, the other two are difference lists (as described in Section 4).

```
rec(L, [L | U], U) :- letter(L).

rec(star(X), U, V) :- rec(X, U, W), rec(star(X), W, V).
rec(star(X), U, U).

rec(union(X, Y), U, V) :- rec(X, U, V).
rec(union(X, Y), U, V) :- rec(Y, U, V).

rec(conc(X, Y), U, V) :- rec(X, U, W), rec(Y, W, V).
```

The above interpreter for regular expressions is admittedly inefficient, since it relies heavily on backtracking. Nevertheless, it might be suitable for fast prototyping. An efficient version of the recognizer may be obtained in three steps:

(1) translation of regular expressions into a nondeterministic automaton containing ϵ moves;
(2) reduction of the automaton in (1) to a deterministic one not containing ϵ moves (in the cases where the empty symbol is in the language, a complete elimination of ϵ moves is not possible);
(3) minimization of the automaton obtained in (2).

The above steps are those performed by LEX, a scanner-generator package developed at Bell Labs. A Brandeis student, Peter Appel, has prototyped a Prolog version of LEX in less than one month. His program can handle practically all features of LEX, but is admittedly slow compared with the original C-version of that package. When compiled, his program can generate a scanner for a mini-language similar to that in the appendix of [1] in about four minutes. However, it should be noted that Appel's program is considerably (about five times) shorter than the C-counterpart, and it took a fairly short time to develop. Since the Prolog programs are deterministic, further gains in efficiency could be expected by applying the optimizations suggested by Mellish [21]. In Section 9 we briefly describe an alternate approach to scanner-generation using proposed extensions of Prolog.

In the remainder of this section we sketch two approaches for prototyping parser generators. The first generates recursive descent parsers, whereas the second produces SLR(1) parsers of the type used in YACC [1].

The recognizer of regular expressions presented earlier in this section can be easily modified to recognize context-free languages specified by rules whose right-hand sides are themselves regular expressions. For example the rule

$$E \rightarrow T\{+T\}$$

can be described by the unit clause

rule(*n*(*e*), *conc*(*n*(*t*), *star*(*conc*(*t*(+), *n*(*t*))))).

The new clause for *rec* becomes

rec(*n*(*A*), *U*, *V*) :– *rule*(*n*(*A*), *R*), *rec*(*R*, *U*, *V*).

It is straightforward to prototype a parser generator by implementing the following steps.

(a) Determine manually the syntax-trees for a grammar B specifying the syntax of the grammar rules themselves. Each nonterminal N has its corresponding syntax-tree τ_N asserted in the database by *rule*(N, τ_N).
(b) Use the modified recognizer *rec* to parse strings of B, that is, a set of grammar rules specifying a context-free grammar G.
(c) Attach actions to *rec* so that it produces the syntax-trees for the grammar G being read. This step has been described in Section 3.
(d) Once the trees for G are generated, *rec* itself can be reused to parse the strings generated by G.

A detailed description of the above steps appears in [14]. A further advantage of this approach is the possibility it offers to generate efficient recursive descent parsers [7]. One may "compile" assembly language code for a parser by "walking" on the syntax-tree of a grammar G.

Another option for parser generation is to use Prolog for producing the tables for SLR(1) parsers, given a set of grammar rules. An *item* of a grammar G is a production of G with a dot at some position of the right-hand side. Each *item* can be computed as a triplet (N, D, L) in which N is the rule number, D the dot position, also an integer, and L is the length of the right-hand side. (One could also have used only N and D and recomputed L for each rule N whenever needed.) States are implemented as lists of triplets. The main procedure generates all new transition states stemming from a given state. Termination occurs when no new states are generated. Ancillary procedures are needed to check if the element preceding a dot in a triplet is a nonterminal, or to test if an item has the dot at the end of a rule. This latter check is readily achieved by testing for (N, L, L). Another auxiliary procedure determines all triplets that should be added to a given state once it is found that that state contains items having a dot preceding a nonterminal.

The predicate *follow* (see Section 5) is called to determine the expected window contents that trigger reductions. These correspond to states containing items ending with a dot. As in YACC, the parser generator can produce tables with multiple entries, allowing the user to select the appropriate entry which renders the parsing deterministic.

A Prolog version of YACC has been prototyped at Brandeis by Cindy Lurie. Her program was developed in a couple of months. In addition to generating a parser, it also produces the code embodying the error-detection and recovery capabilities suggested by Mickunas and Modry [22]; the correction costs being interactively supplied by the user (see Section 9). The performance of the Prolog version of YACC is comparable to that of the LEX counterpart. The previous remarks about the efficiency remain applicable. A word is in order about the generated scanners and parsers. They are C-programs which, when optimized, can approach the efficiency of those generated by LEX and YACC.

7. CODE GENERATION

7.1 Generating Code from Polish

We start by describing a simple program that generates code for a single register computer having the usual arithmetic operations, as well as the *LOAD Var* and *STORE Var* instructions, where *Var* is the location of a variable. The DCG for performing the translation is basically that used to generate the postfixed Polish described in Section 4. The algorithm essentially operates as follows:

(1) When a variable is recognized it is placed on a stack.
(2) When an operation is recognized its two operands are on the top of the stack. If these are variables the following instructions are generated:

LOAD	*1st operand*
Operation	*2nd operand*

Step (2) is continued by replacing the top elements of the stack with the mark *acc* to indicate that, at execution time, the result will be in the accumulator. To take into account this mark we introduce the revised versions of (1) and (2), which handle the cases where one of the operands is an *acc* mark.

(1a) Before pushing a variable onto the stack it is necessary to check if the mark *acc* occupies the position just below the top of the stack. This indicates the need of a temporary storage, since the accumulator was already utilized in a previous operation and it contains a result that should not be destroyed. Thus the mark *acc* is replaced by T_i, the ith element of a pool of temporary locations, and the following instruction is generated: *STO* T_i. It is then possible to push the recognized variable onto the stack.

(1b) If the penultimate element in the stack is not *acc*, the variable is simply pushed onto the stack.

As for operators, two cases need to be considered: one for commutative operations, the other for noncommutative ones. Let $S1$ be the top of the stack and $S2$ the element just below it.

(2a) If neither $S1$, nor $S2$ is an *acc*, then code is generated as in step (2) above.

(2b, c) For the commutative operations (addition and multiplication) it suffices to generate

Operation $S1$ *if* $S2$ is an *acc*, or to generate
Operation $S2$ if $S1$ is an *acc*.

(2d) Noncommutative operations (subtraction and division) will check if $S1$ is an *acc*, in which case the instruction *STO* T_i has to be generated and the stack updated with T_i instead of *acc*, as is done in (1a). The generation proceeds as indicated in (2). The case where $S2$ is an *acc* is processed as in the case of commutative operations (2b).

The above description can be easily summarized in Prolog. For presentation purposes, we assume that the arithmetic expression has been parsed into postfixed Polish notation. We also assume that variables are represented by terms of the form *v*(*Name*) and operators by terms *op*(*Op*). In an actual implementation the semantic actions described below would be triggered directly from the DCG rules.

The procedure *gen_code*(*Polish*, *Stack*, *Temps*) traverses the list *Polish* and outputs the code as soon as it is generated. We remind the reader that a program that produces output using *writes* can easily be modified so that it stores the output in a list and thereby avoids relying on side effects to generate results (see Section 4). The *gen_code* procedure is initiated with a call to the procedure *execute*, defined by

```
execute(L) :- gen_code(L, [ ], [0]).
```

where L is the input in postfix. The operators and operands in the list L trigger calls to the corresponding *operator* and *operand* clauses, which modify the *Stack* as described above, and may either remove or return a location from the list

Temps of available temporary locations:

```
gen_code([op(Op) | Rest], Stack, Temps) :-
  operator(Op, Stack, NewStack, Temps, NewTemps),
  gen_code(Rest, NewStack, NewTemps).
gen_code([v(X) | Rest], Stack, Temps) :-
  operand(X, Stack, NewStack, Temps, NewTemps),
  gen_code(Rest, NewStack, NewTemps).
gen_code([ ], AnyStack, AnyTemps).
```

The *operator* and *operand* clauses have five parameters:

(1) the variable (or operator) being examined,
(2, 3) the starting and resulting stack configurations,
(4, 5) the starting and resulting lists of available temporary locations.

The following remarks will help in understanding the semantic actions of the procedures. The program assumes the availability of an unlimited number of temporary locations which are reused whenever possible: a temporary is returned to its stack after emitting an instruction of the type *LOAD* T_i or *Op* T_i. The list of available temporary locations is initialized to contain only the location T_0. Whenever a new temporary is needed, it is taken from this list, and if the list contains only one element a new temporary is generated (see the second clause of *get_temp* below). The term $t(X)$ is used to represent a temporary location.

```
% Case (1a).
operand(X, [A, acc | Stack], [v(X), A, t(I) | Stack], Temps, NewTemps) :-
  get_temp(t(I), Temps, NewTemps),
  write(sto, t(I)).

% Case (1b).
operand(X, Stack, [v(X) | Stack], Temps, Temps).
```

The first clause of *operand* guarantees that the accumulator is always the first or second element of the stack, if it occurs at all. The other elements in the *Stack* are either temporaries or variables:

```
% Case (2b).
operator(Op, [A, acc | Stack], [acc | Stack], Temps, NewTemps) :-
  codeop(Op, Instruction, AnyOpType),
  gen_instr(Instruction, A, Temps, NewTemps).

% Case (2c).
operator(Op, [acc, A | Stack], [acc | Stack], Temps, NewTemps) :-
  codeop(Op, Instruction, commute),
  gen_instr(Instruction, A, Temps, NewTemps).

% Case (2d).
operator(Op, [acc, A | Stack], [acc | Stack], Temps, NewTemps) :-
  codeop(Op, Instruction, noncommute),
  get_temp(t(I), Temps, Temps0),
  write(sto, t(I)),
  gen_instr(load, A, Temps0, Temps1),
  gen_instr(Instruction, t(I), Temps1, NewTemps).

% Case (2a).
operator(Op, [A, B | Stack], [acc | Stack], Temps, NewTemps) :-
  A ≠ acc, B ≠ acc,
  codeop(Op, Instruction, OpType),
  gen_instr(load, B, Temps, Temps1),
  gen_instr(Instruction, A, Temps1, NewTemps).
```

Notice that at most one of the clauses for *operator* can succeed, since there can be at most one *acc* in the stack. Thus the ordering of the clauses is unimportant, and there is no need for cuts.

The remainder of the program consists of a few auxiliary procedures. The procedure *get_temp* simulates the pop operation for a stack containing the currently available temporary locations. Temporary locations are returned to the stack by the first clause of *gen_instr*.

```
codeop(+, add, commute).
codeop(-, sub, noncommute).
codeop(*, mult, commute).
codeop(/, div, noncommute).
gen_temp(t(I), [I, J | R], [J | R]).
get_temp(t(I), [I], [J]) :-
  J is I + 1.
gen_instr(Instruction, t(I), Temps, [I | Temps]) :-
  write(Instruction, t(I)).
gen_instr(Instruction, v(A), Temps, Temps) :- write(Instruction, A).
```

The code generated for the expression $A^* (A^* B + C - C^* D)$ is

```
LOAD    A
MULT    B
ADD     C
STO     T0
LOAD    C
MULT    D
STO     T1
LOAD    T0
SUB     T1
MULT    A
```

An alternative approach to the method presented here is to generate new Prolog variables to represent the temporaries as they are needed and to ensure, in a subsequent pass, that their usage is optimized.

7.2 Generating Code from Trees

A more general approach to code generation is based on "walks" in the syntax-tree of a program. We start by describing Warren's approach [31] for generating code for a fictitious machine. This computer performs arithmetic operations using a single accumulator. The corresponding instructions are *ADD*, *MULT*, *SUB*, and *DIV*. Operations of the type *ADDI*, *MULTI*, and so on, are also available, and consider the value immediately following them as the second operand in the computation. *LOAD* and *STO* commands are of course present, as well as the unconditional transfer (*JUMP*) or conditional ones such as *J xx*, where *xx* is *EQ*, *NE*, *GT*, and so on. The input/output commands are simply *READ* and *WRITE*. The generator consists of the clause *encode_statement* which identifies the node of the syntax-tree and constructs the corresponding code. The generated code is a list of instructions and labels, (possibly containing embedded sublists), for instance,

[· · · *label(L1)*, [*instr(LOAD, X)*, *instr(ADDI*, 3)], · · ·]

In Warren's paper the arguments of instructions are stored in a dictionary, but remain unbound to actual memory addresses until the very final phase of the compiler. At that time an assembler determines the addresses of labels, and an allocator binds the addresses of the variables and reserves the number of memory locations needed to run the compiled program. We now present some fragments of Prolog programs that perform the generation. An assignment of an expression *Expr* to a variable *X* is translated into the list whose head is the generated code for the expression followed by the instruction *STO X*. The procedure *encode_statement* has three arguments: the syntax-tree, the dictionary *Dict*, and the resulting code. We have

```
encode_statement(assign(name(X), Expr),
                 Dict,
                 [Exprcode, instr(sto, Addr)]) :-
                 lookup(X, Addr, Dict),
                 encode_expr(Expr, Dict, Exprcode).
```

The procedure *lookup* stores the new variable *X* if it is not yet entered in *Dict* and retrieves the unbound variable representing its address (see Section 4).

The procedure *encode_expr* can handle two shapes of arithmetic expression syntax-trees. In the first the right operand is a leaf (i.e., a variable or a constant). In the second the right operand is a subtree. The syntax-tree for arithmetic expressions has internal nodes labeled by the operator *Op*. The more complex case where the right operand is a subtree is presented below. Its action is to translate *expr(Op, Expr*1, *Expr*2) (in which *Expr*2 is of the form *expr(Op, Any*1, *Any*2) into the sequence containing

(1) the code for *Expr*2,
(2) the instruction *STO temp*,
(3) the code for *Expr*1, and finally,
(4) the code for the instruction specified by *Op*.

An added argument *N* is needed to specify the pool of temporary locations. Its initial value is zero. In Prolog we have

```
encode-subexpr(expr(Op, Expr1, Expr2), N, Dict,
               [Expr2code, instr(sto, Addr), Expr1code, instr(Opcode, Addr)]) :-
  complex(Expr2),
  lookup(N, Addr, Dict),
  encode_subexpr(Expr2, N, Dict, Expr2code),
  N1 is N + 1,
  encode_subexpr(Expr1, N1, Dict, Expr1code),
  memoryop(Op, Opcode).
complex(expr(Op, Any1, Any2)).
memoryop(+, add).
memoryop(*, mult).
```

The code generated for the previous expression $A^* (A^* B + C - C^* D)$ now becomes

```
LOAD   C
MULT   D
STO    T0
LOAD   A
MULT   B
ADD    C
SUB    T0
STO    T0
LOAD   A
MULT   T0
```

Note that since a right subtree is evaluated before a left one, the code for $C^* D$ is the first to be generated.

The use of labels is illustrated by the generation of code for *while* statements. The translation consists of transforming the syntax-tree *while(Test, Do)* into the code

```
label(L1): ⟨encode Test⟩
           ⟨encode Do⟩
           jump L1
label(L2):
```

Note that a new argument ($L2$) is needed in the procedure that encodes tests to generate the jump to the exit label. The Prolog program to achieve the translation parallels the above description.

```
encode_statement(while(Test, Do), Dict,
[label(L1), Testcode, Docode, instr(jump, L1), label(L2)]) :-
  encode_test(Test, Dict, L2, Testcode),
  encode_statement(Do, Dict, Docode).
```

7.3 A Machine-Independent Algorithm for Code Generation

An alternate approach to code generation is that proposed by Glanville and Graham [15, 16]. It is assumed that by syntax-directed translation a source program is translated into its prefix Polish counterpart. A second syntax-directed translation of the prefix code then produces actual machine code. The interesting feature of this approach is that the grammar used to recognize the prefix takes into consideration the description of the machine for which code is generated. Consider a register machine whose operations are of the type

```
LOAD   M, R
ADD    R1, R2   or ADD M, R
STO    R, M
ADDI   C, R
```

where M is a memory address, C is a constant, and R is a register, and the first argument is the source, the second the destination. To simplify the presentation, we assume an unlimited pool of registers. The problem of dealing with a limited number of registers is discussed in the next section.

The grammar rule

$$R \rightarrow op\ R\ var \mid var$$

describes a prefix string in which the last operand is always a variable, (e.g., as in + + *a b c*). The code to be generated in this case can be triggered by semantic actions corresponding to the rules

$R \rightarrow var$
Action: Load variable into register *r*

$R \rightarrow op\ R\ var$
Action: 1. recognize (recursively) the left operand *R* assuming that it will use register *r*
2. generate the code: *op var r*

Similar grammar rules are applicable for generating code when the last operand is a constant. The more general case corresponds to the grammar rule

$$R \rightarrow op\ R\ R \mid var \mid const$$

In this case a new register is needed before recursing to the second *R*. Also, a register becomes available after recognizing the second *R*. A natural way of implementing the Glanville–Graham approach is through the use of DCGs. The following simplified grammar rules express assignments:

$$\begin{aligned} &A \rightarrow := var\ R \\ &R \rightarrow op\ R\ var \mid op\ R\ const \\ &R \rightarrow op\ R\ R \\ &R \rightarrow var \mid const \end{aligned}$$

Note that this is an ambiguous grammar, and therefore the use of cuts at the end of each clause is recommended to avoid generating multiple solutions. The recursive descent compiler generated from the DCGs opts, whenever possible, to the first rule defining *R*, instead of the more general second rule.

The procedures listed below specify the syntax-directed translation of prefix Polish into assembly language according to the above grammar rules. The procedure *reg* corresponds to the nonterminal *R* and has three parameters:

(1) generated assembly language sequence,
(2) register containing the final result,
(3) dictionary for storing variables.

Although the presented program assumes an unlimited number of registers, it is fairly straightforward to modify it to consider a finite number only. This can be done by adding extra parameters to the procedure *reg*.

The first two clauses of *reg* treat the special cases where the second parameter is a variable or a constant:

```
% Rule: R --> Op R var.
reg([S1, instr(Op, Addr, R1)], R1, D) -->
  arithop(Op, Optype),
  reg(S1, R1, D),
  [var(Var)],
  {lookup(Var, D, Addr), !}.
```

```
% Rule: R → Op R const.
reg([S1, instr(Constop, C, R1)], R1, D) -->
   arithop(Op, Optype),
   reg(S1, R1, D),
   [const(C)],
   {constop(Op, Constop), !}.
```

where *arithop* and *constop* are defined as

```
arithop(sub, noncommute) --> [-].   arithop(add, commute) --> [+].
arithop(div, noncommute) --> [/].   arithop(mult, commute) --> [*].
        constop(sub, subi).         constop(div, divi).
        constop(add, addi).         constop(mult, multi).
```

It is possible to perform some optimization in the case of commutative operations. For that purpose two additional DCG clauses are included to process the rules:

$$R \rightarrow op\ var\ R \quad \text{and} \quad R \rightarrow op\ const\ R$$

The DCG clause for the first of these rules is given below.

```
% Rule: R → op var R{op is commutative}.
reg([S1, instr(Op, Addr, R1)], R1, D) -->
   arithop(Op, commute),
   [var(Var)],
   reg(S1, R1, D),
   {lookup(Var, Addr, D), !}.
```

The more complex DCG given below corresponds to the rule $R \rightarrow op\ R\ R$.

```
% Rule: R → op R R.
reg([S1, S2, instr(Op, R2, R1)], R1, D) -->
   arithop(Op),
   reg(S1, R1, D),
   {R2 is R1 + 1},
   reg(S2, R2, D), {!}.
```

Two recursive calls are made to *reg* to determine the subsequences *S*1 and *S*2 representing the code for calculating the two operands. The simple DCG clauses for the rules $R \rightarrow var$ and $R \rightarrow const$ generate the necessary instructions that load a register with a variable or with a constant.

```
% Rule: R → var.
reg(instr(load, Addr, R1), R1, D) -->
   [var(Var)],
{lookup(Var, D, Addr), !}.
% Rule: R → const.
reg(instr(loadc, C, R1), R1, D) -->
   [const(C)], {!}.
```

Finally, we present the DCG clause for generating an assignment expressed in prefix by the rule

$$A \rightarrow := var\ R.$$

```
% Code generator for assignments.
instruction([S1, instr(store, 1, Addr)], D) -->
   [assign, var(Var)],
   reg(S1, 0, D),
   {lookup(Var, Addr, D)}.
```

Notice that the chosen grammar relies extensively on backtracking for recognizing the appropriate rule. For example, consider the two rules

$$R \dashrightarrow Op\ R\ const$$
$$R \dashrightarrow Op\ R\ var$$

and the input string (+ + 5 *c d*). Although the first rule will not apply, it will nonetheless be tried, and the code for the expression (+ 5 *c*) will be generated before backtracking. The same code will then have to be regenerated when the second rule is applied. This can be avoided by considering the following transformed equivalent grammar:

$$R \dashrightarrow Op\ R\ R2$$
$$R2 \dashrightarrow Var \mid Const$$

This transformation can be easily generalized to the case at hand, and the resulting parser will not rely on backtracking so there will be no need to insert cuts into the program.

An example of the code generated by this technique for the expression $A*(A*B + C - C*(D - E))$ is

```
LOAD    B, R0
MULT    A, R0
ADD     C, R0
LOAD    D, R1
SUB     E, R1
MULT    C, R1
SUB     R1, R0
MULT    A, R0
```

7.4 Code Generation from a Labelled Tree

We conclude this section by presenting the Prolog programs implementing the optimal code generation applicable to labelled trees as described in [1]

The labelling phase consists of a postorder walk on a syntax-tree in which left leaves are labelled with a 1 and right leaves with a 0. Interior nodes are labelled by *max(left, right)* if the *left* label is different from the *right* one; otherwise the interior node is labelled with *right* + 1. The label of the root specifies the total (optimal) number of registers needed to code the syntax-tree without using temporary locations. In Prolog the labelling is accomplished by the clause *label*, having four parameters: (1) the original syntax-tree, (2) a mark denoting a left or right branch, (3) the generated labelled tree, and (4) the node label itself.

```
label(var(X), left, var(X, 1), 1).
label(var(X), right, var(X, 0), 0).
label(expr(Op, Left, Right), Z, expr(Op, E1, E2, Label), N) :-
  label(Left, left, E1, Label1),
  label(Right, right, E2, Label2),
  max(Label1, Label2, Label).

max(N, N, N1) :- N1 is N + 1.
max(N, N1, N) :- N > N1.
max(N1, N, N) :- N < N1.
```

The actual code generation algorithm is practically the same as that presented on p. 544 of [1]. The parameters of *gencode* are (1) the labelled syntax-tree, (2) the register stack, (3) the maximum number of registers, and (4) the next available temporary location. The procedure will output code for the expression in such a way that the result of the expression is stored in the register at the top of the register stack. (We remind the reader that any program that uses *write* to produce its results can easily be modified to store the results in a list, as described in Section 4).

The first two clauses consider the simplest cases dealing with leaves. The third clause is applicable only when the right subexpression is a tree. It finds the labels of the two subexpressions and calls *gencode*1 which generates code using the minimal number of registers and temporaries. We purposely avoided the use of cuts by ensuring that each of the clauses deal with mutually exclusive cases, and so no backtracking is possible.

```
% case 0: left expression is a leaf.
gencode(var(X, 1), [Reg | RestR], Max, Temp) :- print(move, X, Reg).

% case 1: right expression is a leaf.
gencode(expr(Op, X, var(Y, 0), Label), [Reg | RestR], Max, Temp) :-
   gencode(X, [Reg | RestR], Max, Temp),
   write(Op, Y, Reg).

% cases 2a, 2b, and 2c: left and right expressions are trees.
gencode(expr(Op, L, R, Any_Label), Regs, Max, Temp) :-
   labelvalue(L, NL),
   labelvalue(R, NR), NR > 0
   gencode1(expr(Op, L, R, Any_Label), Regs, Max, NL, NR, Temp).

% case 2a: Left expression can be computed without temporaries.
gencode1(expr(Op, L, R, Any_Label), [Reg1, Reg2 | RestR], Max, N1, N2, Temp) :-
   N1 < N2, N1 < Max,
   gencode(R, [Reg2, Reg1 | RestR], Max, Temp),
   gencode(L, [Reg1 | RestR], Max, Temp),
   write(Op, Reg2, Reg1).

% case 2b: Right expression can be computed without temporaries.
gencode1(expr(Op, L, R, Any_Label), [Reg1, Reg2 | RestR], Max, N1, N2, Temp) :-
   N2 =< N1, N2 < Max,
   gencode(L, [Reg1, Reg2 | RestR], Max, Temp),
   gencode(R, [Reg2 | RestR], Max, Temp),
   write(Op, Reg2, Reg1).

% case 2c: temporaries are required.
gencode1(expr(Op, L, R, Any_Label), [Reg | RestR], Max, N1, N2, Temp) :-
   N1 ≥ Max, N2 ≥ Max,
   gencode(R, [Reg | RestR], Max, Temp),
   write(move, Reg, t(Temp)),
   NextTemp is Temp + 1,
   gencode(L, [Reg | RestR], Max, NextTemp),
   write(Op, t(Temp), Reg).

% miscellaneous.
labelvalue(expr(Any1, Any2, Any3, N), N).
labelvalue(var(Any1, N), N).
```

Assuming that two registers are available, the code generated for the expression $((A - B)/(C - D))/((E - F)/(G - H))$ is

```
MOVE    E, R0
SUB     F, R0
MOVE    G, R1
SUB     H, R1
DIV     R1, R0
MOVE    R0, T0
MOVE    A, R0
SUB     B, R0
MOVE    C, R1
SUB     D, R1
DIV     R1, R0
DIV     T0, R0
```

We conclude this section by pointing out that Cattell's method of code generation is a prime candidate for prototyping using Prolog. In his dissertation, Cattell [5] proposes a method for formalizing and automatically deriving code generators from machine descriptions. The method consists of constructing a syntax-tree-like description for each instruction in the machine's repertoire. A special tree-matching program then generates code sequences by combining the available instruction syntax-trees so that they match the syntax-tree representing a source program. Cattell's approach combines AI techniques with those in current use in compiler construction.

Although the code generators described herein are specialized to the case of Algol-like programs, Prolog has already proved its usefulness in writing Prolog compilers [3]. At Brandeis we have developed a Prolog compiler that compiles Prolog programs into equivalent C programs [8]. A remarkable feature of these compilers is their conciseness and the ease with which they can be changed to generate code in various target languages.

8. OPTIMIZATIONS

8.1 Compile-Time Evaluation

Compile-time evaluation of numerical expressions and algebraic simplification are easily performed by transforming the syntax-trees of arithmetic expressions into equivalent trees containing fewer nodes. Both of the procedures *evaluate* and *simplify* have as arguments the initial and final trees. They also have a similar structure: recursive calls are made to process the left and right branches until the leaves are reached. Then the auxiliary procedure *simp* is called to perform the actual simplifications. This allows successive simplifications to be performed.

```
% leaves are left unchanged
evaluate(const(X), const(X)).
evaluate(var(X), var(X)).
```

```
% internal nodes are optimized after each of its subtrees
% has been optimized
evaluate(expr(Op, Left, Right), Optexp) :-
  evaluate(Left, Optleft),
  evaluate(Right, Optright),
  simp(expr(Op, Optleft, Optright), Optexp).
simp(expr(Op, const(X), const(Y)), const(Z)) :-
  Temp =.. [Op, X, Y], Z is Temp.
```

(In the Edinburgh syntax [6], the operation *Temp* =.. [*Op*, *X*, *Y*] binds *Temp* to the term *Op*(*X*, *Y*)). Note that, unfortunately, this procedure is unable to simplify expressions such as $a + 3 + 2$ into $a + 5$. This may be achieved by writing a simple procedure that transforms left-associative expressions into equivalent right-associative expressions. The procedure that performs algebraic simplifications is

```
simplify(expr(Op, X, Y), U) :-
  simplify(X, Left),
  simplify(Y, Right),
  simp(expr(Op, Left, Right), U).
simplify(X, X).
```

As before, the auxiliary procedure *simp* performs the actual simplifications.

```
simp(expr(Op, X, const(0)), X) :- addop(Op).
simp(expr(Op, X, const(1)), X) :- multop(Op).
simp(expr(*, X, const(0)), const(0)).
simp(expr(*, const(0), X), const(0)).
simp(X, X).
addop(+). addop(-). multop(*). multop(/).
```

8.2 Peephole Optimization

Table I summarizes some of the typical peephole optimizations that can be performed after code generation. The table also indicates the source program segments, resulting in code that can be optimized in this manner.

We first note that if Warren's approach (Section 7.2) is used for code generation, an additional pass is needed to "flatten" the list that makes up the generated code. This list contains sublists resulting from the order in which the clauses *endcode statements* are activated. Assuming that the code consists of a list of elements separated by (right-associative) semicolons, it is a simple matter to express in Prolog the optimizations in Table I:

```
% if pattern is found perform the optimization.
peep([instr(sto, X), instr(load, X) | L], [instr(sto, X) | M]) :- peep(L, M).
peep([instr(subi, 0) | L], M) :- peep(L, M).
peep([label(A), instr(jump, A) | L], M) :- peep([instr(jump, A), L], M).
% keep trying with tail of list.
peep([X | L], [X | M]) :- peep(L, M).
peep([ ], [ ]).
```

Note that the above program can handle the nondeterministic situations arising when the code to be inspected using the first parameter renders more than one *peep* clause applicable. The resulting nondeterministic searches could result in

Table I. Peephole Optimization

Source code	Compiled code	Optimization code
$a := \cdots$	STO a	STO a
$b := a \cdots$	LOAD a	
if $a < 0$ then $\cdots$	SUBI 0	ϵ
while ... do	JUMP a	JUMP b
if ... then ... else;	$\cdots$	$\cdots$
	a: JUMP b	a: JUMP b

longer processing times. It is up to the designer to decide whether this overhead brings significant gains in the execution of the optimized code. A careful ordering of the clauses and a judicious introduction of cuts can reduce some of the overhead. It is also possible to control the amount of backtracking by introducing and tallying costs which, when exceeded, trigger the choice of alternate paths (see Section 9).

Finally, it should be mentioned that David Hildum and the first author were able to implement, using Prolog, all of the (over one hundred) peephole optimizations applicable to a P-code-like intermediate language [27]. This was achieved by prototyping a language for specifying the transformations. An interesting aspect of this implementation is that DCG rules are used to generate another set of DCG rules needed to match and replace patterns.

9. USING PROPOSED EXTENSIONS

Several extensions have been proposed to enhance the capabilities of Prolog. The reader is referred to [8] for a brief description of some of these extensions. Two of them are of special interest in compiler construction and are dealt with in this section: the use of the built-in predicate *freeze* and unification involving infinite trees. These features are available in Prolog II [10], in the interpreter developed by Carlsson [4] and in MU-Prolog [23][2].

The predicate *freeze* (also referred to as lazy evaluation, or coroutining) has the form

$$freeze(Var, Procedure)$$

Its action is to immediately activate the given *Procedure* if the variable *Var* is bound. Otherwise, the *Procedure* becomes a dormant goal until *Var* is bound. In that event, *Procedure* becomes the next goal to be activated. It is straightforward to write a metalevel interpreter that simulates the effect of *freeze* [8]. This interpreter is admittedly inefficient. Nevertheless, when compiled, it is usable for processing small examples.

We illustrate the use of *freeze* in two contexts: (1) coroutining the scanning, parsing, and code generation phases of a compiler, and (2) error detection and recovery.

The coroutining of the phases is particularly useful when parallel processing is available: it allows the intermediate results of one phase to be transmitted to

[2] One purpose of presenting them here is to generate interest, so that they will become more generally available.

the subsequent phase and therefore speed-up the computation by triggering simultaneous executions whenever possible. (In Warren's compiler [31] the phases are strictly sequential.) Consider the simple procedure

```
readlist(L) :- read(X), readrest(X, L).
readrest(stop, [ ]).
readrest(X, [X | L]) :- readrest(L).
```

The built-in predicate *read* reads individual atoms, and *readlist* assembles them into a list. The atom *stop* is used as a flag to terminate the reading.

The availability of *freeze* allows one to write a *writelist* procedure which outputs the elements of a list as soon as they are read:

```
writelist([ ]).
writelist([H | T]) :- freeze(H, write(H)), freeze(T, writelist(T)).
```

The query is: *?- freeze(L, writelist(L)), readlist(L).*

The same ideas can be used to alternately transfer control among the scanner, the parser, and the code generator. The main procedure *compile* is

```
compile :- freeze(Tree, encode_statement(Tree, Dict, Code)),
           freeze(List, parse(List, Tree)),
           scan(List).
```

The above states that *parse* can only be activated as soon as (a part of) a *List* is available. Similarly, *encode_statement* is activated as soon as a (partially) instantiated syntax-tree becomes available. It is of course necessary to "sprinkle" additional *freeze*s within *parse* and *encode_statement.* This is illustrated below by examples. The translated DCG rule for parsing a *while* statement becomes

```
statement(while(Test, Do), [while | D1], D4) :-
  freeze(D1, test(Test, D1, D2)),
  freeze(D2, eq(D2, [do | D3])),
  freeze(D3, statement(Do, D3, D4)).
```

The second and third parameters of *statement* and *test* are the difference lists for parsing strings derived from the corresponding nonterminals. The procedure *eq* is simply the unit clause *eq*(X, X) which unifies its arguments. The effect of freezing on $D1$, $D2$, and $D3$ is to allow *test* and the recursive call of *statement* to be activated only when the pertinent information becomes available. Similarly, the code generator for a *while* node of the syntax-tree (see Section 7.2) becomes

```
encode_statement(while(Test, Do), Dict, . . .) :-
  freeze(Test, encode_test(Test, Dict, L2, Testcode)),
  freeze(Do, encode_statement(Do, Dict, Docode)).
```

Figure 1 shows the alternating flow of control among the scanner, parser, and code generator while compiling a small program using the coroutining technique. The reader might have already suspected that the introduction of *freeze*s could be done automatically. We have indeed developed programs that perform this task, based on user-specified mode declarations (input or output) for each parameter of a procedure.

Another usage of *freeze* is in error detection and recovery. The following example just illustrates the main ideas, which are based on the work of Mickunas

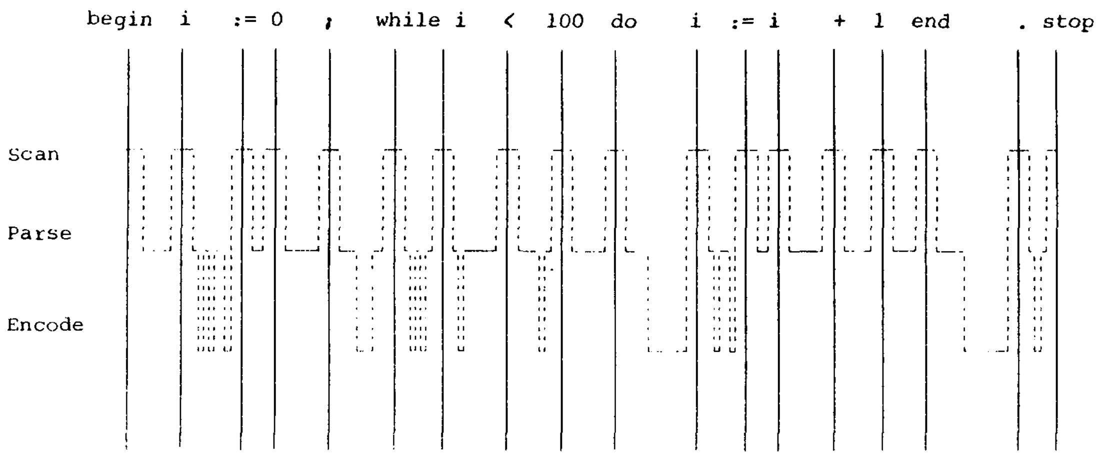


Figure 1

and Modry [22]. At the top level the procedure *recover* has two parameters: (1) the possibly erroneous input string and (2) the corrected string.

recover(*Tokens*, *Tree*) :–
  freeze(*Filtered_tokens*, *parse*(*Filtered_tokens*, *Tree*)),
  correct(*Tokens*, *Filtered_tokens*, 0).

The variable *Filtered_tokens* is initially unbound; *parse* will call the corresponding procedures that use the difference lists. The third parameter of *correct* is the initial cost of correction. The approach consists of attempting to insert or to delete tokens in the input string so that an erroneous string becomes parsable. If necessary, different costs for insertion and deletion, applicable to specific terminals, can be specified by the designer. In the simplified version of *correct* listed below, a unit cost is used for both operations. The database contains a unit clause *cost_ok*(*max*) in which *max* is a number that controls the amount of backtracking.

```
% final scan
correct([ ], [ ], Cost).
% normal scan
correct([X | R], [X | R1], Cost) :– correct(R, R1, Cost).
% deletion
correct([X | R], R1, Cost) :–
  cost_ok(Cost),
  Cost1 is Cost + 1,
  correct(R, R1, Cost1).
% insertion
correct(R, [I | R1], Cost) :–
  cost_ok(Cost),
  Cost1 is Cost + 1,
  correct(R, R1, Cost1).
```

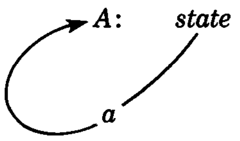
Note that the variable *I* in the insertion clause will be bound by the parser according to the grammar rules.

A more elaborate version of *correct* could reduce the amount of nondeterminism by making insertions and deletions based on examining the (fragments of the) parse tree constructed prior to encountering an error. This is the approach described in [22].

In addition to the two above uses of *freeze*, we have explored its application in dataflow analysis. The iterative methods described in [1] can be implemented using a variant of *freeze* in which the frozen variables simulate the incoming and outgoing flow of information for each block.

The other proposed extension of Prolog that is useful in compiler design deals with the so-called infinite trees. It is Colmerauer's contention that grammars, flowcharts, and programs frequently specify loops or recursion [11], which can be conveniently described using directed graphs. Their use within Prolog requires that the unification operation be extended to handle circular structures instead of trees.

An elegant and novel approach for implementing a scanner generator using infinite trees has been developed by students of the University of Marseilles [12] under the guidance of A. Colmerauer. It consists of using a special type of unification to produce the minimal finite state automaton *directly* from a given regular expression. Most Prolog interpreters perform unification only on trees. A notable exception is the interpreter developed at Marseilles, which can unify special kinds of graphs called *infinite trees* [10]. For example, when the unit clause $eq(X, X)$ is matched with $eq(A, state(a(A)))$, the resulting unification is expressed by the infinite tree:



Terms representing states have an additional component specifying whether the state is final or not. The procedure to translate a regular expression into the corresponding minimal finite state automaton takes as input the expression given by its syntax-tree and produces as result the infinite tree corresponding to the minimal automaton. The highlights of this translation are given in what follows.

If a node of the syntax-tree is a *conc*(*Left*, *Right*), one recursively determines the automata corresponding to the *Left* and *Right* branches and "concatenates" the two automata to obtain the result. Concatenation of two automata $A1$ and $A2$ is performed by checking whether the starting state of $A1$ is *final* or *not-final.* In the first case the resulting automata is the union of the automata $A2$ with the concatenation of automata $A1'$ and $A2$, in which $A1'$ is a modified copy of $A1$ in which the starting state is considered to be *not-final.* In the second case the concatenation of the two automata consists of specifying the proper transitions between the final states of $A1$ directly to the states that stem from the initial state of $A2$. The *union* and *star* operations are processed similarly.

A dictionary is "carried along" as a parameter to provide the information needed to keep a single copy of each of the generated subautomata needed to construct the desired one. Therefore, before proceeding to generate an automaton corresponding to a subpart of a regular expression, the dictionary is used to check if the translation has already been done. If so, the desired subautomaton is retrieved from the dictionary. Otherwise, the automaton is determined and the corresponding entry is placed in the dictionary. This *per se* does not guarantee the construction of a minimal automaton. The program that "prints" the desired infinite tree is actually the one responsible for the minimization [26]. Again with the use of a dictionary, the printing program keeps unique copies of each subtree of the given infinite tree and uses them every time identical subtrees are found. This process has been proved to terminate [11], and for the particular problem at hand it yields the desired minimal automaton.

The authors of this program [12] extended its capabilities to handle the difference and intersection of regular expressions. The program hardly exceeds three pages of code; it also uses another feature that is only available in Marseilles' interpreters: the constraint *diff*(X, Y), meaning $X \neq Y$ is valid even when X and Y are uninstantiated, therefore allowing the program's execution to continue in the forward mode. Backtracking is thus postponed until it is found that the ensemble of constraints becomes unsatisfiable.

10. FINAL REMARKS

In the previous sections we described in Prolog several algorithms that play an important role in the design and construction of compilers. We hope it has become apparent to the reader that the descriptions using Prolog are substantially more concise than those which appear in current textbooks. For example, Aho and Ullman often use a mixture of English, the language of sets, and control primitives usually found in Pascal-like languages. The reader is urged to compare some of their descriptions to those presented in this paper.

The experience we gained with Prolog has convinced us of its effectiveness as a language for rapid prototyping compilers and for developing ancillary tools. Presently, the highest gains are achieved in the development of tools in which performance is not of prime consideration. This is the case of automatically producing code generators, parsers, and scanners. Even if the generation of these components takes considerable computer time (say a few hours), the combined man–machine effort may be inexpensive when compared to the human resources needed to produce their hand-coded counterparts. Another area in which the language has proved its usefulness is in the writing of compilers for Prolog itself. It is fair to say that most Prolog compilers are written in Prolog. The gains are substantial, especially because they have to process relatively short programs, and compilation can be done incrementally as the procedures are developed.

Yet another advantage of using Prolog programs is their ability to perform computations both in the forward and reverse directions. It should therefore be possible to decompile target code to obtain the corresponding source code. Although this is in principle feasible, the use of "impure" Prolog features such as the cut and the assignment (*is*) render the reverse execution impossible. These problems may be circumvented by using the generalized *diff*, mentioned in Section 9, and by ensuring that simple assignments such as those incrementing the values of variables become backtrackable.

Among the shortcomings of Prolog, it should be mentioned that the language is still in evolution and that, presently, a suitable environment for developing larger Prolog programs is not yet available. The language also suffers from the nonexistence of a methodology for documentation, the lack of scoping for variables, the ever-increasing number of parameters, and the resulting profusion of identifier names.

Benchmarks of the parsers and code generators described in this paper showed that their *interpretation* is indeed slower than the compiled equivalent programs written in C or in Pascal. *Compiled* Prolog programs running on a dedicated workstation exhibit 5- to 10-fold speed-ups compared to their interpreted versions. For example, the compiled version of Warren's minicompiler enabled us to generate code for sample programs containing a few hundred statements in a couple of minutes. Such compilation speeds are still admittedly below those attained by equivalent compilers written in C. However, we feel that there is a great potential for improving considerably the performance of compilers written in Prolog. The justifying arguments are as follows.

The advantages of Prolog basically stem from the use of unification and nondeterminism. The present price paid for the advantages are increasing demands in memory and execution time. Since compilers are usually designed to avoid nondeterministic situations, it is important to reduce Prolog's interpreter (or compiled code) overhead for dealing with these situations. Once it is known that a Prolog program is deterministic, several optimizations can be carried out. One of them is to eliminate the need of saving choice points for backtracking purposes. The optimized program can then achieve the efficiency of the corresponding programs written in a functional language (see [30]).

In a recent paper, Mellish [21] provides weak conditions for determining automatically if a set of Prolog procedures is deterministic. His method is based on a dataflow analysis in which properties of programs are determined by iteratively solving a system of equations. The efficiency of the compiled code can also be increased by having the user supply, by a mode declaration, the nature (input or output) of each parameter of a procedure. This allows the compiler not only to discard certain nondeterministic situations, but also to replace costly unifications by the simpler operations of assignments and conditionals.

A possibility that should not be overlooked in the quest to speed-up Prolog programs is the use of parallel processing. In contrast with most other languages, Prolog offers an embarrassment of riches for exploiting parallelism. The experience gained by empirical or theoretical analysis of parallel Prolog compilers may therefore help to shed some light as to which particular approach yields better speed-up gains.

We feel that the initial investment spent in learning Prolog is largely compensated for by the advantages accrued in having a shorter program-development stage and achieving program descriptions that can easily be tried and tested in a computer. It is also possible that other higher level languages such as SETL could be used with the same purpose. What seems certain is that the availability of these languages will make program description less verbose and more accurate. In addition, they will spur the development of optimization techniques capable of rendering efficient the descriptions that are not directly presented in an efficient form. The history of the development of Fortran and other languages indicates that this is not only a desirable goal but likely an unavoidable one.

ACKNOWLEDGMENTS

The first author's initiation to Prolog developed from the close contacts that he has had with the Groupe d'Intelligence Artificielle (GIA) at the University of Marseilles, Luminy, in France. Alain Colmerauer, Michel van Caneghem, Henri

Kanoui, Bob Pasero, and Francis Giannesini were all enthusiastic in sharing with him their knowledge of the language they have developed and refined at GIA. Three graduate students from Marseilles: Sylvie Duchenoy, Robert Kong Win Chang, and Sophie Nabitz helped in testing the programs presented here. In particular, Robert Kong, now at Brandeis, has dedicated countless hours in helping us polish the paper. David Hildum implemented the Glanville–Graham code-generation method described in Section 7. Peter Appel and Cindy Lurie did their honor's projects prototyping Prolog versions of LEX and YACC. We count ourselves lucky to have had the opportunity to interact with the above-mentioned persons.

Finally, we wish to express our gratitude to a referee, David S. Warren, who, following a meticulous reading of the original manuscript, helped identify the major issues of Prolog usage in compiling and urged us to discuss them in the revised paper. The thoughtful and detailed remarks made by this referee provided an added incentive to improve the paper and reaffirmed our respect for the refereeing process.

REFERENCES

1. AHO, A. V., AND ULLMAN, J. D. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1979.
2. BACKHOUSE, R. C. *Syntax of Programming Languages.* Prentice-Hall, Englewood Cliffs, N.J., 1979.
3. CAMPBELL, J. A. (ED.) *Implementations of Prolog.* Wiley, New York, 1984.
4. CARLSSON, M. A microcoded unifier for Lisp machine Prolog. In *IEEE Proceedings 1985 Symposium on Logic Programming* (Boston, July 1985), IEEE, New York, 1985, 162–171.
5. CATTELL, R. G. G. Automatic derivation of code generators from machine descriptions. *ACM Trans. Programm. Lang. Syst. 2*, 2 (Apr. 1980), 173–190.
6. CLOCKSIN, W. F., AND MELLISH, C. S. *Programming in Prolog.* Springer-Verlag, New York, 1981 (2nd ed., 1984).
7. COHEN, J., SITVER, R., AND AUTY, D. Evaluating and improving recursive descent parsers. *IEEE Trans. Softw. Eng. SE-5*, 5 (Sept. 1979), 472–480.
8. COHEN, J. Describing Prolog by its interpretation and compilation. *Commun. ACM 28*, 12 (Dec. 1985), 1311–1324.
9. COLMERAUER, A. Les Grammaires de Métamorphose. Groupe d'Intelligence Artificielle, Univ. of Marseilles–Luminy, 1975. (Appears as *Metamorphosis grammars* in *Natural Language Communication with Computers.* L. Balc, Ed., Springer-Verlag, New York, 1978, 133–189.)
10. COLMERAUER, A., KANOUI, H., AND VAN CANEGHEM, M. Prolog II. Groupe d'Intelligence Artificielle, Univ. of Marseilles–Luminy, 1982.
11. COLMERAUER, A. Prolog and infinite trees. In *Logic Programming,* Clark and Tarnlund (Eds.), Academic Press, New York, 1982, 231-251.
12. COUPET, S., AND DUPLESSIS, F. Prolog programs for transforming regular expressions into the corresponding minimal finite state recognizers. Memoire de D.E.A., GIA, Univ. of Marseilles, June 1984 (in French).
13. EARLEY, J. An efficient context-free parsing algorithm. *Commun. ACM 13*, 2 (Feb. 1970), 94–102.
14. GIANNESINI, F., AND COHEN, J. Parser generation and grammar manipulations using Prolog's infinite trees. *J. Logic Programm.* (Oct. 1984), 253–265.
15. GLANVILLE, R. A machine independent algorithm for code generation and its use in retargetable compilers. Ph.D. dissertation, Univ. of California, Berkeley, 1977.
16. GRAHAM, S. L., HENRY, R. R., AND SHULMAN, R. A. An experiment in table driven generation. In *Proceedings SIGPLAN Symposium on Compiler Construction 17*, 6 (June 1982), 32–43.
17. GRIES, D. *Compiler Construction for Digital Computers.* Wiley, New York, 1971.

18. GRIFFITHS, T. V., AND PETRICK, S. R. On the relative efficiencies of context-free grammar recognizers. *Commun. ACM 8*, 5 (May 1965), 289–300.
19. ICHBIAH, J. D., AND MORSE, S. P. A technique for generating almost optimal Floyd–Evans productions for precedence grammars. *Commun. ACM 13*, 8 (Aug. 1970), 501–508.
20. KOWALSKI, R. *Logic for Problem Solving*. North-Holland, Amsterdam, 1979.
21. MELLISH, C. S. Some global optimizations for a Prolog compiler. *J. Logic Programm. 2*, 1 (Apr. 1985), 43–66.
22. MICKUNAS, M. D., AND MODRY, J. A. Automatic error recovery for LR parsers. *Commun. ACM 21*, 6 (June 1978), 459–465.
23. NAISH, L. Automating control for logic programs. *J. Logic Programm. 3* (1985), 167–183.
24. PEREIRA, F. C. N., AND WARREN, D. H. D. Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics* (Cambridge, Mass., June 1983), Association for Computational Linguistics, 1983, 137–144.
25. PEREIRA, F. C. N., AND WARREN, D. H. D. Definite clause grammars for language analysis. *Artif. Intell. 13* (1980), 231–278.
26. PIQUE, J. F. Drawing trees and their equations in Prolog. In *Proceedings of the 2nd International Logic Programming Conference* (Uppsala, 1984), Ord and Furm, Uppsala, 1984, 23–33.
27. TANENBAUM, A. S., VAN STAVEREN, H., AND STEVENSON, J. W. Using peephole optimization on intermediate code. *ACM Trans. Programm. Lang. Syst. 4*, 1 (Jan. 1982), 21–36.
28. UEHARA, K., OCHITANI, R., AND KAKUSHO, O. A bottom-up parser based on predicate logic. In *IEEE Proceedings 1984, International Symposium on Logic Programming* (Atlantic City, N.J., Feb. 1984), IEEE, New York, 1984, 220–227.
29. WAITE, W. M., AND GOOS, G. *Compiler Construction*. Springer-Verlag, New York, 1984.
30. 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, 1983).
31. WARREN, D. H. D. Logic programming and compiler writing. *Softw. Pract. Exper. 10* (Feb. 1980), 97–125.

Received October 1984; revised October 1985 and April 1986; accepted May 1986