Chapter 4
TWO-LEVEL GRAMMARS

We used attributes in Chapter 3 to augment a context-free grammar in order to verify context sensitivity. This chapter will focus on two-level grammars, another formal technique that starts with a context-free grammar, augmenting it with a higher-level grammar to test context-sensitive properties.

A BNF grammar involves a finite number of terminal symbols and production rules. We think of BNF as a "one-level grammar". The new approach of introducing "rules about rules" is called a "two-level grammar". Although we still have a finite number of terminal symbols, adding a second level to the grammar can be used to generate an infinite number of production rules. Consequently, a two-level grammar is inherently more powerful than a BNF grammar. Two-level grammars are sometimes called W-grammars, named after Aad van Wijngaarden, the researcher who developed this approach. The programming language Algol68 was defined using a two-level grammar to specify its complete syntax, including context-sensitive conditions.

This chapter begins with a brief introduction to two-level grammars; then in section 4.2, we develop a context-sensitive grammar for Wren. Two-level grammars can also be extended into the realm of operational semantics by building a programming language interpreter into the grammar, but this extension is beyond the scope of this text. Interested readers can consult the references described in the further reading section of this chapter. We end the chapter by showing how small two-level grammars can be implemented in Prolog. We also discuss the relationship of two-level grammars and logic programming.

4.1 CONCEPTS AND EXAMPLES

We begin by looking at the part of a two-level grammar that is equivalent to BNF. **Protonotions** correspond to nonterminals and terminals in a BNF grammar. We use a sequence of lowercase, boldface characters to represent protonotions. Terminals are distinguished by the final word **symbol**. Spaces can occur anywhere in a protonotion and do not change its meaning. Examples of protonotions are

program is equivalent to the nonterminal <program>.

program symbol is equivalent to the keyword **program**.

The correspondence between the protonotions ending in **symbol** and the actual terminal symbols is presented in a representation table.

A grammar rule for defining protonotions is called a **hyper-rule.** The following conventions are used in hyper-rules:

- A colon separates the left- and right-hand side of a rule.
- A comma indicates the juxtaposition of protonotions.
- A semicolon indicates alternatives on the right-hand side of a rule.
- A period terminates a hyper-rule.

For example, the following hyper-rule corresponds to the productions for <element> in Wren.

element : numeral;
variable;
left par en symbol, integer expr , right par en symbol;
negation symbol, element.

A complete (context-free) grammar for Wren using two-level grammar notation is shown in Figure 4.1, with the corresponding representation table given in Figure 4.2.

Next we look at the terms and notation associated with the "second level" of two-level grammars. A **metanotion** can stand for any number of protonotions. Metanotions are written in boldface, uppercase letters with no embedded spaces. The allowed protonotions are specified in the form of **metarules**, which use the following notational conventions:

---

program : pr ogram symbol, identifier , is symbol, block.

block : declaration seq, begin symbol, command seq, end symbol.

declaration seq : empty;
declaration, declaration seq.

empty : .

declaration : var symbol, variable list, colon symbol,
type, semicolon symbol.

type : integer symbol;
boolean symbol.

variable list : variable;
variable, comma symbol, variable list.

command seq : command;
command, semicolon symbol, command seq.

Figure 4.1: Grammar for Wren Using Two-level Notation (Part 1)

```
command : variable, assign symbol, expr ession;
          read symbol, variable;
          write symbol, integer expr;
          skip symbol;
          while symbol, boolean expr , do symbol,
                                command seq, end while symbol;
          if symbol, boolean expr , then symbol,
                                command seq, end if symbol;
          if symbol, boolean expr , then symbol, command seq,
                          else symbol, command seq, end if symbol.
expression : integer expr;
             boolean expr .

integer expr : term;
               integer expr , weak op, ter m.

term : element;
       term, strong op, element.

element : numeral;
          variable;
          left par en symbol, integer expr , right par en symbol;
          negation symbol, element.

boolean expr : boolean ter m;
               boolean expr , or symbol, boolean ter m.

boolean ter m : boolean element;
                boolean ter m, and symbol, boolean element.

boolean element : true symbol;
                  false symbol;
                  variable;
                  comparison;
                  left par en symbol, Boolean expr , right par en symbol;
                  not symbol, left par en symbol,
                                boolean expr , right par en symbol.

comparison : integer expr , relation, integer expr .

variable : identifier .
```

Figure 4.1: Grammar for Wren Using Two-level Notation (Part 2)

- A double colon (**::**) separates the left- and right-hand sides of a metarule.
- A space indicates the juxtaposition of items.
- A semicolon indicates alternatives on the right-hand side of a metarule.
- A period terminates a metarule.

```
relation : less or equal symbol;
           less symbol;
           equal symbol;
           greater symbol;
           greater or equal symbol;
           not equal symbol.

weak op : plus symbol;
          minus symbol.

strong op : multiply symbol;
            divide symbol.

identifier : letter;
             letter, identifier;
             letter, digit.

letter : a symbol;  b symbol;  c symbol;  d symbol;  e symbol;  f symbol;
         g symbol;  h symbol;  i symbol;  j symbol;  k symbol;  l symbol;
         m symbol;  n symbol;  o symbol;  p symbol;  q symbol;  r symbol;
         s symbol;  t symbol;  u symbol;  v symbol;  w symbol;  x symbol;
         y symbol;  z symbol.

numeral : digit;
          digit, numeral.

digit : zero symbol;  one symbol;  two symbol;  three symbol;
        four symbol;  five symbol;  six symbol;  seven symbol;
        eight symbol;  nine symbol.
```

Figure 4.1: Grammar for Wren Using Two-level Notation (Part 3)

The following metarule specifies that the metanotion **ALPHA** stands for 26 alternative protonotions.

```
ALPHA :: a; b; c; d; e; f; g; h; i; j; k; l; m;
                  n; o; p; q; r; s; t; u; v; w; x; y; z.
```

A metarule can also contain metanotions on the right-hand side.

```
NOTION :: ALPHA;  NOTION ALPHA.
```

In this case a **NOTION** is any sequence of bold, lowercase letters—in other words, a protonotion. Metanotions can also appear anywhere in a hyper-rule. For example, the following hyper-rule for Wren

program symbol	program	colon symbol	:
is symbol	is	semicolon symbol	;
begin symbol	begin	comma symbol	,
end symbol	end	assign symbol	:=
var symbol	var	plus symbol	+
integer symbol	integer	minus symbol	-
boolean symbol	boolean	multiply symbol	*
skip symbol	skip	divide symbol	/
read symbol	read	negation symbol	-
write symbol	write	left par en symbol	(
while symbol	while	right par en symbol	)
do symbol	do	less or equal symbol	<=
end while symbol	end while	less symbol	<
if symbol	if	equal symbol	=
then symbol	then	greater symbol	>
else symbol	else	greater or equal symbol	>=
end if symbol	end if	not equal symbol	<>
or symbol	or	a symbol	a
and symbol	and	: :	:
not symbol	not	z symbol	z
true symbol	true	zero symbol	0
false symbol	false	: :	:
		nine symbol	9

Figure 4.2: Representation Table for Wren

letter : a symbol; b symbol; c symbol; d symbol; e symbol; f symbol;
g symbol; h symbol; i symbol; j symbol; k symbol; l symbol;
m symbol; n symbol; o symbol; p symbol; q symbol; r symbol;
s symbol; t symbol; u symbol; v symbol; w symbol;
x symbol; y symbol; z symbol.

can now be written simply as

letter: ALPHA symbol.

By introducing a metanotion, we have one hyper-rule that stands for 26 possible rules: **letter : a symbol.** , **letter : b symbol.** , and so on.

Traditionally, two-level grammars are printed in boldface. The basic terms introduced so far can be summarized as follows:

- A **protonotion** is a sequence of lowercase, boldface characters that define a nonterminal; however, if the nonterminal is followed by the protonotion **symbol**, then protonotion corresponds to a terminal in the target language as given in a representation table.
- A **metanotion** is a sequence of uppercase characters, used to represent any number of protonotions.
- A **hyper-rule** is a production rule that define a protonotion or a class of protonotions; it may contain protonotions and metanotions on the left- and right-hand sides, and substitutions of protonotions for metanotions must be consistent.
- A **metarule** is a production rule that defines the single metanotion on its left-hand side using protonotions or metanotions on its right-hand side.

As indicated at the start of this chapter, it is possible to generate an infinite number of production rules. Suppose that we want to specify a nonempty list structure as a sequence of one or more items separated by commas. Wren uses a list structure to define lists of variables:

variable list : variable; variable, comma symbol, variable list.

We can generalize this list concept by introducing the metanotion **NOTION** to provide a template for list construction:

NOTION list : NOTION; NOTION, comma symbol, NOTION list.

If **NOTION** has the value **variable**, this hyper-rule specifies our metarule from Wren. If **NOTION** takes other values, say **integer** or **character**, then we are specifying a list of integers or a list of characters. We do require *consistent substitution* of protonotions for a metanotion within a hyper-rule, thus guaranteeing that our list contains elements of the same type. It will not be possible to produce a hyper-rule such as

integer list : character; integer , comma symbol, variable list. (illegal!)

Since **NOTION** can match any protonotion (without embedded spaces), we have given a single hyper-rule that can match an infinite number of productions specifying nonempty lists containing items of the same kind.

We now introduce some notational conveniences. The metanotion **EMPTY** can stand for the empty protonotion:

EMPTY :: .

Suppose that we want to use a tally notation to define the concept of number. For example, the tally **iiiiiiii** represents the number 8. We can specify a tally by the metarule

TALLY :: i ; T ALLY i.

We cannot represent the number zero by using **TALLY**, so we use the **EMPTY** protonotion

TALLETY :: EMPTY ; TALLY.

A conventional notation in two-level grammars is to have the suffix **-ETY** allow the **EMPTY** protonotion as one of the alternatives.

A second notational convenience allows us to relax (at least notationally) the consistent substitution principle. Consider the nonempty list example again. Initially assume lists can contain integers or characters, but not both.

LISTITEM list :
LISTITEM; LISTITEM, comma symbol, LISTITEM list.
LISTITEM :: integer; character .

Now suppose we want to mix integers and characters in a single list. A possible specification is

mixed list : LISTITEM; LISTITEM1, comma symbol, mixed list.
LISTITEM1 :: LISTITEM.

The hyper-rule **mixed list** has four different possibilities:

1. **mixed list : integer; integer , comma symbol, mixed list.**
2. **mixed list : integer; character , comma symbol, mixed list.**
3. **mixed list : character; character , comma symbol, mixed list.**
4. **mixed list : character; integer , comma symbol, mixed list.**

We adopt the convention that a metanotion ending in a digit stands for the same set of protonotions as the metanotion without the digit, so we do not have to specify metarules such as **LISTITEM1 :: LISTITEM.** given above. (It should be noted that **LISTITEM1** was not strictly required to produce the mixed list since **LISTITEM** and **LISTITEM1** appear only once in different alternatives; however, the intent of this example should be clear.)

Fortran String Literals

In older versions of Fortran having no character data type, character string literals were expressed in the following format:

<string literal> ::= <numeral> H <string>

where the <numeral> is a base-ten integer (≥ 1), H is a keyword (named after Herman Hollerith), and <string> is a sequence of characters. The (context-sensitive) syntax of this string literal will be correct if the numeric value of the base-ten numeral matches the length of the string. A two-level grammar is developed for a Fortran string literal. The initial version assumes that the

numeral is a single digit from 1 to 9. The hyper-rules for the digit symbols are as follows:

i digit : digit one symbol.
ii digit : digit two symbol.
iii digit : digit thr ee symbol.
iiii digit : digit four symbol.
iiiii digit : digit five symbol.
iiiiii digit : digit six symbol.
iiiiiii digit : digit seven symbol.
iiiiiiii digit : digit eight symbol.
iiiiiiiii digit : digit nine symbol.

The string literal is a sequence of lowercase letters, which we call **LETTERSEQ** , as specified by the following metarules:

APLHA :: a; b; c; d; e; f; g; h; i; j; k; l; m;
n; o; p; q; r; s; t; u; v; w; x; y; z.
LETTER :: letter ALPHA.
LETTERSEQ :: LETTER; LETTERSEQ LETTER.

The string literal, called **hollerith** , is specified by the following hyper-rule. Notice that the consistent substitution for **TALLY**, defined previously, provides the desired context sensitivity.

hollerith : T ALLY digit, hollerith symbol, T ALLY LETTERSEQ.

Finally, we need to show how a **TALLY LETTERSEQ** decomposes. The basic idea is that every time we remove an **i** from **TALLY** we also remove a **LETTER** from **LETTERSEQ** . We must eventually reach a single **i** followed by a single **LETTER** . The following two hyper-rules express these ideas.

TALLY i LETTER LETTERSEQ : i LETTER, T ALLY LETTERSEQ.
i LETTER : LETTER symbol.

The representation table for this two-level grammar involves symbols for lowercase letters, digits, and the separator **H.**

Representation Table

letter a symbol	**a**	**digit one symbol**	**1**
: : :	**:**	**: : :**	**:**
letter z symbol	**z**	**digit nine symbol**	**9**
hollerith symbol	**H**		

Derivation Trees

Since two-level grammars are simply a variant of BNF, they support the idea of a derivation tree to exhibit the structure of a string of symbols.

Definition : A **derivation tr ee** of a string in a two-level grammar displays a derivation of a sentence in the grammar. The nodes in the tree are labeled with protonotions, and the protonotions for all leaf nodes end with **symbol**, indicating terminal symbols. Traversing the leaf nodes from left to right (a preorder traversal) and replacing the protonotion symbols with the corresponding tokens in the Representation Table produces the string being parsed. Empty leaves just disappear. ∎

The derivation tree for “3Habc” is shown in Figure 4.3.

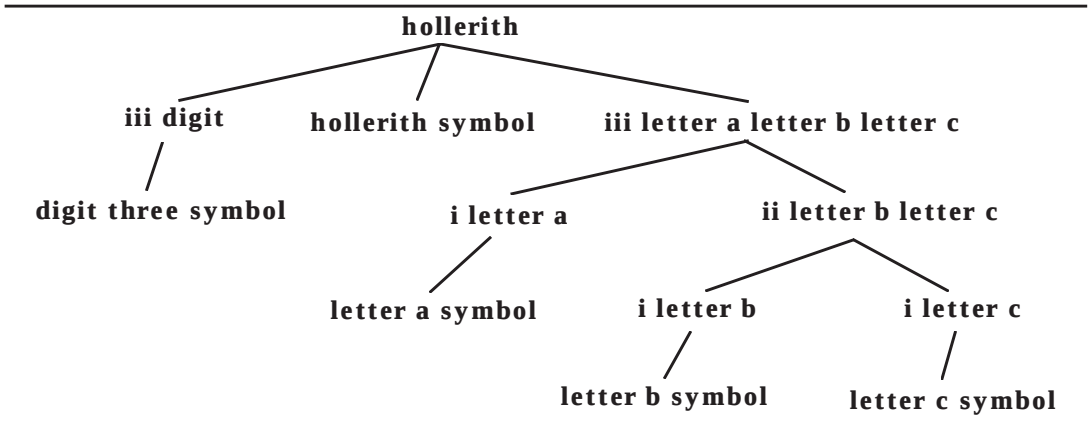

Figure 4.3: Derivation Tree for “3Habc”

When a string does not obey the context-sensitive conditions, no derivation is possible. Figure 4.4 shows an attempt to draw a derivation tree for “4Habc”, but, as indicated, it is not possible to complete the tree.

If an arbitrary numeral is allowed to the left of the **hollerith symbol** in the previous example, a mechanism is needed to transform the base-ten numeral into a tally representation. For example, if the leaves of the subtree for a numeral are **digit two symbol** followed by **digit thr ee symbol** , the **TALLY** will be **iiiiiiiiiiiiiiiiiiiiii** , a sequence of 23 **i**’s. We need to allow for a zero digit at any position, other than the leading digit, and for the corresponding empty tally.

EMPTY digit : digit zer o symbol.

TALLETY :: T ALLY; EMPTY .

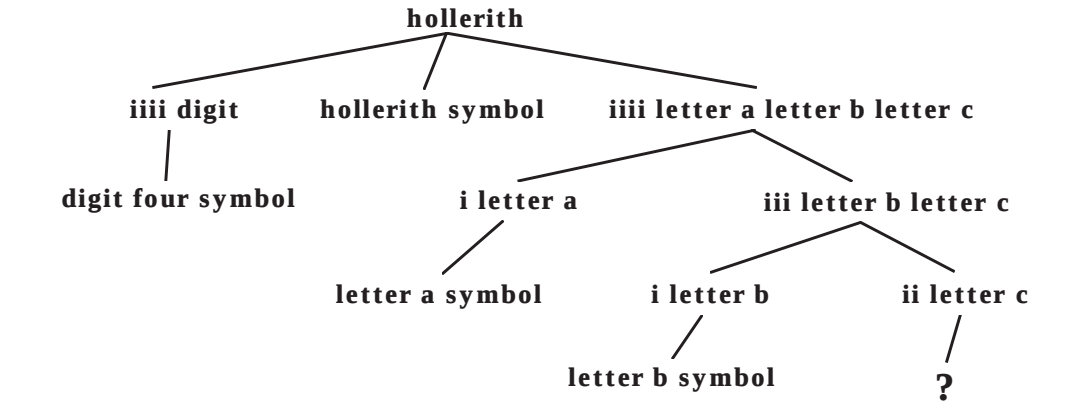

Figure 4.4: Attempted Derivation Tree for "4Habc"

We develop a hyper-rule for **TALLY constant** that captures the semantics of base-ten numerals. For a multiple digit numeral $d_1d_2 \ldots d_{n-1}d_n$, we know that

value($d_1d_2 \ldots d_{n-1}d_n$) is 10•value($d_1d_2 \ldots d_{n-1}$) + value(d_n).

In a tally numeration system, multiplying by 10 is accomplished by concatenating ten copies of the tally together. So we rewrite our equation as

tally($d_1d_2 \ldots d_{n-1}d_n$) is ten copies of tally($d_1d_2 \ldots d_{n-1}$)
followed by one copy of tally(d_n).

A **where** clause gives us a mechanism for expressing this equality, as evidenced by the following hyper-rule that includes both the base case and the general case:

TALLETY constant :
TALLETY digit;
TALLETY2 constant, T ALLETY3 digit, wher e TALLETY is
TALLETY2 T ALLETY2 T ALLETY2 T ALLETY2 T ALLETY2
TALLETY2 T ALLETY2 T ALLETY2 T ALLETY2 T ALLETY2
TALLETY3.
where TALLETY is T ALLETY : EMPTY .

The **where** clause is similar to condition checking in attribute grammars. If the condition holds, the **EMPTY** protonotion is the leaf on the tree; otherwise, the parse fails. This **where** clause is best illustrated by drawing the complete derivation tree, as shown for the numeral 23 in Figure 4.5. Notice that we rely on the fact that spaces in a protonotion do not change the protonotion.

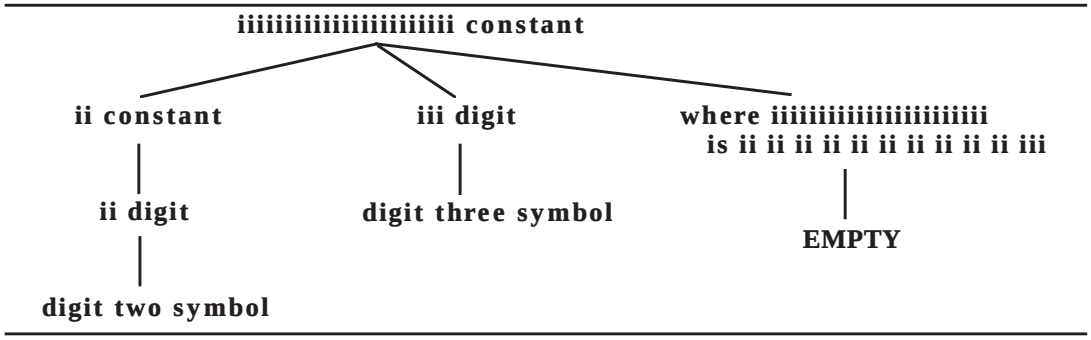

Figure 4.5: Derivation Tree for the Numeral 23

Exercises

1. Suppose that we have a programming language that allows mixed types for the arithmetic operations. Setting aside issues of precedence for the moment, we can describe expressions by the following two-level grammar:

   EXPR TYPE :: integer expr ession;
   real expr ession; complex expr ession.

   OPERATION :: addition; subtraction; multiplication; division.

   expression : EXPR TYPE, OPERA TION, EXPR TYPE.

   How many different hyper-rules are possible for expression?

2. Suppose we have the following productions:

   TYPE :: integer; r eal; complex; character; string.

   assignment : TYPE1 name, assign symbol, TYPE2 expr ession,
   where TYPE1 is assignment compatible with TYPE2.

   Write the hyper-rules for this **where** condition assuming success if the types are the same, an integer expression can be assigned to a real or complex variable, a real expression can be assigned to a complex variable, and a character can be assigned to a string variable.

3. Develop a two-level grammar that parses only strings of the form $a^n b^n c^n$. Use the consistent substitution of a metanotion into a hyper-rule to guarantee that the values of n are the same. Test the grammar by drawing the derivation tree for "**aaabbbccc**". Also show why there is no valid derivation tree for "**aabbbcc**".

4. Some implementations of Pascal limit the size of identifiers to *eight or fewer* characters. The following hyper-rule expresses this constraint:

   identifier : letter , upto iiiiiii alphanum.

Develop a general hyper-rule to express the meaning of

upto TALLY NOTION

Draw the derivation tree for the identifier “count”.

4.2 A TWO-LEVEL GRAMMAR FOR WREN

The two-level grammar that we construct for Wren performs all necessary context checking. The primary focus is on ensuring that identifiers are not multiply declared, that variables used in the program have been declared, and that their usage is consistent with their types (see Figure 1.11). All declaration information is present in a metanotion called DECLSEQ, which is associated with the context-sensitive portions of commands. We use the following Wren program for illustration as we develop the two-level grammar:

```
program p is
    var x, y : integer;
    var a : boolean;
begin
    read x; read y;
    a := x < y;
    if a then write x else write y end if
end
```

The program is syntactically correct if we can build a complete derivation tree for it. Recall that a tree is complete if every leaf is a terminal symbol or empty and that a preorder traversal of the leaf nodes matches the target program once the symbols have been replaced with the corresponding tokens from the representation table. We introduce metarules and hyper-rules on an “as needed” basis while discussing the sample program. The complete two-level grammar, with all rules identified by number, will appear later in this chapter in Figures 4.12 and 4.13. The representation table for Wren has already been presented in Figure 4.2.

We first introduce the metanotions that serve as some of the basic building blocks in Wren.

(m1) **ALPHA :: a; b; c; d; e; f; g; h; i; j; k; l; m;**
n; o; p; q; r; s; t; u; v; w; x; y; z.

(m2) **NUM :: zer o; one; two; thr ee; four; five; six; seven; eight; nine.**

(m3) **ALPHANUM :: ALPHA; NUM.**

(m4) **LETTER :: letter ALPHA.**

(m5) **DIGIT :: digit NUM.**

(m6) **LETTERDIGIT :: LETTER; DIGIT .**

A **NAME** starts with a letter followed by any sequence of letters or digits. One possible **NAME** is

letter r digit two letter d digit two

A **NUMERAL** is a sequence of digits.

(m7) **NAME :: LETTER; NAME LETTERDIGIT .**

(m8) **NUMERAL :: DIGIT ; NUMERAL DIGIT .**

Declarations

A declaration associates a name with a type. Suppose that a Wren program contains the following declarations:

var sum1 : **integer** ;
var done : **boolean** ;

These declarations will be represented in our two-level grammar derivation tree as

letter s letter u letter m digit 1 type integer
letter d letter o letter n letter e type boolean

The following metanotions define a declaration and a declaration sequence. Since a valid Wren program may have no declarations—for example, it may be a program that writes only constant values—we need to allow for an empty declaration sequence.

(m9) **DECL :: NAME type TYPE.**

(m10) **TYPE :: integer; boolean; pr ogram.**

(m11) **DECLSEQ :: DECL; DECLSEQ DECL.**

(m12) **DECLSEQETY :: DECLSEQ; EMPTY .**

(m13) **EMPTY :: .**

These metanotions are sufficient to begin construction of the declaration information for a Wren program. The most difficult aspect of gathering together the declaration information is the use of variable lists, such as

var w, x, y, z **: integer;**

which should produce the following **DECLSEQ** :

letter w type integer letter x type integer
letter y type integer letter z type integer

The difficulty is that **integer** appears only once as a terminal symbol and has to be "shared" with all the variables in the list. The following program fragments should produce this same **DECLSEQ**, despite the different syntactic form:

```
var w : integer;
var x : integer;
var y : integer;
var z : integer;
```

and

```
var w, x : integer;
var y, z : integer;
```

A DECLSEQ permits three alternatives: (1) a sequence followed by a single declaration, (2) a single declaration, or (3) an empty declaration.

(h3) **DECLSEQ DECL declaration seq :**
DECLSEQ declaration seq, DECL declaration.

(h4) **DECLSEQ declaration seq : DECLSEQ declaration.**

(h5) **EMPTY declaration seq : EMPTY .**

It should be noted that these three hyper-rules can be expressed as two rules (h4 is redundant), but we retain the three alternatives for the sake of clarity. If all variables are declared in a separate declaration, we will require a single hyper-rule for a declaration:

(h6) **NAME type TYPE declaration : var symbol, NAME symbol,**
colon symbol, TYPE symbol, semicolon symbol.

Figure 4.6 shows how h6, in combination with h3 and h4, can parse the definition of w, x, y, and z in separate declarations. For pedagogical emphasis, the corresponding metanotions are shown in italics to the left of the nodes in the tree, but these metanotions are not part of the tree itself. Observe that the specification of **NAME symbol** restricts identifier names to single characters. Since this makes the derivation tree more manageable with regard to depth, the example programs use only single letter identifiers. An exercise at the end of this section asks what modifications are needed in the grammar to allow for multiple character identifiers.

The same declaration sequence should be produced by the single declaration:

```
var w, x, y, z : integer;
```

This is accomplished by adding three hyper-rules:

1. The first variable in the list, which must be preceded by **var symbol** and followed by **comma symbol** .

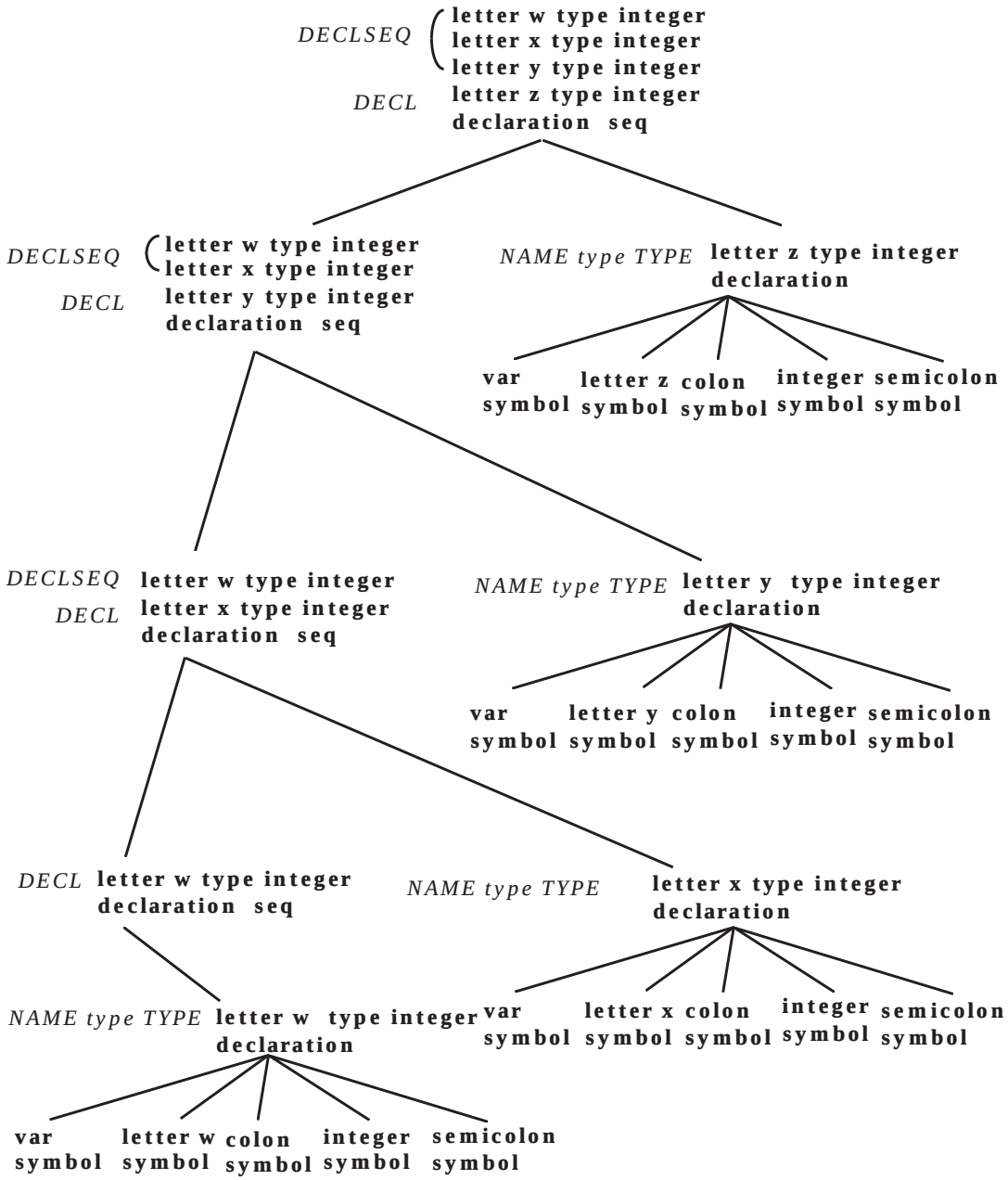

Figure 4.6: Parsing Individual Declarations

2. The last variable in the list, which must be followed by **colon symbol** , the type, and **semicolon symbol**
3. "In between" variables, each of which is followed by **comma symbol**

The general strategy is to have the type information passed from right to left. Here are the three hyper-rules that are used in conjunction with h4 to build the declaration information.

(h7) **DECLSEQ NAME type TYPE declaration :**
DECLSEQ NAME type TYPE var list,
NAME symbol, colon symbol, TYPE symbol, semicolon symbol.

(h8) **DECLSEQ NAME1 type TYPE NAME2 type TYPE var list :**
DECLSEQ NAME1 type TYPE var list,
NAME1 symbol, comma symbol.

(h9) **NAME1 type TYPE NAME2 type TYPE var list :**
var symbol, NAME1 symbol, comma symbol.

Figure 4.7 shows the derivation tree for the declaration of the variables w, x, y, and z in a single declaration statement.

We now return to our sample program. To develop the derivation tree from the program node, we need these hyper-rules for **program** and **block**.

(h1) **program : program symbol, NAME symbol, is symbol,**
block with NAME type pr ogram DECLSEQETY ,
where NAME type pr ogram DECLSEQETY unique.

(h2) **block with NAME type pr ogram DECLSEQETY :**
DECLSEQETY declaration seq, begin symbol,
NAME type pr ogram DECLSEQETY command seq, end symbol.

Notice that the program identifier name is added to the declaration sequence with type **program**. This information is passed to the command sequence and is also checked for multiple declarations of the same identifier by a **where** rule. A top-level derivation tree for the example program is shown in Figure 4.8.

The **where** rule checks for the uniqueness of declarations. All leaf nodes for the **where** clauses will be **EMPTY** if the identifiers are unique. Since our hyper-rules for variable lists have produced separate declaration information for each identifier, this checking is relatively straightforward, albeit lengthy. The easiest case is a single declaration, which is obviously unique.

(h22) **where DECL unique : EMPTY .**

In the case of multiple declarations, we separate the last declaration in the list, and we use the following rule to ensure that the name is not contained in any declarations to the left

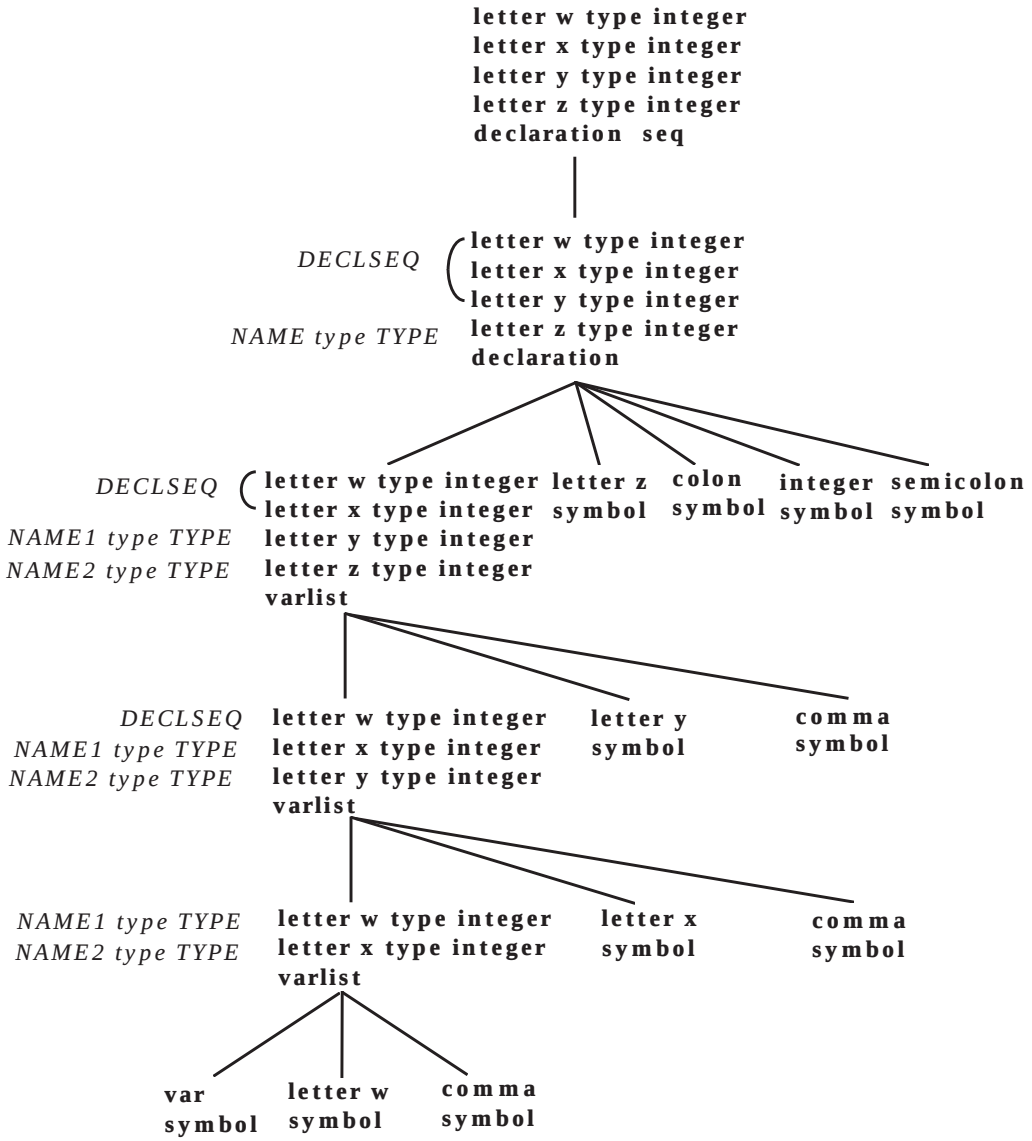

Figure 4.7: Parsing the Declaration: **var** w, x, y, z : **integer**;

(h23) **where DECLSEQ NAME type TYPE unique :**
where DECLSEQ unique,
where NAME not in DECLSEQ.

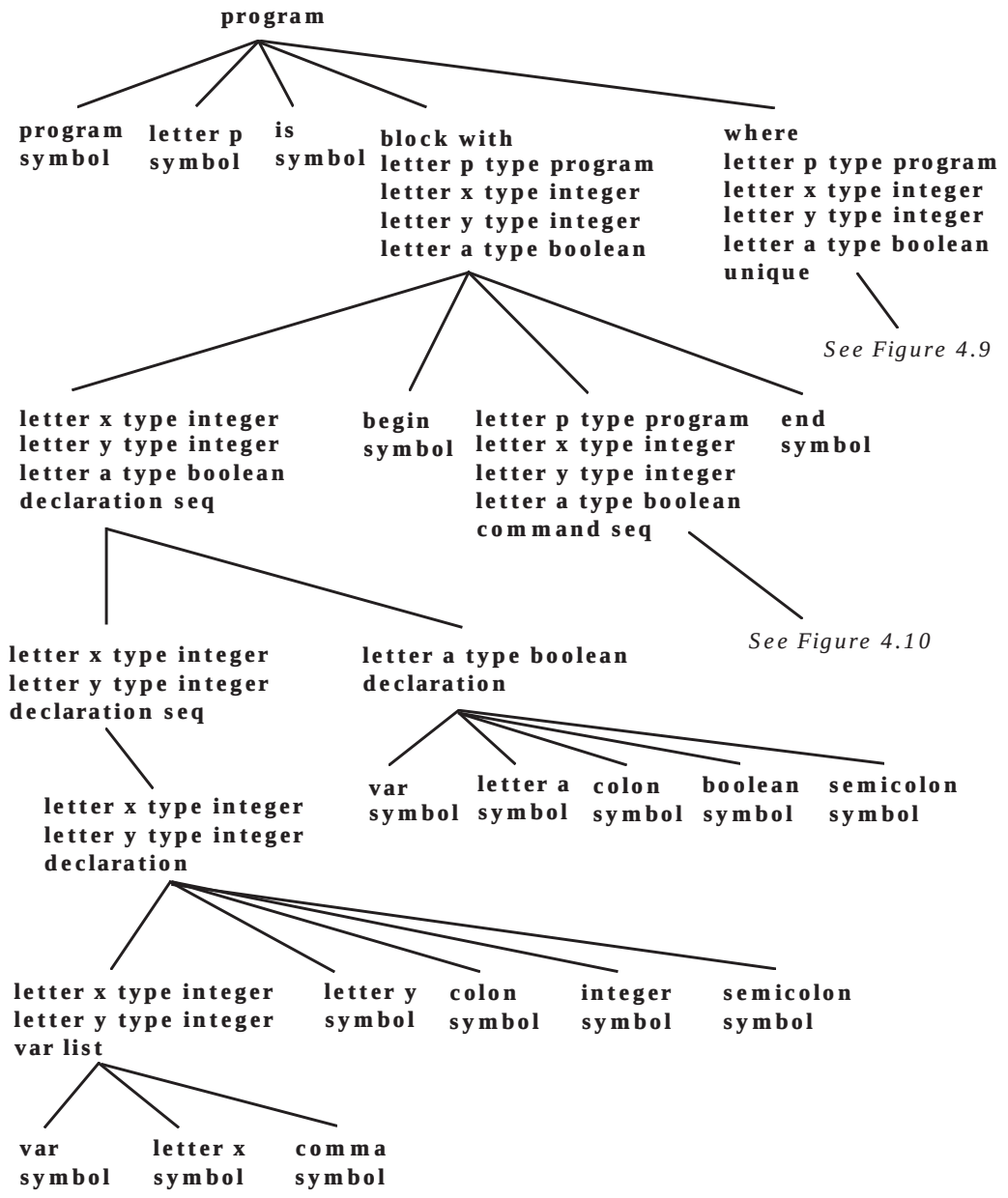

Figure 4.8: Top-Level Derivation Tree of Sample Program

To ensure that a name is not contained in a declaration sequence, we check one declaration at a time from right to left.

(h24) **where NAME not in DECLSEQ DECL :**
where NAME not in DECLSEQ,
where NAME not in DECL.

The type information in a declaration sequence is not needed to check for name uniqueness, so the hyper-rule simply checks that two names are not the same.

(h25) **where NAME1 not in NAME2 type TYPE :**
where NAME1 is not NAME2.

Names are separated into a sequence of characters, which is possibly empty, followed by a final character. We need to use the metanotions for **NOTION** and **NOTETY** introduced in section 4.1.

(m14) **NOTION :: ALPHA; NOTION ALPHA.**

(m15) **NOTETY :: NOTION; EMPTY .**

The identifiers contain either alphabetic characters or digit characters. If characters are of different kind, they are not equal. If characters are of the same kind but they are not the same, then one character appears before the other character in the appropriate character set. This test is applied to all characters in the sequence until a mismatch is found.

(h26) **where NOTETY1 NOTION1 ALPHANUM1 is not**
NOTETY2 NOTION2 ALPHANUM2 :
where NOTETY1 is not NOTETY2;
where NOTION1 dif ferent kind NOTION2;
where ALPHANUM1 pr ecedes ALPHANUM2
in abcdefghijklmnopqrstuvwxyz;
where ALPHANUM2 pr ecedes ALPHANUM1
in abcdefghijklmnopqrstuvwxyz;
where ALPHANUM1 pr ecedes ALPHANUM2
in zero one two thr ee four five six seven eight nine;
where ALPHANUM2 pr ecedes ALPHANUM1
in zero one two thr ee four five six seven eight nine.

A **letter** is always different than a **digit**.

(h27) **where letter dif ferent kind digit : EMPTY .**

(h28) **where digit dif ferent kind letter : EMPTY .**

Finally, two hyper-rules check whether a character or a digit precedes another.

(h29) **where ALPHA1 pr ecedes ALPHA2**
in NOTETY1 ALPHA1 NOTETY2 ALPHA2 NOTETY3 : EMPTY .

(h30) **where NUM1 pr ecedes NUM2**
in NOTETY1 NUM1 NOTETY2 NUM2 NOTETY3 : EMPTY .

Figure 4.9 shows the use of these **where** rules to check the uniqueness of the identifiers in our sample program.

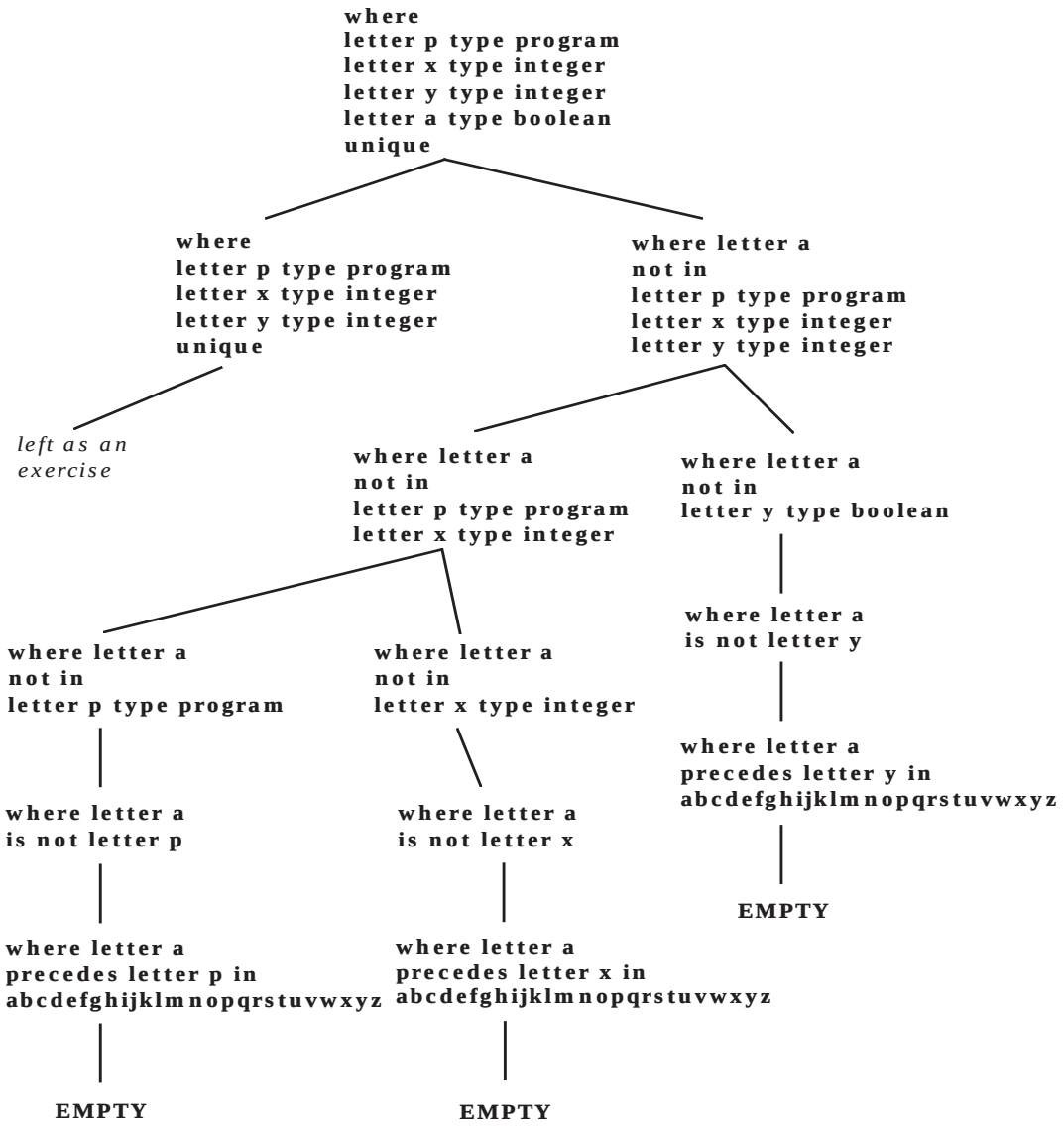

Figure 4.9: Checking Uniqueness of Declared Identifiers

Commands and Expressions

We complete the development of our two-level grammar for Wren by examining how declaration information is used to check for the proper use of variables that appear in commands. To reduce the number of hyper-rules, we introduce metanotions for the arithmetic operators and comparison operators.

(m16) **WEAKOP :: plus symbol; minus symbol.**

(m17) **STRONGOP :: multiply symbol; divide symbol.**

(m18) **RELATION ::**
less or equal symbol; less symbol; not equal symbol;
greater symbol; gr eater or equal symbol; equal symbol.

Declaration information is passed to each individual command. Note that an empty declaration sequence need not be allowed since every program must be named, even if no variables are declared.

(h10) **DECLSEQ command seq :**
DECLSEQ command;
DECLSEQ command, semicolon symbol,
DECLSEQ command seq.

Commands use the declaration information in different ways:

- The **skip** command uses no declaration information.
- The **write**, **while**, and **if** commands pass the declaration information to their constituent parts.
- The **read** command uses the declaration information to check that the associated variable is of type integer.

Here is the hyper-rule that separates these cases:

(h11) **DECLSEQ command :**
TYPE NAME in DECLSEQ, assign symbol,
TYPE expr ession in DECLSEQ;
skip symbol;
read symbol, integer NAME in DECLSEQ;
write symbol, integer expr ession in DECLSEQ;
while symbol, boolean expr ession in DECLSEQ, do symbol,
DECLSEQ command seq, end while symbol;
if symbol, boolean expr ession in DECLSEQ, then symbol,
DECLSEQ command seq, end if symbol;
if symbol, boolean expr ession in DECLSEQ, then symbol,
DECLSEQ command seq, else symbol,
DECLSEQ command seq, end if symbol.

The **read** command illustrates hyper-rules of the form **TYPE NAME in DECLSEQ** that perform two important functions: They produce the appropriate **NAME** symbol and they check that the **NAME** appears in the **DECLSEQ** with the appropriate **TYPE**.

(h19) **TYPE NAME in DECLSEQ :**
NAME symbol, wher e NAME type TYPE found in DECLSEQ.

The **DECLSEQ** is checked one declaration at a time from left to right. The **EMPTY** notion is produced if the appropriate declaration is found; otherwise, the parse fails.

(h20) **where NAME type TYPE found in**
NAME type TYPE DECLSEQETY : EMPTY .

(h21) **where NAME1 type TYPE1 found in NAME2 type TYPE2**
DECLSEQETY : wher e NAME1 type TYPE1
found in DECLSEQETY .

The remainder of the two-level grammar, dealing with expressions and comparisons, is straightforward. The portion of the grammar dealing with Boolean expressions has been left as an exercise.

(h12) **integer expr ession in DECLSEQ :**
ter m in DECLSEQ;
integer expr ession in DECLSEQ, WEAKOP , ter m in DECLSEQ.

(h13) **ter m in DECLSEQ :**
element in DECLSEQ;
ter m in DECLSEQ, STRONGOP , element in DECLSEQ.

(h14) **element in DECLSEQ :**
NUMERAL symbol;
integer NAME in DECLSEQ;
left par en symbol, integer expr ession in DECLSEQ,
right par en symbol;
negation symbol, element in DECLSEQ.

(h15) **boolean expr ession in DECLSEQ :** *left as an exercise.*

(h16) **boolean ter m in DECLSEQ :** *left as an exercise.*

(h17) **boolean element in DECLSEQ :** *left as an exercise.*

(h18) **comparison in DECLSEQ :**
integer expr ession in DECLSEQ, RELA TION,
integer expr ession in DECLSEQ.

Figures 4.10 and 4.11 illustrate a partial derivation tree for the command sequence in the sample program. The unfinished branches in the tree are left as exercises.

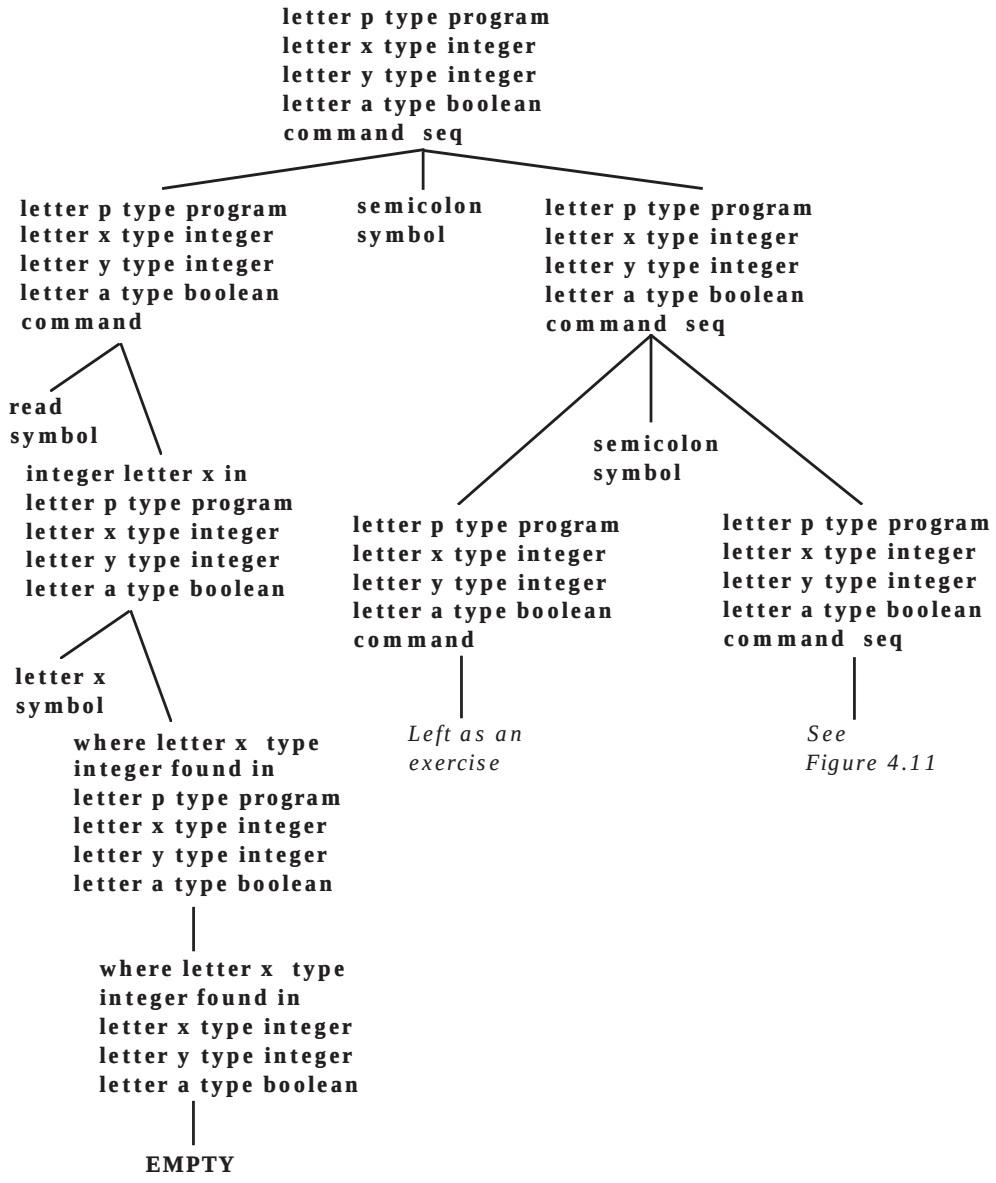

Figure 4.10: Partial Derivation Tree for a Command Sequence

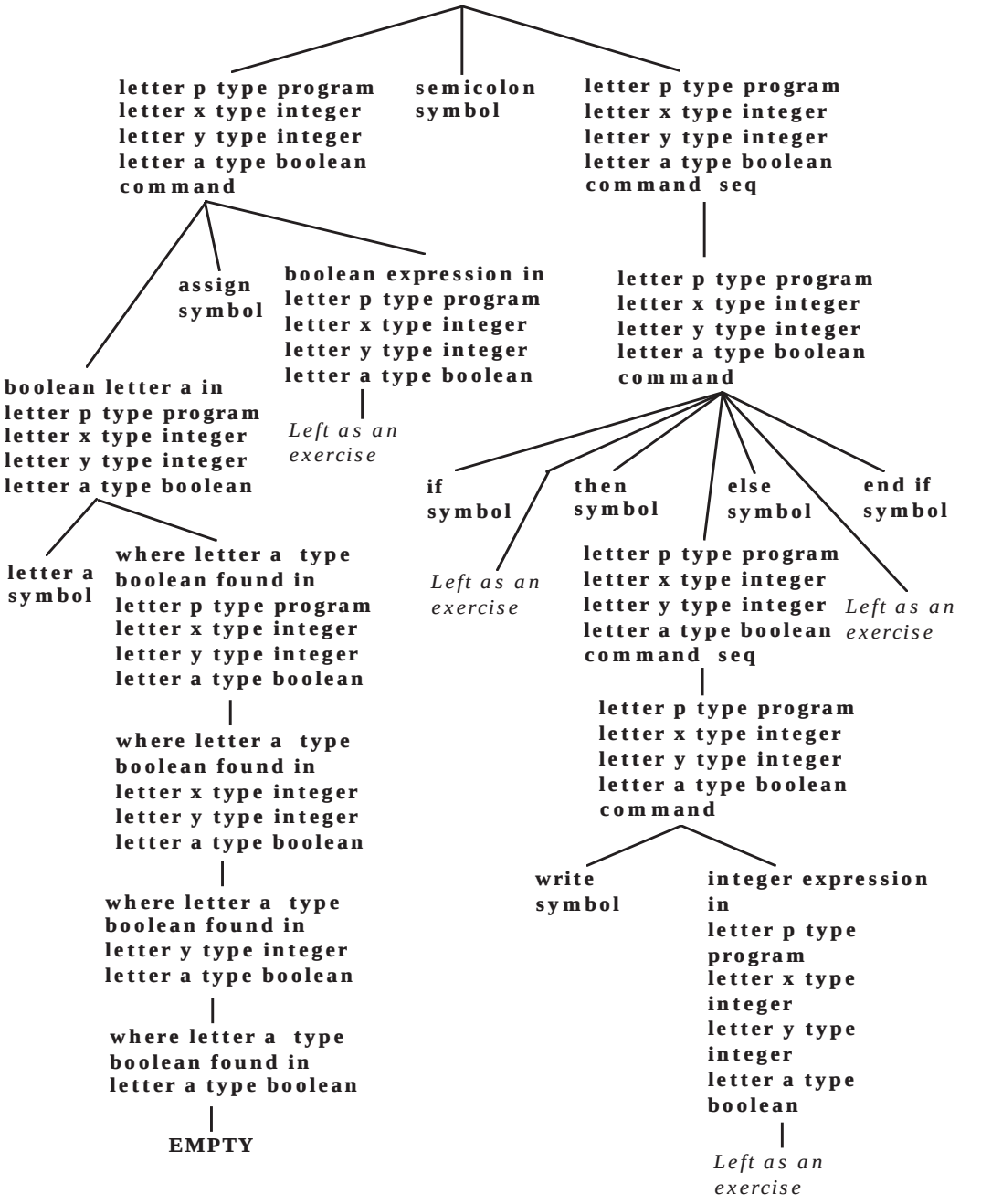

Figure 4.11: Partial Derivation Tree for a Command Sequence (continued)

(m1) **ALPHA :: a; b; c; d; e; f; g; h; i; j; k; l; m;**
n; o; p; q; r; s; t; u; v; w; x; y; z.

(m2) **NUM :: zer o; one; two; thr ee; four; five; six; seven; eight; nine.**

(m3) **ALPHANUM :: ALPHA; NUM.**

(m4) **LETTER :: letter ALPHA.**

(m5) **DIGIT :: digit NUM.**

(m6) **LETTERDIGIT :: LETTER; DIGIT .**

(m7) **NAME :: LETTER; NAME LETTERDIGIT .**

(m8) **NUMERAL :: DIGIT ; NUMERAL DIGIT .**

(m9) **DECL :: NAME type TYPE.**

(m10) **TYPE :: integer; boolean; pr ogram.**

(m11) **DECLSEQ :: DECL; DECLSEQ DECL.**

(m12) **DECLSEQETY :: DECLSEQ; EMPTY .**

(m13) **EMPTY :: .**

(m14) **NOTION :: ALPHA; NOTION ALPHA.**

(m15) **NOTETY :: NOTION; EMPTY .**

(m16) **WEAKOP :: plus symbol; minus symbol.**

(m17) **STRONGOP :: multiply symbol; divide symbol.**

(m18) **RELATION :: less or equal symbol; less symbol; not equal symbol;**
greater symbol; gr eater or equal symbol; equal symbol.

Figure 4.12: Metarules for Wren

(h1) **program : program symbol, NAME symbol, is symbol,**
block with NAME type pr ogram DECLSEQETY ,
where NAME type pr ogram DECLSEQETY unique.

(h2) **block with NAME type pr ogram DECLSEQETY :**
DECLSEQETY declaration seq, begin symbol,
NAME type pr ogram DECLSEQETY command seq, end symbol.

(h3) **DECLSEQ DECL declaration seq :**
DECLSEQ declaration seq, DECL declaration.

(h4) **DECLSEQ declaration seq :**
DECLSEQ declaration.

(h5) **EMPTY declaration seq : EMPTY .**

Figure 4.13: Hyper-rules for Wren (Part 1)

(h6) **NAME type TYPE declaration : var symbol, NAME symbol,**
colon symbol, TYPE symbol, semicolon symbol.

(h7) **DECLSEQ NAME type TYPE declaration :**
DECLSEQ NAME type TYPE var list,
NAME symbol, colon symbol, TYPE symbol, semicolon symbol.

(h8) **DECLSEQ NAME1 type TYPE NAME2 type TYPE var list :**
DECLSEQ NAME1 type TYPE var list,
NAME1 symbol, comma symbol.

(h9) **NAME1 type TYPE NAME2 type TYPE var list :**
var symbol, NAME1 symbol, comma symbol.

(h10) **DECLSEQ command seq :**
DECLSEQ command;
DECLSEQ command, semicolon symbol,
DECLSEQ command seq.

(h11) **DECLSEQ command :**
TYPE NAME in DECLSEQ, assign symbol,
TYPE expr ession in DECLSEQ;
skip symbol;
read symbol, integer NAME in DECLSEQ;
write symbol, integer expr ession in DECLSEQ;
while symbol, boolean expr ession in DECLSEQ, do symbol,
DECLSEQ command seq, end while symbol;
if symbol, boolean expr ession in DECLSEQ, then symbol,
DECLSEQ command seq, end if symbol;
if symbol, boolean expr ession in DECLSEQ, then symbol,
DECLSEQ command seq, else symbol,
DECLSEQ command seq, end if symbol.

(h12) **integer expr ession in DECLSEQ :**
ter m in DECLSEQ;
integer expr ession in DECLSEQ, WEAKOP , ter m in DECLSEQ.

(h13) **ter m in DECLSEQ :**
element in DECLSEQ;
ter m in DECLSEQ, STRONGOP , element in DECLSEQ.

(h14) **element in DECLSEQ :**
NUMERAL symbol;
integer NAME in DECLSEQ;
left par en symbol, integer expr ession in DECLSEQ,
right par en symbol;
negation symbol, element in DECLSEQ.

(h15) **boolean expr ession in DECLSEQ :** *left as exercise*

(h16) **boolean ter m in DECLSEQ :** *left as exercise*

Figure 4.13: Hyper-rules for Wren (Part 2)

(h17) **boolean element in DECLSEQ :** *left as exercise*

(h18) **comparison in DECLSEQ :**
integer expr ession in DECLSEQ, RELA TION,
integer expr ession in DECLSEQ.

(h19) **TYPE NAME in DECLSEQ :**
NAME symbol, wher e NAME type TYPE found in DECLSEQ

(h20) **where NAME type TYPE found in NAME type TYPE DECLSEQETY :**
EMPTY.

(h21) **where NAME1 type TYPE1 found in NAME2 type TYPE2**
DECLSEQETY : wher e NAME1 type TYPE1 found in
DECLSEQETY .

(h22) **where DECL unique : EMPTY .**

(h23) **where DECLSEQ NAME type TYPE unique :**
where DECLSEQ unique,
where NAME not in DECLSEQ.

(h24) **where NAME not in DECLSEQ DECL :**
where NAME not in DECLSEQ,
where NAME not in DECL.

(h25) **where NAME1 not in NAME2 type TYPE :**
where NAME1 is not NAME2.

(h26) **where NOTETY1 NOTION1 ALPHANUM1 is not**
NOTETY2 NOTION2 ALPHANUM2 :
where NOTETY1 is not NOTETY2;
where NOTION1 dif ferent kind NOTION2;
where ALPHANUM1 pr ecedes ALPHANUM2
in abcdefghijklmnopqrstuvwxyz;
where ALPHANUM2 pr ecedes ALPHANUM1
in abcdefghijklmnopqrstuvwxyz;
where ALPHANUM1 pr ecedes ALPHANUM2
in zer o one two thr ee four five six seven eight nine;
where ALPHANUM2 pr ecedes ALPHANUM1
in zer o one two thr ee four five six seven eight nine.

(h27) **where letter dif ferent kind digit : EMPTY .**

(h28) **where digit dif ferent kind letter : EMPTY .**

(h29) **where ALPHA1 pr ecedes ALPHA2**
in NOTETY1 ALPHA1 NOTETY2 ALPHA2 NOTETY3 : EMPTY .

(h30) **where NUM1 pr ecedes NUM2**
in NOTETY1 NUM1 NOTETY2 NUM2 NOTETY3 : EMPTY .

Figure 4.13: Hyper-rules for Wren (Part 3)

Exercises

1. Show the derivation tree for the following declaration sequence:
   var w, x **: integer;**
   var y, z **: integer;**

2. Complete the remaining branches in Figure 4.9.

3. Complete the following hyper-rules:
   (h15) **boolean expr ession in DECLSEQ :**
   (h16) **boolean ter m in DECLSEQ :**
   (h17) **boolean element in DECLSEQ :**

4. Complete the remaining branches in Figures 4.10 and 4.11.

5. Draw the complete derivation tree for the following program:

```
program p is
    var n, f : integer;
begin
  read n; f := 1;
  while n > 0 do f := f * n; n := n – 1 end while;
  write f
end
```

6. Suppose the declaration in exercise 5 is changed to
   var n **: integer;**
   var f **: boolean;**
   Show all locations in the command sequence where the parse fails, assuming the declaration sequence **letter n type integer letter f type boolean** .

7. Show all changes necessary to metarules and hyper-rules to allow for multiple character identifiers. Some rules already allow for multiple characters whereas others, such as those with **NAME symbol** , will have to be modified. Additional rules may be needed.

4.3 TWO-LEVEL GRAMMARS AND PROLOG

The consistent substitution of protonotions for a metanotion within a hyper-rule may seem similar to the binding of identifiers in a Prolog clause. In fact, a close relationship exists between two-level grammars and logic programming. Rather than present a complete implementation of the two-level gram-

mar for Wren in this section, we present a brief example of implementing a two-level grammar in Prolog and then discuss some of the relationships between two-level grammars and logic programming.

Implementing Two-Level Grammars in Prolog

We implement the Hollerith string literal grammar from Section 4.1 to give the flavor a two-level grammar in Prolog. The top-level predicate, named hollerith, is called with three arguments:

hollerith(<list of digit symbols>, hollerith, <list of lowercase letters>).

The program should print either "valid Hollerith string" or "invalid Hollerith string", as appropriate for the data. We assume the list of digits and list of letters are syntactically correct. A sample session appears below.

```
| ?- hollerith([digitSixSymbol], hollerith, [a,b,c,d,e,f]).
valid Hollerith string
yes

| ?- hollerith([digitSixSymbol], hollerith, [a,b,c,d,e]).
invalid Hollerith string
yes

| ?- hollerith([digitTwoSymbol,digitFiveSymbol], hollerith,
     [a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y]).
valid Hollerith string
yes

| ?- hollerith([digitTwoSymbol,digitFiveSymbol], hollerith,
     [a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z]).
invalid Hollerith string
yes
```

This interface is not very elegant, but it will serve adequately to illustrate the intended performance. An exercise suggests techniques for improving the interface. Observe that we have allowed for numerals of any size, so a **where** clause must be used in the grammar. The two-level grammar for Hollerith string literals is summarized in Figure 4.14.

```
hollerith : T ALLY digit, hollerith symbol, T   ALLY LETTERSEQ.

TALLY i LETTER LETTERSEQ : i LETTER, T   ALLY LETTERSEQ.
i LETTER: LETTER symbol.

i digit : digit one symbol.
ii digit : digit two symbol.
iii digit : digit thr  ee symbol.
iiii digit : digit four symbol.
iiiii digit : digit five symbol.
iiiiii digit : digit six symbol.
iiiiiii digit : digit seven symbol.
iiiiiiii digit : digit eight symbol.
iiiiiiiii digit : digit nine symbol.

TALLETY constant :
        TALLETY digit;
        TALLETY2 constant, T  ALLETY3 digit, wher  e TALLETY is
                TALLETY2 T ALLETY2 T ALLETY2 T ALLETY2 T ALLETY2
                TALLETY2 T ALLETY2 T ALLETY2 T ALLETY2 T ALLETY2
                TALLETY3.
where TALLETY is T ALLETY : EMPTY .

EMPTY digit : digit zer  o symbol.
TALLETY :: T ALLY; EMPTY.

APLHA :: a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p; q; r; s; t; u; v; w; x; y; z.
LETTER :: letter ALPHA.
LETTERSEQ :: LETTER; LETTERSEQ LETTER.
```

Figure 4.14: Two-level Grammar for Hollerith String Literals

The clauses for single digits are simple.

```
digit([digitZeroSymbol], [ ]).
digit([digitOneSymbol], [i]).
digit([digitTwoSymbol], [i,i]).
digit([digitThreeSymbol], [i,i,i]).
digit([digitFourSymbol], [i,i,i,i]).
digit([digitFiveSymbol], [i,i,i,i,i]).
digit([digitSixSymbol], [i,i,i,i,i,i]).
digit([digitSevenSymbol], [i,i,i,i,i,i,i]).
digit([digitEightSymbol], [i,i,i,i,i,i,i,i]).
digit([digitNineSymbol], [i,i,i,i,i,i,i,i,i]).
```

A constant is a single digit or a sequence of digits. We use the technique of concatenating ten copies of the tally for the leading digits to the tally for the units digit to produce a final tally. Supporting clauses are used to split the digits into the leading digits and units digit, to concatenate the ten copies of the leading digit's tally to the units digit's tally, and to perform the concatenation itself.

```
constant(DIGIT, TALLETY) :- digit(TALLETY, DIGIT).

constant(DIGITS, TALLETY) :-
                    splitDigits(DIGITS, LeadingDIGITS, UnitDIGIT),
                    constant(LeadingDIGITS, TALLETY2),
                    digit(UnitDIGIT, TALLETY3),
                    concatTenPlusDigit(TALLETY2, TALLETY3, TALLETY).

splitDigits([D], [ ], [D]).

splitDigits([Head|Tail],[Head|Result],Unit) :- splitDigits(Tail, Result, Unit).

concatTenPlusDigit(TALLETY2, TALLETY3, TALLETY) :-
                    concat(TALLETY2, TALLETY2, TwoTimes),
                    concat(TwoTimes, TwoTimes, FourTimes),
                    concat(FourTimes, FourTimes, EightTimes),
                    concat(EightTimes, TwoTimes, TenTimes),
                    concat(TenTimes, TALLETY3, TALLETY).

concat([ ],L,L).

concat([Head|Tail],L,[Head|Result]) :- concat(Tail,L,Result).
```

The tally is generated from the number part, and it is used to check the length of the letter sequence. Each time a tally symbol is removed, a letter is removed. One base case is a single tally and a single letter, resulting in a valid hollerith string. If either the tally or the letter sequence becomes empty, the other base cases, the hollerith string is invalid. Each letter is checked to ensure that it is a lowercase character.

```
hollerith(Number,hollerith,Letters) :-  constant(Number, TALLETY),
                                        letterSeq(TALLETY, Letters).

letterSeq([i],[Letter]) :-   alpha(Letter),
                             write('valid Hollerith string'), nl.

letterSeq([i|TALLETY],[Letter|Letters]) :-  alpha(Letter),
                                            letterSeq(TALLETY, Letters).

letterSeq([ ],Letters) :- write('invalid Hollerith string'), nl.

letterSeq(Number,[ ]) :- write('invalid Hollerith string'), nl.
```

```
alpha(a).  alpha(b).  alpha(c).  alpha(d).  alpha(e).  alpha(f).
alpha(g).  alpha(h).  alpha(i).  alpha(j).  alpha(k).  alpha(l).
alpha(m).  alpha(n).  alpha(o).  alpha(p).  alpha(q).  alpha(r).
alpha(s).  alpha(t).  alpha(u).  alpha(v).  alpha(w).  alpha(x).
alpha(y).  alpha(z).
```

Two-level Grammars and Logic Programming

Hyper-rules in two-level grammars are similar to clauses in Prolog. In two-level grammars we have consistent substitution of the same value for a particular metanotion in a rule. In Prolog we have the consistent binding of the same value to a particular variable in a clause. Some of the syntax and pattern matching in two-level grammars are also similar to Prolog. Several researchers have investigated the relationships between two-level grammars (also known as W-grammars) and logic programming. We briefly summarize one of those approaches, originally presented by S. J. Turner in a paper entitled "W-Grammars for Logic Programming" [Turner84].

A programming language seldom completely represents a programming paradigm. For example, Common Lisp is not a purely functional language, and Prolog is not a purely logical language. Turner believes that two-level grammars as an implementation mechanism for logic programming have many advantages over Prolog, the most popular logic programming language. He implements a logic programming system based on a two-level (or W-) grammar in a system called WLOG. He claims this system overcomes some of the disadvantages of Prolog (see [Turner84], page 352).

- Understanding Prolog requires a detailed understanding of the backtracking mechanism built into the implementation of Prolog.
- The meaning of a Prolog program is highly dependent on the order of the clauses in the database, making the formal analysis of the semantics of Prolog very difficult.
- Many built-in predicates in Prolog have side effects that make parallel implementations difficult.
- Minor programming errors, such as misspelling, are difficult to find since the entire program fails with no indication of where the error occurred.

Consider the following example taken from Turner's paper:

MAN :: geor ge; john; paul.

WOMAN :: jane; mary; sue.

PERSON :: MAN; WOMAN.
THING :: flowers; food; football; food; wine.
LEGAL :: PERSON likes THING; PERSON likes PERSON.

A fact is usually stated as a hyper-rule with an empty right side, such as:

mary likes football : .
john likes wine : .
paul likes mary : .
jane likes sue : .
paul likes food : .
george likes football : .

We can make a query, such as finding out what paul likes:

paul likes THING?

which succeeds with THING matching mary and food. We can have compound queries, such as:

PERSON1 likes PERSON, PERSON likes THING?

which succeeds with paul likes mary and mary likes football. Hyper-rules with right-hand sides are used to express rules of logic. Consider the logical rule: Two people who like the same object like each other, which is expressed as:

PERSON1 likes PERSON2 : PERSON1 likes OBJECT ,
PERSON2 likes OBJECT .

From the database given above, we can conclude that mary likes john.

Turner's paper gives a formal definition of WLOG and discusses an implementation based on non-deterministic finite automata. This implementation uses a breadth-first search, which means that the order of the database is not critical and that certain types of parallelism can be realized. The system also handles *not* in a more understandable manner than Prolog. With the database given above, if we pose the query

paul likes WOMAN?

then only mary is found in the database. In WLOG, if we make the query

not [paul likes WOMAN]?

then the values of sue and jane satisfy the query. This should be compared with Prolog where this query fails. In WLOG, not[not[X]]? is satisfied by the same values as X?, but this is not true in Prolog, which is based on negation as failure. This completes our brief look at the relationship between two-level grammars and pure logic programming.

Exercises

1. Build a "front end" for the Hollerith string checker to prompt the user for string input and then print the appropriate evaluation message. Use the built-in Prolog clause name for converting a string to a sequence of ascii codes.

2. Implement a two-level grammar that parses strings of the form $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$. The program should either print that the string obeys the grammar or that it does not. Assume that the strings are syntactically correct, in the sense that they are a sequence of **a**'s followed by a sequence of **b**'s, followed by a sequence of **c**'s.

3. Implement a two-level grammar to recognize valid Pascal identifiers assuming that an identifier starts with a letter followed by a sequence of alphanumeric characters and that the length of the identifier is eight or fewer characters (see exercise 4 in section 4.1).

4.4 FURTHER READING

Two-level grammars are also called W-grammars after their developer, Aad van Wijngaarden, who described them in an early paper [vanW ijngaarden66]. The formal definition of Algol68 using two-level grammars appears in [van Wijngaarden76]. [Kupka80] has applied two-level grammars to model information processing.

Several references include two-level grammars as part of an overview of formal techniques. Most notable are [Pagan81], [Marcotty76], and [Cleaveland77]. Pagan develops an interpreter for a language with parameterized procedures using a two-level grammar.

Some of the theoretical issues relating to two-level grammars are discussed in [Deussen75] and [Slintzoff67]. The relationship between two-level grammars and programming language design is explored in [van Wijngaarden82], [Maluszynski84], and [Turner84].

The most active group of researchers in the United States working on two-level grammars is at the University of Alabama, Birmingham. Barrett Bryant and Balanjaninath Edupuganty are coauthors of several papers dealing with applications of two-level grammars to a wide variety of problem domains [Bryant86a], [Bryant86b], [Bryant88], [Edupuganty85], [Edupuganty88], [Edupuganty89], and [Sundararaghavan87].