Chapter 2
INTRODUCTION TO LABORATORY ACTIVITIES

The laboratory activities introduced in this chapter and used elsewhere in the text are not required to understand definitional techniques and formal specifications, but we feel that laboratory practice will greatly enhance the learning experience. The laboratories provide a way not only to write language specifications but to test and debug them. Submitting specifications as a prototyping system will uncover oversights and subtleties that are not apparent to a casual reader. This laboratory approach also suggests that formal definitions of programming languages can be useful. The laboratory activities are carried out using Prolog. Readers not familiar with Prolog, should consult Appendix A or one of the references on Prolog (see the further readings at the end of this chapter) before proceeding.

In this chapter we develop a "front end" for a programming language processing system. Later we use this front end for a system that check the context-sensitive part of the Wren grammar and for prototype interpreters based on semantic specifications that provide implementations of programming languages.

The front end consists of two parts:

1. A scanner that reads a text file containing a Wren program and builds a Prolog list of tokens representing the meaningful atomic components of the program.
2. A parser that matches the tokens to the BNF definition of Wren, producing an abstract syntax tree corresponding to the Wren program.

Our intention here is not to construct production level software but to formulate an understandable, workable, and correct language system. The Prolog code will be kept simple and easy to read, since the main purpose here is to understand the definitional techniques studied in the text. Generally, only primitive error handling is provided, so that the scanner-parser system merely fails when a program has incorrect context-free syntax.

The system requests the name of a file containing a program, scans the program producing a token list, and parses it, creating an abstract syntax tree

representing the structure of the program. The transcript below shows a typical execution of the scanner and parser. User input is depicted in boldface. The token list and abstract syntax tree have been formatted to make them easier to read.

```
| ?- go.

>>> Scanning and Parsing Wren <<<

Enter name of source file: switch.wren

    program switch is
      var sum,k : integer;
      var switch : boolean;
    begin
      switch := true; sum := 0;  k := 1;
      while k<10 do
        switch := not(switch);
        if switch then sum := sum+k end if;
        k := k+1
      end while;
      write sum
    end

Scan successful

[program,ide(switch),is,var,ide(sum),comma,ide(k),colon,integer,
 semicolon,var,ide(switch),colon,boolean,semicolon,begin,
 ide(switch),assign,true,semicolon,ide(sum),assign,num(0),
 semicolon,ide(k),assign,num(1),semicolon,while,ide(k),less,
 num(10),do,ide(switch),assign,not,lparen,ide(switch),rparen,
 semicolon,if,ide(switch),then,ide(sum),assign,ide(sum),plus,
 ide(k),end,if,semicolon,ide(k),assign,ide(k),plus,num(1),end,
 while,semicolon,write,ide(sum),end,eop]

Parse successful

prog([dec(integer,[sum,k]),dec(boolean,[switch])],
     [assign(switch,true),assign(sum,num(0)),assign(k,num(1)),
      while(exp(less,ide(k),num(10)),
         [assign(switch,bnot(ide(switch))),
          if(ide(switch),
             [assign(sum,exp(plus,ide(sum),ide(k)))],skip),
          assign(k,exp(plus,ide(k),num(1)))]),
      write(ide(sum))])

yes
```

Observe that the program "switch.wren", although satisfying the context-free syntax of Wren, is syntactically illegal. Review the context constraints in Figure 1.11 to identify the (minor) error. Several predefined predicates, primarily for input and output, are used to build the front end of the language processing system. See Appendix A for a brief description of these predicates.

2.1 SCANNING

The scanner reads the program text and produces a list of tokens according to the lexical syntax of the programming language. Recall that the lexical syntax can be defined using a regular grammar—for example,

<numeral> ::= **0** | **1** | ... | **9** | **0** <numeral>
| **1** <numeral> | ... | **9** <numeral>,

which we abbreviate as

<numeral> ::= <digit> | <digit> <numeral>
<digit> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**.

First we recognize a digit by specifying a span of ascii values.

```
digit(C)  :- 48 =< C, C =< 57.                % 0-9
```

The symbol "%" signals a comment that extends to the end of the line.

The form of these productions fits nicely with processing a stream of characters in Prolog. We name the predicate that collects a sequence of digits into a numeral `getnum` and write the productions for numeral as

```
getnum ::= digit | digit getnum.
```

The first digit tells us that we have a numeral to process. We split the production into two parts, the first to start the processing of a numeral and the second to continue the processing until a nondigit occurs in the input stream.

```
getnum ::= digit restnum

restnum ::= ε | digit restnum      % ε represents an empty string
```

We describe these regular productions using the transition diagram shown in Figure 2.1.

Parameters are then added to the nonterminals, the first to hold a readahead character and the second to contain the numeral being constructed, either as a Prolog number or as a list of ascii values representing the digits.

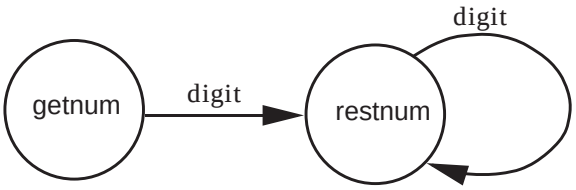


Figure 2.1: A Transition Diagram for getnum

The predicates are defined by

```
getnum(C,N) :- digit(C), get0(D), restnum(D,Lc), name(N,[C|Lc]).
restnum(C,[C|Lc]) :- digit(C), get0(D), restnum(D,Lc).
restnum(C,[ ]).                % end numeral if C is not a digit
```

and the numeral processor is initiated by the query

```
get0(C), getnum(C,N).
```

The first get0 acts as a priming read that gets (inputs) the first character, which is bound to C for getnum to process. Then getnum verifies that the first character is a digit, gets the next character, which is bound to D, and asks restnum to construct the tail Lc of the numeral. When restnum returns with the tail, the entire list [C|Lc] of digits passes to the predefined predicate name, which converts it into a Prolog number N. The predicate restnum reads characters forming them into a list until a nondigit is obtained, finishing the list with an empty tail.

Figure 2.2 shows a trace following the processing of the numeral given by the characters "905" followed by a return. This string generates the ascii values 57, 48, 53, and 10 on the input stream. Note that variables are numbered, since each recursive call requires a fresh set of variables to be instantiated. For example, the predicate get0 is called four times with the variables C, D, D1, and D2, respectively.

This example illustrates the basic principle behind the scanner—namely, to get the first item in a list and then call another predicate to construct the tail of the list. This second predicate is called repeatedly, with the items in the list being constructed as the first elements of the subsequent tails. When no more items are possible, the empty tail is returned. In order to comprehend the scanner better, we uniformly name the variables found in it:

Character (an ascii value): C, D, E

Token (a Prolog atom or simple structure): T, U

List of Characters: Lc

List of Tokens: Lt

Query	Bindings
get0(C)	C = 57
getnum(57,N)	
digit(57)	
get0(D)	D = 48
restnum(48,Lc)	Lc = [48|Lc1]
digit(48)	
get0(D1)	D1 = 53
restnum(53,Lc1)	Lc1 = [53|Lc2]
digit(53)	
get0(D2)	D2 = 10
restnum(10,Lc2)	Lc2 = [10|Lc3]
digit(10)	fails
restnum(10,Lc2)	Lc2 = [ ]
name(N,[57|Lc])	
where Lc = [48|Lc1] = [48,53|Lc2] = [48,53]	
name(N,[57,48,53]) gives N=905.	

Figure 2.2: Reading the Numeral 905

The predicates used in the scanner are described informally below, following the convention that input variables are marked by "+" and output variables by "–". Although most predicates in Prolog are invertible, meaning that variables can act as input or output parameters in different applications, the scanner needs to act in one direction only since it involves the side effect of reading a text file. The marking of parameters by "+" and "–" below will help the reader understand the execution of the scanner. Observe that some of the predicates have two variables for characters, one for the current lookahead character and one for the next lookahead character.

scan(Lt)

– Construct a list of tokens Lt.

restprog(T,D,Lt)

+ T is the previous token.

+ D is the lookahead character.

– Construct a list of tokens Lt from the rest of the program.

getch(C)

– Get the next character C and echo the character.

gettoken(C,T,D)

+ C is the lookahead character.
– Construct the next token T and
– find the next lookahead character D.

restnum(C,[C|Lc],E)

+ C is the lookahead character.
– Construct the tail of the number Lc and
– find the next lookahead character E.

restid(C,[C|Lc],E)

+ C is the lookahead character.
– Construct the tail of the identifier or reserved word string Lc and
– find the next lookahead character E.

To enable the scanner to recognize the different classes of characters, we define predicates, each representing a particular set of characters.

```
lower(C) :- 97=<C, C=<122.          % a-z
upper(C) :- 65=<C, C=<90.           % A-Z
digit(C) :- 48 =< C, C=< 57.        % 0-9
space(32).          tabch(9).          period(46).          slash(47).
endline(10).        endfile(26).       endfile(-1).
whitespace(C) :- space(C) ; tabch(C) ; endline(C).
idchar(C) :- lower(C) ; digit(C).
```

At the top level of the scanner, scan gets the first character, calls gettoken to find the first token, and then uses restprog to construct the rest of the token list. Each line of the program listing is indented four spaces by means of tab(4). Both scan and restprog invoke gettoken with a lookahead character C. When the end of the file is reached, gettoken returns a special atom eop symbolizing the end of the program. Note that getch performs special duties if the current character represents the end of a line or the end of the file.

```
scan([T|Lt]) :- tab(4), getch(C), gettoken(C,T,D), restprog(T,D,Lt).
getch(C) :- get0(C), (endline(C),nl,tab(4) ; endfile(C),nl ; put(C)).
restprog(eop,C,[ ]).          % end of file reached with previous character
restprog(T,C,[U|Lt]) :- gettoken(C,U,D), restprog(U,D,Lt).
```

To classify symbolic tokens, we need to identify those that are constructed from a single symbol and those that are constructed from two characters. Unfortunately, the first character in the two-character symbols may also stand alone. Therefore we classify symbols as single or double and provide a predicate to recognize the two-character symbols. Symbols specified by the predi-

cate single consist of a single character. This predicate associates a token name with the ascii code of each character.

```
single(40,lparen).      single(41,rparen).      single(42,times).
single(43,plus).        single(44,comma).       single(45,minus).
single(47,divides).     single(59,semicolon).   single(61,equal).
```

Characters that may occur as a symbol by themselves or may be the first character in a string of length two are recognized by the predicate double. The second argument for double names the token given by the one-character symbol.

```
double(58,colon).       double(60,less).        double(62,grtr).
```

If, however, the symbol is two characters long, pair succeeds and provides the name of the token.

```
pair(58,61,assign).     % :=
pair(60,61,lteq).       % <=
pair(60,62,neq).        % <>
pair(62,61,gteq).       % >=
```

We also need to recognize the reserved words in Wren. The predicate reswd defines the set of reserved words.

```
reswd(program).   reswd(is).      reswd(begin).    reswd(end).
reswd(var).       reswd(integer). reswd(boolean).  reswd(read).
reswd(write).     reswd(while).   reswd(do).       reswd(if).
reswd(then).      reswd(else).    reswd(skip).     reswd(or).
reswd(and).       reswd(true).    reswd(false).    reswd(not).
```

Figure 2.3 displays a transition diagram for analyzing the kinds of tokens in Wren. The Prolog code for scanning tokens is given below. Numerals are handled in the manner we discussed earlier. Although the productions for identifiers permit reserved words to be treated as identifiers, the scanner will first check each character string to see whether it is an identifier or a reserved word. Identifier tokens take the form ide(sum) while reserved words stand for themselves as Prolog atoms.

```
gettoken(C,num(N),E) :- digit(C), getch(D), restnum(D,Lc,E), name(N,[C|Lc]).
restnum(C,[C|Lc],E) :- digit(C), getch(D), restnum(D,Lc,E).
restnum(C,[ ],C).           % end of number if C is not a digit

gettoken(C,T,E) :- lower(C), getch(D), restid(D,Lc,E),
                           name(Id,[C|Lc]), (reswd(Id),T=Id ; T=ide(Id)).
restid(C,[C|Lc],E) :- idchar(C), getch(D), restid(D,Lc,E).
restid(C,[ ],C).            % end of identifier if C is not an id character
```

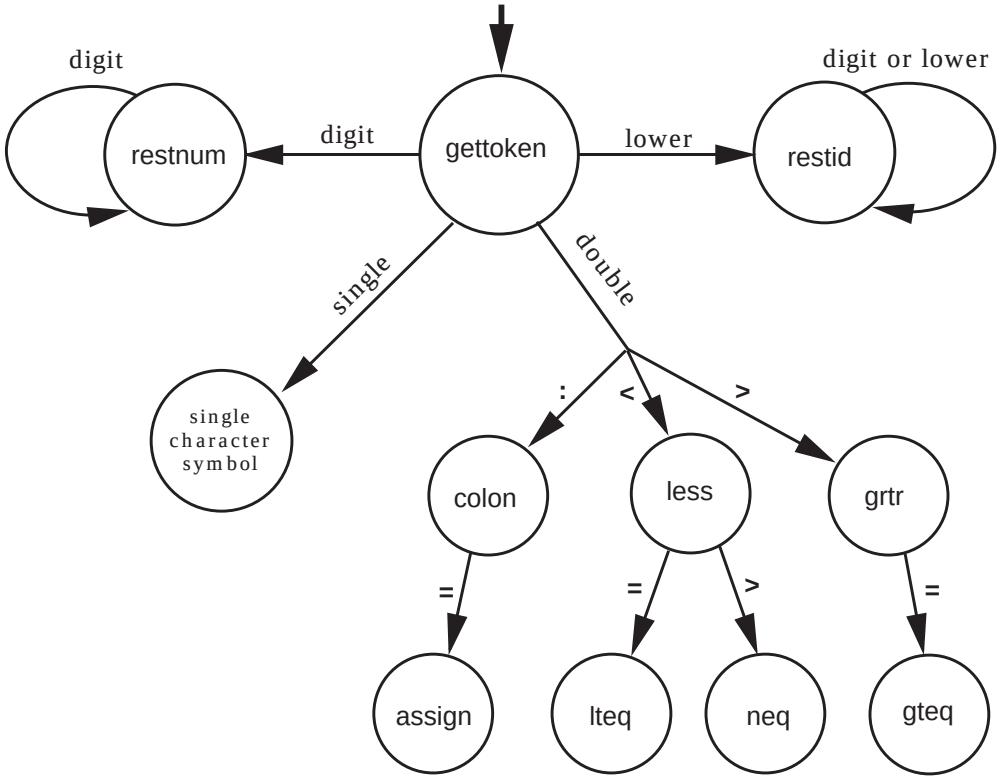


Figure 2.3: Classifying Tokens

```
gettoken(C,T,D) :- single(C,T), getch(D).
gettoken(C,T,E) :- double(C,U), getch(D), (pair(C,D,T),getch(E) ; T=U,E=D).
gettoken(C,eop,0) :- endfile(C).
gettoken(C,T,E) :- whitespace(C), getch(D), gettoken(D,T,E).
gettoken(C,T,E) :- write('Illegal character: '), put(C), nl, abort.
```

Single-character symbols are handled directly, while the two-character symbols require a decision guarded by pair. If pair succeeds, a new lookahead character is needed; otherwise, the token name is taken from the double predicate and the original lookahead character is used. When an end-of-file character occurs as the lookahead character, the token eop is returned. The predicate gettoken also allows for whitespace to be skipped, as seen in the next to the last clause. Finally, any illegal characters are trapped in the last clause, causing the scanner to abort with an error message.

To make the scanner easy to use, we define a predicate go that requests the name of the text file containing the Wren program to be scanned and invokes the scanner. Notice how it opens the text file for reading and closes it after

the scanning is complete. The list of tokens is displayed by means of the predefined predicate write.

```
go :-  nl, write('>>> Scanning Wren <<<'), nl, nl,
       write('Enter name of source file: '), nl, getfilename(fileName), nl,
       see(fileName), scan(Tokens), seen, write('Scan successful'), nl, nl,
       write(Tokens), nl.
```

The predicate for reading the file name is patterned on the code for scanning a numeral or an identifier. A priming read (get0) is followed by a predicate that accumulates a list of ascii values for the characters in the file name. We permit both uppercase and lowercase letters as well as digits, period, and slash in our file names. That enables the scanner to handle file names such as “gcd.wren” and “Programs/Factorial”. Other symbols may be added at the user’s discretion.

```
getfilename(W) :- get0(C), restfilename(C,Cs), name(W,Cs).
restfilename(C,[C|Cs]) :- filechar(C), get0(D), restfilename(D,Cs).
restfilename(C,[ ]).

filechar(C) :- lower(C) ; upper(C) ; digit(C) ; period(C) ; slash(C).
```

The transcript at the beginning of this chapter shows an execution of the scanner on a Wren program.

Exercises

1. Modify the scanner for Wren so that it accepts and recognizes the following classes of tokens:

   a) Character strings of the form "abcde".

   b) Character constants of the form 'a' or #\a.

   c) Fixed point numerals of the form 123.45.

2. Change the scanner for Wren so that “/=” is recognized instead of “<>”.

3. Change the scanner for Wren so that “<=” and “>=” can also be entered as “=<” and “=>”.

4. Add a repeat-until command to Wren and change the scanner appropriately.

5. Write a scanner for English using the alphabet of uppercase and lowercase letters and the following punctuation symbols: period, comma, ques-

tion mark, semicolon, colon, and exclamation. Each word and punctuation symbol in the text will be a token in the scanner.

6. Write a scanner that constructs a token list for a Roman numeral. Ignore any characters that are not part of the Roman numeral.

7. Write a scanner for the language of list expressions described in exercise 9 at the end of section 1.2.

2.2 LOGIC GRAMMARS

The parser forms the second part of the front end of our language processing system. It receives a token list from the scanner, and, following the BNF definition of the language, produces an abstract syntax tree. Prolog provides a mechanism, **definite clause grammars** , that makes the parser particularly easy to construct. Although the resulting system cannot compare in efficiency to present-day compilers for parsing, these grammars serve admirably for our prototype systems. Definite clause grammars are also called **logic grammars** , and we use these terms interchangeably.

Concrete Syntax

<sentence> ::= <noun phrase> <verb phrase> **.**

<noun phrase> ::= <determiner> <noun>

<verb phrase> ::= <verb> | <verb> <noun phrase>

<determiner> ::= **a** | **the**

<noun> ::= **boy** | **girl** | **cat** | **telescope** | **song** | **feather**

<verb> ::= **saw** | **touched** | **surprised** | **sang**

Abstract Syntax

Sentence ::= NounPhrase Predicate

NounPhrase ::= Determiner Noun

Predicate ::= Verb | Verb NounPhrase

Determiner ::= **a** | **the**

Noun ::= **boy** | **girl** | **cat** | **telescope** | **song** | **feather**

Verb ::= **saw** | **touched** | **surprised** | **sang**

Figure 2.4: An English Grammar

First, we motivate and explain the nature of parsing in Prolog with an example based on a subset of the English grammar found in Figure 1.1. To

simplify the problem, we consider an English grammar without prepositional phrases. The BNF and abstract syntax are displayed in Figure 2.4. The abstract syntax closely resembles the concrete syntax with a slight change in names for syntactic categories and the deletion of the period.

Given a string from the language, say "**the girl sang a song.** ", our goal is to construct an abstract syntax tree exhibiting the structure of this sentence—for example, the tree in Figure 2.5. This abstract syntax tree is quite similar to a derivation tree for the sentence.

Since we plan to carry out the parsing in Prolog, the resulting abstract syntax tree will be represented as a Prolog structure, with function symbols used to tag the syntactic categories:

```
sent(nounph(det(the), noun(girl)), pred(verb(sang), nounph(det(a), noun(song)))).
```

Observe how nicely Prolog describes a tree structure in a linear form. Recall that we view the front end for our English language grammar as a two-step process: the scanner takes a string of characters, "**the girl sang a song.** ", and creates the token list [the, girl, sang, a, song, '.']; and the parser takes the token list and constructs an abstract syntax tree as a Prolog structure, such as the one above.

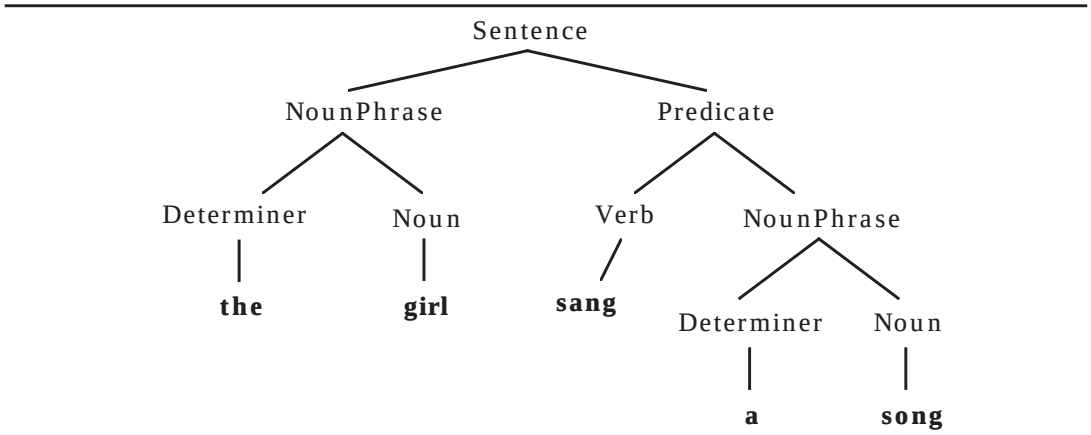


Figure 2.5: An Abstract Syntax Tree for "**the girl sang a song.** "

Motivating Logic Grammars

Although logic grammars in Prolog can be used without understanding how they work, we choose to explain their mechanism. The reader who wants to ignore this topic may skip to the subsection **Prolog Grammar Rules**.

Assume that the token list [the, girl, sang, a, song, '.'] has been generated by the scanner. The logic programming approach to analyzing a sentence according to a grammar can be seen in terms of a graph whose edges are labeled by the tokens that are terminals in the language.

Two terminals are contiguous in the original string if they share a common node in the graph.

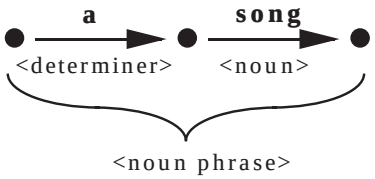
A sequence of contiguous labels constitutes a nonterminal if it corresponds to the right-hand side of a production rule for that nonterminal in the BNF grammar. For example, the three productions

<determiner> ::= **a**,

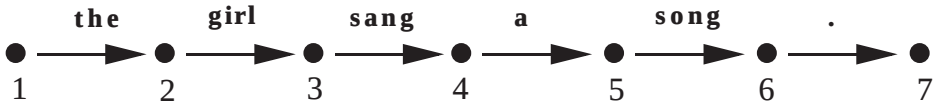
<noun> ::= **song**, and

<noun phrase> ::= <determiner> <noun>

tell us that "a song" can serve as a noun phrase. Since these two terminals lie next to each other in the graph, we know they constitute a noun phrase.



To enable these conditions to be expressed in logic, we give each node in the graph an arbitrary label—for example, using positive integers.



A predicate nounPhrase(K,L) is defined as asserting that the path from node K to node L can be interpreted as an instance of the nonterminal <noun phrase>. For example, nounPhrase(4,6) holds because edge <4,5> is labeled by a determiner **a** and edge <5,6> is labeled by a noun **song**.

The appropriate rule for <noun phrase> is

```
nounPhrase(K,L) :- determiner(K,M), noun(M,L).
```

The common variable M makes the two edges contiguous. The complete BNF grammar written in logic is listed in Figure 2.6.

```
sentence(K,L) :- nounPhrase(K,M), predicate(M,N), period(N,L).
nounPhrase(K,L) :- determiner(K,M), noun(M,L).
predicate(K,L) :- verb(K,M), nounPhrase(M,L).
predicate(K,L) :- verb(K,L).
determiner(K,L) :- a(K,L).
determiner(K,L) :- the(K,L).
noun(K,L) :- boy(K,L).
noun(K,L) :- girl(K,L).
noun(K,L) :- cat(K,L).
noun(K,L) :- telescope(K,L).
noun(K,L) :- song(K,L).
noun(K,L) :- feather(K,L).
verb(K,L) :- saw(K,L).
verb(K,L) :- touched(K,L).
verb(K,L) :- surprised(K,L).
verb(K,L) :- sang(K,L).
```

Figure 2.6: Parsing in Prolog

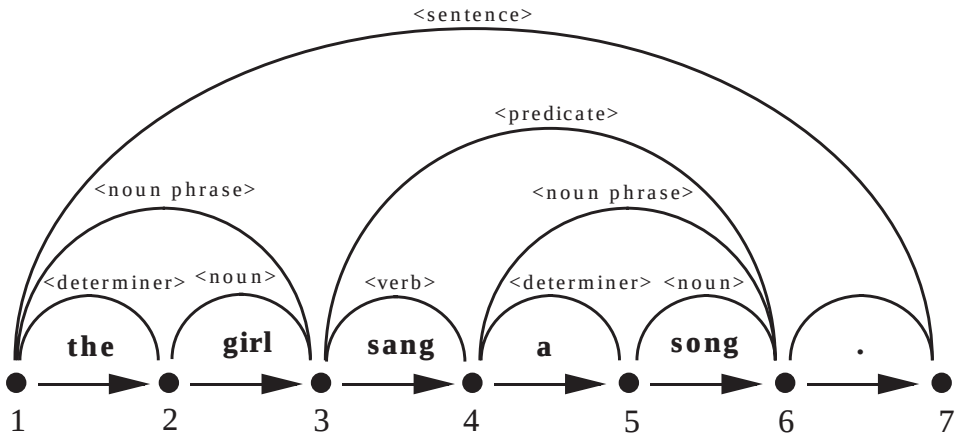
The graph for the sentence "**the girl sang a song.** " can be created by entering the following facts:

```
the(1,2).        girl(2,3).
sang(3,4).       a(4,5).
song(5,6).       period(6,7).
```

The syntactic correctness of the sentence, "**the girl sang a song.** " can be determined by either of the following queries:

```
?- sentence(1,7).
yes
?- sentence(X,Y).
X = 1
Y = 7
yes
```

The sentence is recognized by the logic program when paths in the graph corresponding to the syntactic categories in the grammar are verified as building an instance of the nonterminal <sentence>.



Note the similarity of the structure exhibited by the paths in the graph with the derivation tree for the sentence.

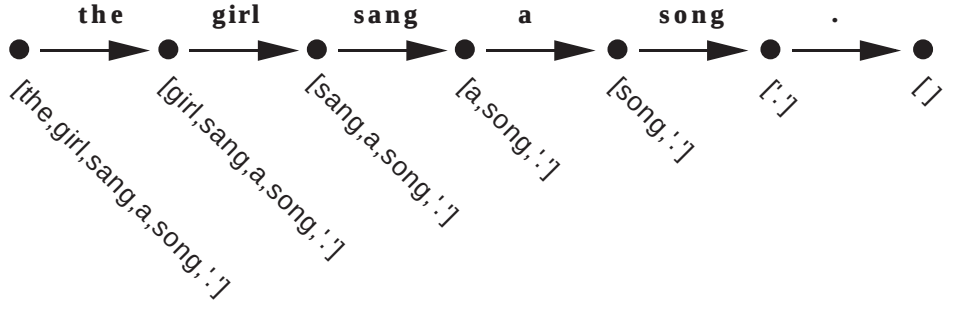
Improving the Parser

Two problems remain before this parser will be easy to use:

1. Entering the graph using predicates the(1,2), girl(2,3), … is awkward since the scanner produces only a list of tokens—namely, [the,girl,sang a, song,'.'].

2. So far the logic program recognizes only a syntactically valid sentence and does not produce a representation of the abstract syntax tree for the sentence.

The first problem can be solved and the logic program simplified by using sublists of the token list to label the nodes of the graph. These lists are called **difference lists** since the difference between two adjacent node labels is the atom that labels the intervening edge.



In general, a difference list is a Prolog structure consisting of two Prolog lists, with possibly uninstantiated components, having the property that the second list is or can be a suffix of the first one. Together they represent those items in the first list but not in the second list. For example,

difflist([a,b,c,d],[c,d]) represents the list [a,b], and

difflist([a,b,c|T],T) represents the list [a,b,c].

The concatenation of difference lists can be performed in constant time under certain conditions. Therefore many algorithms have very efficient versions using difference lists. For more on this technique of programming in Prolog, see the further readings at the end of the chapter.

The next version of the grammar exploits the same definitions for sentence, nounPhrase, and predicate, but it handles the tokens using a predicate 'C' (for "connect"), which is predefined in most Prolog implementations. The query 'C'(K,boy,L) succeeds if the edge joining the nodes K and L is labeled by the token boy. Figure 2.7 gives the Prolog code for the improved parser. The variables in the Prolog code stand for difference lists now instead of natural numbers.

```
sentence(K,L) :- nounPhrase(K,M), predicate(M,R), 'C'(R,'.',L).
nounPhrase(K,L) :- determiner(K,M), noun(M,L).
predicate(K,L) :- verb(K,M), nounPhrase(M,L).
predicate(K,L) :- verb(K,L).
determiner(K,L) :- 'C'(K,a,L).
determiner(K,L) :- 'C'(K,the,L).
noun(K,L) :- 'C'(K,boy,L).
noun(K,L) :- 'C'(K,girl,L).
noun(K,L) :- 'C'(K,cat,L).
noun(K,L) :- 'C'(K,telescope,L).
noun(K,L) :- 'C'(K,song,L).
noun(K,L) :- 'C'(K,feather,L).
verb(K,L) :- 'C'(K,saw,L).
verb(K,L) :- 'C'(K,touched,L).
verb(K,L) :- 'C'(K,surprised,L).
verb(K,L) :- 'C'(K,sang,L).
'C'([H|T],H,T).      % Edge from node [H|T] to node T is labeled with atom H
```

Figure 2.7: Improved Parsingin Prolog

An advantage of this approach is that the graph need not be explicitly created when this representation is employed. The syntactic correctness of the sentence "**the girl sang a song.** " can be recognized by the following query:

```
?- sentence([the,girl,sang,a,song,'.'],[ ]).
yes
```

The parsing succeeds because the node labeled with [the,girl,sang,a,song,'.'] can be joined to the node labeled with [] by a path representing the sentence predicate. Now the parsing query fits the scanner, since the arguments to sentence are the token list and the tail that remains when the tokens in the sentence are consumed.

By exploiting the invertibility of logic programming, it is possible to use the logic grammar to generate sentences in the language with the following query:

```
?- sentence(S, [ ]).
S = [a,boy,saw,a,boy,.] ;
S = [a,boy,saw,a,girl,.] ;
S = [a,boy,saw,a,cat,.] ;
S = [a,boy,saw,a,telescope,.] ;
S = [a,boy,saw,a,song,.] ;
S = [a,boy,saw,a,feather,.] ;
S = [a,boy,saw,the,boy,.] ;
S = [a,boy,saw,the,girl,.] ;
S = [a,boy,saw,the,cat,.]
yes
```

Using semicolons to resume the inference engine, we initiate the construction of all the sentences defined by the grammar. If the grammar contains a recursive rule, say with the conjunction **and**,

NounPhrase ::= Determiner Noun
| Determiner Noun **and** NounPhrase,

then the language allows infinitely many sentences, and the sentence generator will get stuck with ever-lengthening nounPhrase phrases, such as "**a boy saw a boy.** ", "**a boy saw a boy and a boy.** ", "**a boy saw a boy and a boy and a boy.** ", and so on.

Prolog Grammar Rules

Most implementations of Prolog have a preprocessor that translates special grammar rules into regular Prolog clauses that allow the recognition of correct sentences as seen above. The BNF definition of the English subset takes

the form of the logic grammar in Prolog shown in Figure 2.8. Logic grammars use a special predefined infix predicate "-->" to force this translation into normal Prolog code.

```
sentence --> nounPhrase, predicate, ['.'].
nounPhrase --> determiner, noun.
predicate --> verb, nounPhrase.
predicate --> verb.
determiner --> [a].
determiner --> [the].
noun --> [boy] ; [girl] ; [cat] ; [telescope] ; [song] ; [feather].
verb --> [saw] ; [touched] ; [surprised] ; [sang].
```

Figure 2.8: A Logic Grammar

The similarity between Figure 2.8 and the concrete syntax (BNF) in Figure 2.1 demonstrates the utility of logic grammars. Note that terminal symbols appear exactly as they do in the source text, but they are placed inside brackets. Since they are Prolog atoms, tokens starting with characters other than lowercase letters must be delimited by apostrophes. The Prolog interpreter automatically translates these special rules into normal Prolog clauses identical to those in Figure 2.7. Each predicate is automatically given two parameters in the translation. For example, the logic grammar clauses are translated as shown in the examples below:

```
nounPhrase --> determiner, noun.
                    becomes  nounPhrase(K,L) :- determiner(K,M),noun(M,L).
predicate --> verb. becomes  predicate(K,L) :- verb(K,L).
noun --> [boy].     becomes  noun(K,L) :- 'C'(K,boy,L).
```

Since a Prolog system generates its own variable names, listing the translated code is unlikely to show the names K, L, and M, but the meaning will be the same.

Parameters in Grammars

The second problem, that of producing an abstract syntax tree as a sentence is parsed, can be handled by using parameters in the logic grammar rules. Predicates defined by using Prolog grammar rules may have arguments in addition to the implicit ones created by the preprocessor. These additional arguments are usually inserted by the translator in front of the implicit arguments. (Some Prolog implementations insert the additional arguments after the implicit ones.)

For example, the grammar rule

```
sentence(sent(N,P)) --> nounPhrase(N), predicate(P), ['.'].
```

will be translated into the normal Prolog clause

```
sentence(sent(N,P),K,L) :- nounPhrase(N,K,M), predicate(P,M,R), 'C'(R,'.',L).
```

Figure 2.9 presents the complete BNF grammar with parameters added to build a derivation tree.

```
sentence(sent(N,P)) --> nounPhrase(N), predicate(P), ['.'].
nounPhrase(nounph(D,N)) --> determiner(D), noun(N).
predicate(pred(V,N)) --> verb(V), nounPhrase(N).
predicate(pred(V)) --> verb(V).
determiner(det(a)) --> [a].
determiner(det(the)) --> [the].
noun(noun(boy)) --> [boy].
noun(noun(girl)) --> [girl].
noun(noun(cat)) --> [cat].
noun(noun(telescope)) --> [telescope].
noun(noun(song)) --> [song].
noun(noun(feather)) --> [feather].
verb(verb(saw)) --> [saw].
verb(verb(touched)) --> [telescope].
verb(verb(surprised)) --> [surprised].
verb(verb(sang)) --> [sang].
```

Figure 2.9: A Logic Grammar with Parameters

A query with a variable representing an abstract syntax tree produces that tree as its answer:

```
?- sentence(Tree, [the,girl,sang,a,song,'.'], []).
Tree = sent(nounph(det(the), noun(girl)),
            pred(verb(sang), nounph(det(a), noun(song))))
yes
```

A subphrase can be parsed as well.

```
?- predicate(Tree, [sang,a,song], []).
Tree = pred(verb(sang), nounph(det(a), noun(song)))
yes
```

Executing Goals in a Logic Grammar

Prolog terms placed within braces in a logic grammar are not translated by the preprocessor. They are executed as regular Prolog clauses unchanged. For example, the first clause in the English grammar can be written

```
sentence(S) --> nounPhrase(N), predicate(P), ['.'], {S=sent(N,P)}.
```

The resulting Prolog clause after translation is

```
sentence(S,K,L) :-
        nounPhrase(N,K,M), predicate(P,M,R), 'C'(R,'.',L), S=sent(N,P).
```

As a second example, we add a word-counting facility to the English grammar in Figure 2.9 (only those clauses that need to be changed are shown):

```
sentence(WC,sent(N,P)) -->
        nounPhrase(W1,N), predicate(W2,P), ['.'], {WC is W1+W2}.

nounPhrase(WC,nounph(D,N)) --> determiner(D), noun(N), {WC is 2}.

predicate(WC,pred(V,N)) --> verb(V), nounPhrase(W,N), {WC is W+1}.

predicate(1,pred(V)) --> verb(V).
```

If the word-counting feature is used, conditions may be placed on the sentences accepted by the grammar; for example, if only sentences with no more than ten words are to be accepted, the first clause can be written

```
sentence(WC,sen(N,P)) -->
        nounPhrase(W1,N), predicate(W2,P), ['.'], {WC is W1+W2, WC <= 10}.
```

Any sentence with more than ten words will fail to parse in this augmented grammar because of the condition. Computing values and testing them illustrates the basic idea of attribute grammar, the subject of the next chapter.

The astute reader may have noticed that in the English grammar in this chapter, each sentence has exactly five words. The condition on word count makes more sense if applied to a grammar that includes prepositional phrases or allows **and**'s in the <noun phrase> strings.

Exercises

1. Write a definite clause grammar for an English grammar that includes prepositional phrases as in Chapter 1. To avoid ambiguity, add prepositional phrases only to noun phrases.

2. Modify <noun phrase> to allow **and** according to the productions in Figure 2.4. Construct a logic grammar, and try to generate all the sentences. Make sure you recognize the shortest noun phrase first.

3. This grammar is a BNF specification of the language of Roman numerals less than 500.

   <roman> ::= <hundreds> <tens> <units>

   <hundreds> ::= <empty> | **C** | **CC** | **CCC** | **CD**

   <tens> ::= <low tens> | **XL** | **L** <low tens> | **XC**

   <low tens> ::= <empty> | <low tens> **X**

   <units> ::= <low units> | **IV** | **V** <low units> | **IX**

   <low units> ::= <empty> | <low units> **I**

   Write a logic grammar that parses strings in this language and also enforces a constraint that the number of X's in <low tens> and I's in <low units> can be no more than three.

4. Write a logic grammar for the language of list expressions described in exercise 9 in section 1.2.

2.3 PARSING WREN

Prolog's definite clause grammars provide a mechanism for parsing Wren as well as our English language fragment. Again, we start with the BNF specification of Wren's concrete syntax and convert the productions into logic grammar clauses with as few changes as required. Parameters to the clauses construct an abstract syntax tree for the program being parsed.

We illustrate the process with a couple of the straightforward productions.

<program> ::= **program** <identifier> **is** <block>

becomes

```
program(AST) --> [program], [ide(I)], [is], block(AST).
```

and

<block> ::= <declaration seq> **begin** <command seq> **end**

becomes

```
block(prog(Decs,Cmds)) --> decs(Decs), [begin], cmds(Cmds), [end].
```

Observe that the reserved words and identifiers are recognized as Prolog atoms and ide structures inside brackets. The logic grammar needs to match the form of the tokens produced by the scanner. Also, note how the abstract syntax tree (AST) for a block is constructed from the two subtrees for declarations and commands.

The BNF specification for commands can be converted into logic grammar clauses with little modification.

<command> ::= <variable> **:=** <expr>

becomes

```
command(assign(V,E)) --> [ide(V)], [assign], expr(E).
```

and

<command> ::= **while** <boolean expr> **do** <command seq> **end while**

becomes

```
command(while(Test,Body)) -->
                    [while], boolexpr(Test), [do], cmds(Body), [end, while].
```

Parsing Wren involves collecting lists of items for several of its syntactic categories: command sequences, declaration sequences, and lists of variables. We describe the pattern for handling these lists by means of command sequences and leave the other two as exercises. Our approach follows the strategy for building a list in the scanner—that is, we obtain the first object in the list and then call a predicate to construct the (possibly empty) tail of the list. In each case, we use Prolog lists for the representation of the subtrees in the abstract syntax tree.

The productions

<command seq> ::= <command> | <command> **;** <command seq>

become the two predicates

```
cmds(Cmds) --> command(Cmd), restcmds(Cmd,Cmds).
restcmds(Cmd,[Cmd|Cmds]) --> [semicolon], cmds(Cmds).
restcmds(Cmd,[Cmd]) --> [ ].
```

A variable list can be formed in exactly the same way; but remember that declaration sequences may be empty, thereby producing an empty list [ ] as the abstract syntax subtree.

Handling Left Recursion

In defining the syntax of programming languages, BNF specifications frequently use left recursion to define lists and expressions; in fact, expressions with left associative operations are naturally formulated using left recursion. Unfortunately, parsing left recursion can lead the interpreter down an infinite branch of the search tree in the corresponding logic program.

As an example, consider a language of expressions with left associative addition and subtraction of numbers:

<expr> ::= <expr> <opr> <numeral>

<expr> ::= <numeral>

<opr> ::= **+** | **–**

<numeral> ::= ... % as before

Using a Prolog definite clause grammar produces the following rules:

```
expr(plus(E1,E2)) --> expr(E1), ['+'], [num(E2)].
expr(minus(E1,E2)) --> expr(E1), ['–'], [num(E2)].
expr(E) --> [num(E)].
```

which translate into the following Prolog clauses:

```
expr(plus(E1,E2),K,L) :- expr(E1,K,M), 'C'(M,'+',N), 'C'(N,num(E2),L).
expr(minus(E1,E2),K,L) :- expr(E1,K,M), 'C'(M,'–',N), 'C'(N,num(E2),L).
expr(E,K,L) :- 'C'(K,num(E),L).
```

Suppose the string “5–2” runs through the scanner, and the logic grammar is invoked with the query

```
?- expr(E, [num(5), '–', num(2)], [ ]).
```

The Prolog interpreter repeatedly tries expr with an uninstantiated variable as the first argument, creating an endless search for a derivation, as shown in Figure 2.10.

The depth-first strategy for satisfying goals makes it impossible for Prolog to find the consequence defined by the logic program. The logic interpreter needs to satisfy the initial goal in the goal list first. The usual way to remove left recursion from a BNF grammar is to define a new syntactic category that handles all but the first token:

<expr> ::= <numeral> <rest of expr>

<rest of expr> ::= <opr> <numeral> <rest of expr>

<rest of expr> ::= ε

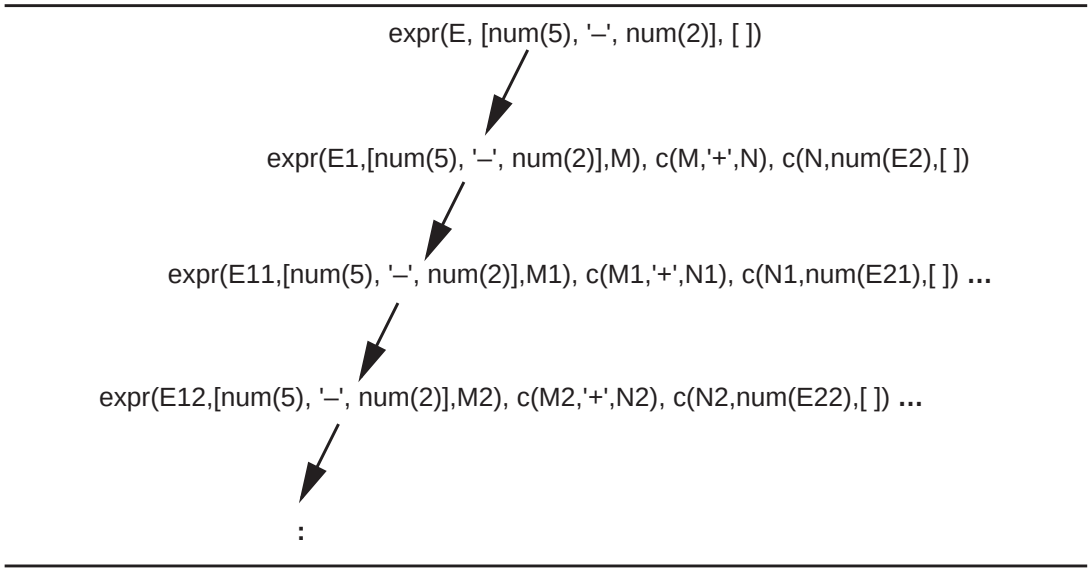


Figure 2.10: Parsing the expression "5–2"

The corresponding logic grammar has the property that each rule starts with a goal that can be verified without going down an infinite branch of the search tree. A careful definition of the parameters enables the grammar to construct a left associative parse tree even though the logic grammar is right recursive.

```
expr(E) --> [num(E1)], restexpr(E1,E).

restexpr(E1,E) --> ['+'], [num(E2)], restexpr(plus(E1,E2),E).

restexpr(E1,E) --> ['–'], [num(E2)], restexpr(minus(E1,E2),E).

restexpr(E,E) --> [ ].
```

The predicate restexpr(E1,E) means that the expression E1 has been constructed from the symbols encountered so far, and the resulting parse tree will be E once the rest of the symbols making up the expression have been processed. The last rule "restexpr(E,E) --> [ ]." states that when no tokens are left to build an expression, the result is the expression created so far—namely, the first argument.

For Wren, we use logic grammar clauses

```
expr(E) --> intexpr(E).
expr(E) --> boolexpr(E).
```

```
intexpr(E) --> term(T), restintexpr(T,E).
restintexpr(T,E) --> weakop(Op), term(T1), restintexpr(exp(Op,T,T1),E).
restintexpr(E,E) --> [ ].
term(T) --> element(P), restterm(P,T).
restterm(P,T) --> strongop(Op), element(P1), restterm(exp(Op,P,P1),T).
restterm(T,T) --> [ ].
element(num(N)) --> [num(N)].
element(Ide(I)) --> [ide(I)].
weakop(plus) --> [plus].          weakop(minus) --> [minus].
strongop(times) --> [times].      strongop(divides) --> [divides].
comparison(bexp(R,E1,E2)) --> intexpr(E1), rel(R), intexpr(E2).
rel(equal) --> [equal].   rel(neq) --> [neq].     rel(less) --> [less].
rel(grtr) --> [grtr].     rel(gteq) --> [gteq].   rel(lteq) --> [lteq].
```

following the pattern shown above for integer expressions. Many of the BNF rules translate directly into logic grammar clauses. For example, the BNF productions for handling parentheses and unary minus in integer expressions,

<element> ::= **(** <integer expr> **)**

<element> ::= **–** <element>

become the logic grammar clauses,

```
element(E) --> [lparen], intexpr(E), [rparen].
element(minus(E)) --> [minus], element(E).
```

Note that we can use the same function symbol minus for both unary minus and subtraction since the number of parameters help define the structure. Boolean expressions are handled in a similar manner.

Recall that we suggested two different formats for the abstract syntax of expressions formed from binary operations:

```
exp(plus,E1,E2)
```

and

```
plus(E1,E2).
```

The choice between these two templates is largely subjective, depending on the purpose for which the trees will be used. We elect to use the exp(plus,E1,E2) format when we develop an interpreter for Wren in later chapters because it eases the handling of arithmetic expressions. In this chapter we have used both formats to emphasize that the difference is primarily cosmetic.

Left Factoring

Sometimes two productions have a common initial string of terminals and nonterminals to be processed. If the first production fails, the second one has to recognize that initial string all over again. Factoring the initial string as a separate rule leads to a more efficient parsing algorithm.

Suppose now that expressions have right associative operators at two precedence levels:

```
expr(plus(E1,E2)) --> term(E1), ['+'], expr(E2).
expr(minus(E1,E2)) --> term(E1), ['–'], expr(E2).
expr(E) --> term(E).
term(times(T1,T2)) --> [num(T1)], ['*'], term(T2).
term(divides(T1,T2)) --> [num(T1)], ['/'], term(T2).
term(T) --> [num(T)].
```

The problem here is that when processing a string such as "2*3*4*5*6 – 7", the term "2*3*4*5*6" must be recognized twice, once by the first clause that expects a plus sign next, and once by the second clause that matches the minus sign. This inefficiency is remedied by rewriting the grammar as follows:

```
expr(E) --> term(E1), restexpr(E1,E).
restexpr(E1,plus(E1,E2)) --> ['+'], expr(E2).
restexpr(E1,minus(E1,E2)) --> ['–'], expr(E2).
restexpr(E,E) --> [ ].
term(T) --> [num(T1)], restterm(T1,T).
restterm(T1,times(T1,T2)) --> ['*'], term(T2).
restterm(T1,divides(T1,T2)) --> ['/'], term(T2).
restterm(T,T) --> [ ].
```

Now the term "2*3*4*5*6" will be parsed only once.

Left factoring can also be used in processing the **if** commands in Wren.

<command> ::= **if** <boolean expr> **then** <command seq> **end if**
| **if** <boolean expr> **then** <command seq> **else** <command seq> **end if**

becomes

```
command(Cmd) -->
        [if], boolexpr(Test), [then], cmds(Then), restif(Test,Then,Cmd).
```

```
restif(Test,Then,if(Test,Then,Else)) --> [else], cmds(Else), [end], [if].
restif(Test,Then,if(Test,Then)) --> [end], [if].
```

Observe that we construct either a ternary structure or a binary structure for the command, depending on whether we encounter **else** or not. Again, we use a predicate go to control the system:

```
go :- nl,write('>>> Interpreting: Wren <<<'), nl, nl,
      write('Enter name of source file: '), nl, getfilename(fileName), nl,
      see(fileName), scan(Tokens), seen, write('Scan successful'), nl, !,
      write(Tokens), nl, nl,
      program(AST,Tokens,[eop]), write('Parse successful'), nl, !,
      write(AST), nl, nl.
```

Note that cut operations "!" have been inserted after the scanning and parsing phases of the language processing. This ensures that the Prolog interpreter never backtracks into the parser or scanner after each has completed successfully. Such backtracking can only generate spurious error messages. A cut acts as a one-way passage. It always succeeds once, but if the backtracking attempts the cut a second time, the entire query fails. Except for the go clause, we refrain from using cuts in our Prolog code because we want to avoid their nonlogical properties. See the references for details on the cut operation.

Exercises

1. Write the logic grammar clauses that parse declaration sequences and variable lists.

   <declaration seq> ::= ε | <declaration> <declaration seq>

   <declaration> ::= **var** <variable list> **:** <type> **;**

   <type> ::= **integer** | **boolean**

   <variable list> ::= <variable> | <variable> **,** <variable list>

2. Write the logic grammar clauses that parse Boolean expressions. Use the tag bexp for these expressions.

   <boolean expr> ::= <boolean term>
   | <boolean expr> **or** <boolean term>
   <boolean term> ::= <boolean element>
   | <boolean term> **and** <boolean element>

<boolean element> ::= **true** | **false** | <variable> | <comparison>
| **not (** <boolean expr> **)** | **(** <boolean expr> **)**

3. Add these language constructs to Wren and modify the parser to handle them:

   a) repeat-until commands
   <command> ::= ... | **repeat** <command seq> **until** <boolean expr>

   b) conditional expressions
   <expression> ::= ...
   | **if** <boolean expr> **then** <integer expr> **else** <integer expr>

   c) expressions with side effects
   <expression> ::= ... | **begin** <command seq> **return** <expr> **end**

2.4 FURTHER READING

Many books provide a basic introduction to Prolog. Our favorites include the classic textbook by Clocksin and Mellish that is already in its third edition [Clocksin87]; Ivan Bratko's book [Bratko90], which emphasizes the use of Prolog in artificial intelligence; and the comprehensive text by Sterling and Shapiro [Sterling94]. These books also include descriptions of the operational semantics of Prolog with information on unification, the resolution proof strategy, and the depth-first search method used by Prolog. The last book has a good discussion of programming with difference lists in Prolog. The model for our scanner can be found in the Clocksin and Mellish text where the lexical analysis of an English sentence is presented.

Most Prolog texts cover the definite clause grammars that we used to build the parsers in this chapter. In addition to [Clocksin87] and [Sterling86], see [Kluzniak85], [Malpas87], [Covington88], and [Saint-Dizier90] for material on logic grammars.

The roots of Prolog lie in language processing. It has been said that Prolog was invented by Robert Kowalski in 1974 and implemented by Alain Colmerauer in 1973. To explain this apparent contradiction, we note that Prolog originated in Colmerauer's interest in using logic to express grammar rules and to formalize the parsing of natural language sentences. He developed the mechanism of syntactic analysis in logic before the power of Prolog as a general purpose programming language was made apparent by Kowalski. For more information on the early development of Prolog and logic grammars see [Colmerauer78] and [Kowalski79].

Some of the issues discussed in this chapter, such as left recursion and left factoring, are handled in compiler writing texts (see [Aho86] and [Parsons92]). Prolog was given credibility as a vehicle for language processing in 1980 by David Warren in a paper that describes a compiler written in Prolog [Warren80].