

Specifying Syntax

Once you've learned how to program in some language, learning a new programming language isn't all that hard. When learning a new language you need to know two things. First, you need to know what the keywords and constructs of the language look like. In other words, you need to know the mechanics of putting a program together in the programming language. Are the semicolons in the right places? Do you use begin...end or do you use curly braces (i.e. { and }). Learning how a program is put together is called learning the syntax of the language. Syntax refers to the words and symbols of a language and how to write the symbols down in the right order.

Semantics is the word that is used when deriving meaning from what is written. The semantics of a program refers to what the program will do when it is executed. Informally, it may be much easier to say what a program does than to describe the syntactic structure of the program. However, syntax is a lot easier to describe formally than semantics. In either case, if you are learning a new language, you need to learn something about the syntax of the language first.

2.1 Terminology

Once again, **syntax** of a programming language determines the well-formed or grammatically correct programs of the language. **Semantics** describes how or whether such programs will execute.

- **Syntax** is how things look
- **Semantics** is how things work (the meaning)

Many questions we might like to ask about a program either relate to the syntax of the language or to its semantics. It is not always clear which questions pertain to syntax and which pertain to semantics. Some questions may concern semantic issues that can be determined statically, meaning before the program is run. Other semantic issues may be dynamic issues, meaning they can only be determined at run-time. The difference between static semantic issues and syntactic issues is sometimes a difficult distinction to make.

Example 2.1

Apparently

```
a=b+c;
```

is correct C++ syntax. But is it really a correct statement?

1. Have `b` and `c` been declared as a type that allows the `+` operation?
2. Is `a` assignment compatible with the result of the expression `b+c`?
3. Do `b` and `c` have values?
4. Does the assignment statement have the proper form?

There are lots of questions that need to be answered about this assignment statement. Some questions could be answered sooner than others. When a C++ program is compiled it is translated from C++ to machine language as described in the previous chapter. Questions 1 and 2 are issues that can be answered when the C++ program is compiled. However, the answer to the third question above might not be known until the C++ program executes. The answers to questions 1 and 2 can be answered at *compile-time* and are called *static* semantic issues. The answer to question 3 is a *dynamic* issue and is probably not determinable until run-time. In some circumstances, the answer to question 3 might also be a static semantic issue. Question 4 is definitely a syntactic issue.

Unlike the dynamic semantic issues discussed above, the correct syntax of a program is definitely statically determinable. Said another way, determining a syntactically valid program can be accomplished without running the program. The syntax of a programming language is specified by something called a grammar. But before discussing grammars, the parts of a grammar must be defined. A **terminal** or **token** is a symbol in the language.

- C++ terminals: **while**, **const**, (`,` `;`, **5**, **b**
- Terminal types are keywords, operators, numbers, identifiers, etc.

A **syntactic category** or **nonterminal** is a set of objects (strings) that will be defined in terms of symbols in the language (terminal and nonterminal symbols).

- C++ nonterminals: `<statement>`, `<expression>`, `<if-statement>`, etc.
- Syntactic categories define parts of a program like statements, expressions, declarations, and so on.

A **metalanguage** is a higher-level language used to specify, discuss, describe, or analyze another language. English is used as a metalanguage for describing programming languages, but because of the ambiguities in English, more formal metalanguages have been proposed. The next section describes a formal metalanguage for describing programming language syntax.

2.2 Backus Naur Form (BNF)

Backus Naur Format (i.e. BNF) is a formal metalanguage for describing language syntax. The word *formal* is used to indicate that BNF is unambiguous. Unlike English, the BNF language is not open to our own interpretations. There is only one way to read a BNF description.

BNF was used by John Backus to describe the syntax of Algol in 1963. In 1960, John Backus and Peter Naur, a computer magazine writer, had just attended a conference on Algol. As they returned from the trip it became apparent that they had very different views of what Algol would look like. As a result of this discussion, John Backus worked on a method for describing the grammars of languages. Peter Naur slightly modified it. The notation is called BNF, or Backus Naur Form or sometimes Backus Normal Form. BNF consists of a set of rules that have this form:

<syntactic category> ::= a string of terminals and nonterminals
 ”::=” means ”is composed of ” (sometimes written as \rightarrow)

Often, multiple rules defining the same syntactic category are abbreviated using the ”|” character which can be read as ”or” and means set union. That is the entire language. It’s not a very big metalanguage, but it is powerful. Consider the following examples.

Example 2.2

BNF Examples from Java

<primitive type> ::= boolean
 <primitive type> ::= char

Abbreviated

<primitive type> ::= boolean | char | byte | short | int | long | float | ...
 <argument list> ::= <expression> | <argument list> , <expression>
 <selection statement> ::=
 if (<expression>) <statement>
 | if (<expression>) <statement> else <statement>
 | switch (<expression>) <block>
 <method declaration> ::=
 <modifiers> <type specifier> <method declarator>
 throws <method body>
 | <modifiers> <type specifier> <method declarator> <method body>
 | <type specifier> <method declarator> throws <method body>
 | <type specifier> <method declarator> <method body>

The above description can be described in English as *the set of method declarations is the union of the sets of method declarations that explicitly throw an exception with those that don’t explicitly throw an exception with or without modifiers attached to their definitions*. The BNF is much easier to understand and is not ambiguous like this English description.

2.3 The EWE Language

EWE is an extension of a primitive language called RAM designed by Sethi[29] as a teaching language. RAM stands for Random Access Machine. You might ask, “Did you intentionally name this language EWE?”. “Yes!”, I’d sheepishly respond. You can think of the EWE language as representing the language of a simple computer. EWE is an interpreter much the way the Java Virtual Machine is an interpreter of Java byte codes. EWE is much simpler than the language of the Java Virtual Machine.

Example 2.3

Consider the C++ program fragment.

```
1  int a=0;
2  int b=5;
3  int c=b+1;
4  a=b*c;
5  cout << a;
```

The EWE code below implements the C++ program fragment above.

```
1  a := 0
2  b := 5
3  one := 1
4  c := b + one
5  a := b * c
6  writeInt(a)
7  halt
8  equ a M[0]    b M[1]    c M[2]    one M[3]
```

As you can see, there is a very close correspondence between the C++ program and the EWE program. You can’t write **c=b+1** in EWE directly. That required a little extra work. Of course, that’s not the only program that might implement the C++ program fragment given above.

Example 2.4

Here’s another EWE program that computes the same thing as the C++ program fragment given above. This EWE program isn’t quite as straightforward as the last one, but they do the same thing.

```
1  # int a=0;
2  R0:=0    # load 0 into R0
3  M[SP+12]:=R0
4  # int b = 5;
5  R1:=5    # load 5 into R1
6  M[SP+13]:=R1
7  # int c = b+1;
8  R2:=SP   # b+1
9  R2:=M[R2+13] # load b into R2
```

```

10 R3:=1      # load 1 into R3
11 R2:=R2+R3
12 M[SP+14]:=R2
13 # a = b*c;
14 R4:=SP     # b*c
15 R4:=M[R4+13] # load b into R4
16 R5:=SP
17 R5:=M[R5+14] # load c into R5
18 R4:=R4*R5
19 R6:=SP
20 M[R6+12]:=R4
21 R7:=M[SP+12]
22 writeInt(R7)
23 halt
24 equ SP M[10]    equ R0 M[0]    equ R1 M[0]
25 equ R2 M[0]    equ R3 M[1]    equ R4 M[0]
26 equ R5 M[1]    equ R6 M[1]    equ R7 M[0]

```

The EWE language's interpreter recognizes one statement per line. Comments begin with a `#` and extend to the end of the line. The statements are followed by equates that equate identifiers to memory locations. The EWE computation model consists of:

- data memory locations specified by `M[...]`
- an instruction memory containing statements

Statements in a EWE program are executed in sequence unless a `goto` statement is executed. Statement execution terminates when an error occurs or the **halt** statement is executed.

EWE BNF

The syntax of the EWE language is completely specified by the BNF given on page 26. The semantics of the interpreter is not. The `null` symbol is there to draw attention to the fact that the equates part may be empty (there might not be any equates in a program). Keywords are not case sensitive. Strings are delimited by single or double quotes.

The `readStr` function reads a string and places the first character in the first memref location. It continues putting characters of the string in successive memory locations until either the string ends or the string surpasses the length stored in the second memref minus 1. Strings are terminated with a null (i.e. 0) character. Note that while a single memory location is big enough to hold four characters, only one character is placed in each memory location.

The `writeStr` function writes a string starting at the memref location and extending in successive memory locations until a null character is encountered. If a null character does not terminate the string, the interpreter will raise an illegal memory reference exception.

26 Specifying Syntax

```
1 <eweprog> ::= <executable> <equates> EOF
2
3 <executable> ::=
4     <labeled instruction>
5     | <labeled instruction> <executable>
6
7 <labeled instruction> ::=
8     Identifier ":" <labeled instruction>
9     | <instr>
10
11 <instr> ::=
12     <memref> ":" Integer
13     | <memref> ":" String
14     | <memref> ":" "PC" "+" Integer
15     | "PC" ":" <memref>
16     | <memref> ":" <memref>
17     | <memref> ":" <memref> "+" <memref>
18     | <memref> ":" <memref> "-" <memref>
19     | <memref> ":" <memref> "*" <memref>
20     | <memref> ":" <memref> "/" <memref>
21     | <memref> ":" <memref> "%" <memref>
22     | <memref> ":" "M" "[" <memref> "+" Integer "]"
23     | "M" "[" <memref> "+" Integer "]" ":" <memref>
24     | "readInt" "(" <memref> ")"
25     | "writeInt" "(" <memref> ")"
26     | "readStr" "(" <memref> "," <memref> ")"
27     | "writeStr" "(" <memref> ")"
28     | "goto" Integer
29     | "goto" Identifier
30     | "if" <memref> <condition> <memref> "then" "goto" Integer
31     | "if" <memref> <condition> <memref> "then" "goto" Identifier
32     | "halt"
33     | "break"
34
35 <equates> ::=
36     null
37     | "equ" Identifier "M" "[" Integer "]" <equates>
38
39 <memref> ::=
40     "M" "[" Integer "]"
41     | Identifier
42
43 <condition> ::= ">=" | ">" | "<=" | "<" | "=" | "<>"
```

Listing 2.1: The EWE BNF

Practice 2.1

The following program is not a valid EWE program. Using the BNF for EWE list the problems with this program.

```

1  readln(A);
2  readln(B);
3  if A-B < 0 then
4      writeln(A)
5  else
6      writeln(B);

```

How could you rewrite this program so that it does what this program intends to do?

Practice 2.2

Write a EWE program to read a number from the keyboard and print out the sum of all the numbers from 1 to that number.

Example 2.5

EWE is essentially an assembly language. It contains a few higher-level constructs, but very few. The EWE program given below upper cases all the characters in a string read from the keyboard. The simple way to write an assembly language program is to first write it in a high-level language. For instance, the program might look something like this in a C-like language.

```

1  s = input();
2  i = 0;
3  while s[i] != 0 {
4      if ('a' <= s[i] && s[i] <= 'z')
5          s[i] = s[i] - 'a' + 'A';
6      i++;
7  }
8
9  printf("%s", s)

```

When writing the program in EWE you will want to program the opposite of any if-then or while loop conditions you wrote in the high-level language. This is because you are going to use a **goto** statement to assist in completing the code. If the condition is false in an if-then statement you will jump around the **then** part of the statement by jumping to code that is after the **then** part. The code below shows you the EWE code with the appropriate C code intermingled as comments. Comments in EWE begin with a pound sign (i.e. #).

```

1  zero:=0
2  one:=1
3  littlea := 97
4  littlez := 122
5  diff:=32

```

```
6 # s = input();
7 len:=100
8 readStr(s,len)
9 # i=0;
10 i:=100
11 # while s[i]!=0 {
12 loop: tmp:=M[i+0]
13 if tmp = zero then goto end
14 #   if ('a' <= s[i] && s[i] <= 'z')
15 if littlea > tmp then goto skip
16 if tmp > littlez then goto skip
17 #   s[i] = s[i] - 32;
18 tmp:=tmp-diff
19 M[i+0]:=tmp
20 skip:
21 #   i++;
22 i:=i+one
23 goto loop
24 # printf("%s",s)
25 end: writeStr(s)
26 halt
27
28 equ zero M[0] equ one M[1] equ littlea M[2]
29 equ littlez M[3] equ diff M[4] equ len M[5]
30 equ s M[100] equ tmp M[6] equ i M[7]
```

Practice 2.3

Write a EWE program that reads a list of numbers from the screen and prints them out in reverse order. In order to do this exercise you need to know something about indexed addressing (see the example above).

HINT: What kind of data structure lets you reverse the elements of a list?

2.4 Context-Free Grammars

Another name for a BNF grammar is a context-free grammar. The only difference is in the metalanguage used to write the grammar. A context-free grammar is defined as a four tuple:

$$G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$$

where

- \mathcal{N} is a set of symbols called nonterminals or syntactic categories.
- \mathcal{T} is a set of symbols called terminals or tokens.
- \mathcal{P} is a set of productions of the form $n \rightarrow \alpha$ where n is a nonterminal and α is a string of terminals and nonterminals.
- \mathcal{S} is a special nonterminal called the start symbol of the grammar.

Example 2.6

A grammar for expressions in programs can be specified as $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$$\begin{aligned}\mathcal{N} &= \{E, T, F\} \\ \mathcal{T} &= \{\textit{identifier}, \textit{number}, +, -, *, /, (,)\} \\ \mathcal{P} &\text{ is defined by the set of productions} \\ E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T / F \mid F \\ F &\rightarrow (E) \mid \textit{identifier} \mid \textit{number}\end{aligned}$$

2.5 Derivations

A *sentence* of a grammar is a string of tokens from the grammar. A sentence belongs to the language of a grammar if it can be derived from the grammar. This process is called constructing a derivation. A *derivation* is a sequence of sentential forms that starts with the start symbol of the grammar and ends with the sentence you are trying to derive. A *sentential form* is a string of terminals and nonterminals from the grammar. In each step in the derivation, one nonterminal of a sentential form, call it A , is replaced by a string of terminals and nonterminals, β , where $A \rightarrow \beta$ is a production in the grammar.

While the previous paragraph is a bit dense to read the first time it really isn't that hard. An example should clear things up.

Example 2.7

Prove that the expression $(5*x)+y$ is a member of the language defined by the grammar given in example 2.6 by constructing a derivation for it.

The derivation begins with the start symbol of the grammar and ends with the sentence.

$$\begin{aligned}E &\Rightarrow \underline{E + T} \Rightarrow T + T \Rightarrow F + T \Rightarrow \underline{(E) + T} \Rightarrow (T) + T \Rightarrow \underline{(T * F) + T} \Rightarrow (F * F) + \\ T &\Rightarrow \underline{(5 * F) + T} \Rightarrow (5 * x) + T \Rightarrow \underline{(5 * x) + F} \Rightarrow \underline{(5 * x) + y}\end{aligned}$$

The underlined parts are all examples of sentential forms.

 **Practice 2.4**

Construct a derivation for the expression $4 + (a - b) * x$.

Types of Derivations

A sentence of a grammar is **valid** if there exists at least one derivation for it using the grammar. There are typically many different derivations for a particular sentence of a grammar. However, there are two derivations that are of some interest to us in understanding programming languages.

- Left-most derivation - Always replace the left-most nonterminal when going from one sentential form to the next in a derivation.
- Right-most derivation - Always replace the right-most nonterminal when going from one sentential form to the next in a derivation.

Example 2.8

The derivation of the sentence $(5 * x) + y$ in example 2.7 is a left-most derivation. A right-most derivation for the same sentence is:

$$E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + y \Rightarrow T + y \Rightarrow F + y \Rightarrow (E) + y \Rightarrow (T) + y \Rightarrow (T * F) + y \Rightarrow (T * x) + y \Rightarrow (F * x) + y \Rightarrow (5 * x) + y$$

🔗 Practice 2.5

Construct a right-most derivation for the expression $x * y + z$.

2.6 Parse Trees

A grammar for a language can be used to build a tree representing a sentence of the grammar. This kind of tree is called a *parse tree* for reasons that will become clear in the next section. A parse tree is another way of representing a sentence of a given language. A parse tree is constructed with the start symbol of the grammar at the root of the tree. The children of each node in the tree must appear on the right hand side of a production with the parent on the left hand side of the same production. A program is syntactically valid if there is a parse tree for it using the given grammar.

While there are typically many different derivations of a sentence in a language, there is only one parse tree. This is true as long as the grammar is not ambiguous. In fact that's the definition of ambiguity in a grammar. A grammar is *ambiguous* if and only if there is a sentence in the language of the grammar that has more than one parse tree. See section 2.11 for more information.

Example 2.9

The parse tree for the sentence derived in example 2.7 is depicted in figure 2.1. Notice the similarities between the derivation and the parse tree.

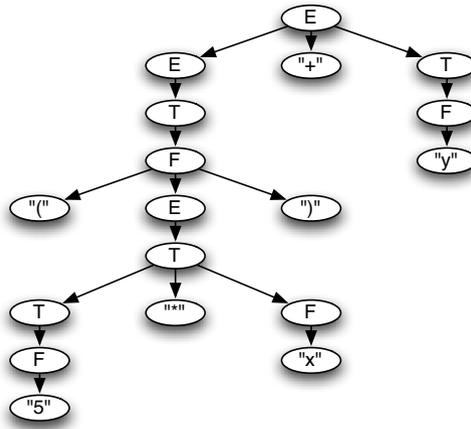


Fig. 2.1: A Parse Tree

Practice 2.6

What does the parse tree look like for the right-most derivation of $(5*x)+y$?

Practice 2.7

Construct a parse tree for the expression $4+(a-b)*x$.

HINT: What has higher precedence, “+” or “*”? The grammar given above automatically makes “*” have higher precedence. Try it the other way and see why!

2.7 Parsing

Parsing is the process of detecting whether a given string of tokens is a valid sentence of a grammar. Every time you compile a program or run a program in an interpreter the process described in this section is executed. Sometimes it completes successfully and sometimes it doesn't. When it doesn't you are told there is a syntax error in your program. A *parser* is a program that given a sentence, checks to see if the sentence is a member of the language of the given grammar. It may or may not construct a parse tree for the sentence at the same time.

- A top-down parser starts with the root of the tree
- A bottom-up parser starts with the leaves of the tree

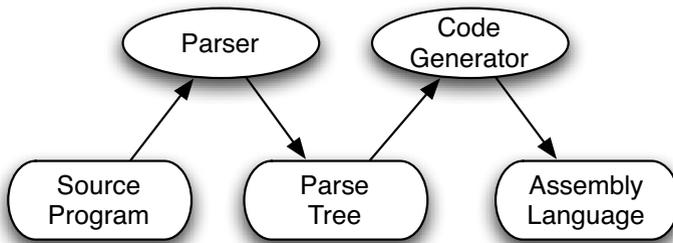


Fig. 2.2: Flow of Data surrounding a Parser

Top-down and bottom-up parsers check to see if a sentence belongs to a grammar by constructing a derivation for the sentence, using the grammar. A parser either reports success (and possibly returns the parse tree) or reports failure (hopefully with a nice error message). The flow of data is pictured in figure 2.2.

2.8 Parser Generators

A parser generator is a program that given a grammar, constructs a parser for the language specified by the grammar. This is a program that generates a program as pictured in figure 2.3. Examples of parser generators are yacc and ml-yacc. They both generate bottom-up parsers.

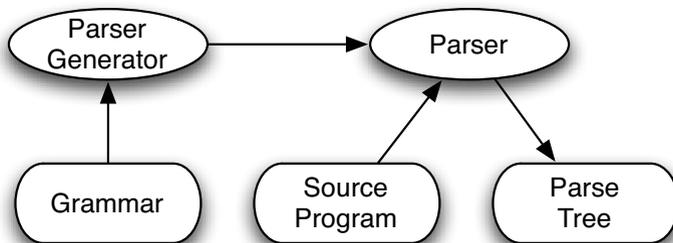


Fig. 2.3: Flow of Data surrounding a Parser Generator

2.9 Bottom-Up Parsers

As described above, bottom-up parsers are generally generated by a parser generator like ml-yacc (used by ML programs) or yacc (used by C and C++ programs). Parser generators construct a parse tree from the bottom up. We can be more specific. They actually construct a reverse right-most derivation of the sentence (i.e. Source program).

A parser generator works by (possibly) looking at the next token (i.e. terminal) in the input and then decides based on that and the partial derivation so far which production to apply to get the next step in the reverse right-most derivation. This algorithm uses a particular type of abstract machine called a push-down automaton. You need a particular kind of grammar to construct a push-down automaton called an LALR(1) grammar. Many grammars are LALR(1). You can learn more about push-down automata in a compiler construction text. It is beyond the scope of this book.

2.10 Top-Down Parsers

Top-down parsers are generally written by hand. They are sometimes called recursive descent parsers because they can be written as a set of mutually recursive functions. A top-down parser constructs a left-most derivation of the sentence (i.e. source program).

A top-down parser operates by (possibly) looking at the next token in the source file and deciding what to do based on the token and where it is in the derivation. To operate correctly, a top-down parser must be designed using a special kind of grammar called an LL(1) grammar.

2.11 Other Forms of Grammars

As a computer programmer you will likely learn at least one new language and probably a few during your career. New application areas frequently cause new languages to be developed to make programming applications in that area more convenient. Java, JavaScript, and ASP.NET are three new languages that were created because of the world wide web. A recent trend in programming languages is to develop domain specific languages. So if you are designing elevator controllers you may be programming in a language that was specially designed for that purpose.

Programming language references almost always contain some kind of reference that describes the constructs of the language. Many of these programming references give the grammar of the language using a variation of a context free grammar. A few examples of these grammar variations are given here to make you aware of notation that is often used in language references.

CBL (Cobol-like) Grammars

These were originally used in the description of Cobol. They are not as formal as BNF.

1. Optional elements are enclosed in brackets: [].
2. Alternate elements are vertically enclosed in braces: { }.
3. Optional alternates are vertically enclosed in brackets.
4. A repeated element is written once followed by an ellipsis: ...
5. Required key words are underlined; optional noise words are not.
6. Items supplied by the user are written as lower case or as syntactic categories from which an item may be taken.

Example 2.10

Here is the description of the COBOL ADD statement.

```
<Cobol Add statement> ::=
  ADD { identifier } [ , identifier ] ... TO
      identifier [ROUNDED][, identifier [ROUNDED] ] ...
      [ ON SIZE ERROR <statement> ]
```

One such add statement might be:

```
ADD A, 5 TO B ROUNDED, D
      ON SIZE ERROR PERFORM E-ROUTINE
```

Extended BNF (EBNF)

Since a BNF description of the syntax of a programming language relies heavily on recursion to provide lists of items, many definitions use these extensions:

1. **item?** or **[item]** means item is optional.
2. **item*** or **{item}** means to take zero or more occurrences of an item.
3. **item+** means to take one or more occurrences of an item
4. Parentheses are used for grouping

Example 2.11

Here is an example of method declarations in Java.

```
<method declaration> ::=
  <modifiers>? <type specifier>
  <method declarator> throws ? <method body>
```

Syntax Diagrams

A syntax diagram is a graph or graphs that have been used to describe Pascal and other programming languages.

1. A terminal is shown in a circle or oval.
2. A syntactic category is placed in a rectangle.
3. The concatenation of two objects is indicated by a flowline.
4. The alternation of two objects is shown by branching.
5. Repetition of objects is represented by a loop.

Example 2.12

Here are some descriptions of simple expressions in Pascal. Each of these different methods describe the same simple expressions in Pascal. Notice that some descriptions are more compact than the BNF. Each of them are unambiguous in their descriptions.

While BNF is less compact, it is the easiest to enter on a keyboard and for computer programs to read. There is a trade-off between computer readability and human readability that is at the center of many of our decisions about how to formally define programming languages.

BNF

```

<simple expr> ::=
    <term>
    | <sign> <term>
    | <simple expr> <adding operator> <term>
<sign> ::= "+" | "-"
<adding operator> ::= "+" | "-" | "or"

```

CBL

```

<simple expr> ::=
    [ + ] <term> [ { - } or ] <term> ...

```

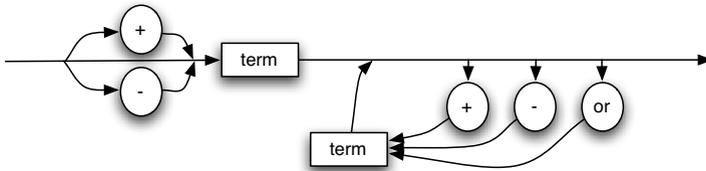
EBNF

```

<simple expr> ::= [ <sign> ] <term> { <adding operator> <term> }

```

Syntax Diagram



Practice 2.8

According to the syntactic specification in example 2.12, which of these terminal strings are simple expressions, assuming that a, b, and c are legal terms:

1. a+b-c
2. -a or b+c
3. b - - c

Ambiguous Grammars

As stated above, a grammar is ambiguous if there exists more than one parse tree for a given sentence of the language.

Example 2.13

The classic example is nested if-then-else statements. Consider the following Pascal statement:

```

1   if a<b then
2     if b<c then
3       writeln("a<c")
4   else
5     writeln("?")

```

Which *if* statement does the *else* go with? It's not entirely clear. According to the grammar, it could go with either. This means there is some ambiguity in the grammar for Pascal. This resolved by deciding the *else* should go with the nearest *if*. In a bottom-up parser this is called a shift/reduce conflict. In this case it is resolved by shifting instead of reducing.

🔗 Practice 2.9

Consider the expression grammar

$$\begin{aligned} \langle \text{expr} \rangle &::= \text{identifier} \mid \langle \text{expr} \rangle \langle \text{operator} \rangle \langle \text{expr} \rangle \\ \langle \text{operator} \rangle &::= "+" \mid "*" \end{aligned}$$

Consider the terminal string $a * b + c$.

Give two parse trees for this expression. This ambiguity could be resolved by specifying a precedence of operators in the grammar. However, there are better methods than specifying precedence. Precedence of operators can also be specified by introducing extra productions. See example 2.6 on page 29 for a better way of writing the grammar for this language.

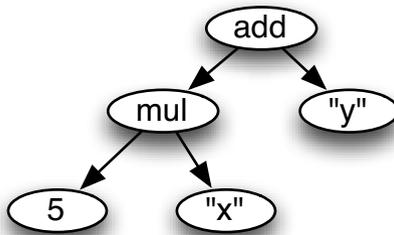
2.12 Abstract Syntax Trees

There is a lot of information in a parse tree that isn't really needed to encapsulate the program that it represents. An abstract syntax tree is like a parse tree except that non-essential information is removed. More specifically,

- Nonterminal nodes in the tree are replaced by nodes that reflect the part of the sentence they represent.
- Unit productions in the tree are collapsed.

Example 2.14

For example, the parse tree from figure 2.1 on page 31 can be represented by the following abstract syntax tree.



This tree eliminates all the unnecessary information and leaves just what is essential for evaluating the expression. Abstract syntax trees are used by compilers while generating code and by interpreters when running your program. Parse trees are usually not built by the parser, but the parser still constructs a derivation to check the syntax of a program. Usually, at the same time the abstract syntax tree is built.

Practice 2.10

What does the abstract syntax tree of $4+(a-b)*x$ look like?

2.13 Infix, Postfix, and Prefix Expressions

The abstract syntax tree in example 2.14 represents a computation. We can recover the infix expression it represents by doing an inorder traversal of the abstract syntax tree. To recall, an inorder traversal operates as follows:

```
1 Inorder_traverse(t a tree)
2   If t is an empty tree, do nothing
3   inorder_traverse(left subtree of t)
4   print the data of the root node in the tree t
5   inorder_traverse(right subtree of t)
```

Practice 2.11

Assume there is a `BTNode` class in your favorite object-oriented language with appropriate constructors, and `getData`, `getLeft`, and `getRight` member functions which return the data at a node, the left subtree, and the right subtree respectively. Write some code to implement this inorder traversal of a tree. Assume the AST in example 2.14 is given as input. What is the output? Is there anything wrong?

Practice 2.12

How does this code change to do a postorder traversal? What is the output given the tree in example 2.14.

2.14 Limitations of Syntactic Definitions

The concrete syntax for a language is almost always an incomplete description. Not all terminal strings generated are regarded as valid programs. For instance, consider the EWE BNF on page 26. A memory reference can be an identifier. The identifier must be defined in an equ statement. But, there is nothing in the grammar specifying this relationship.

In fact, there is no BNF (or EBNF or Syntax Diagram) grammar that generates only legal EWE programs. The same is true for C++, Java, ML, and all programming languages. A BNF grammar defines a context-free language: the left-hand side of each rule contains only one syntactic category. It is replaced by one of its alternative

definitions regardless of the context in which it occurs. The set of programs in any interesting language is not context-free.

Context-sensitive features may be formally described as a set of restrictions or context conditions. Context-sensitive issues deal mainly with declarations of identifiers and type compatibility.

Example 2.15

These are all context-sensitive issues.

- In an array declaration in C++, the array size must be a nonnegative value.
- Operands for the `&&` operation must be boolean in Java.
- In a method definition, the return value must be compatible with the return type in the method declaration.
- When a method is called, the actual parameters must match the formal parameter types.

2.15 Exercises

1. What does the word syntax refer to? How does it differ from semantics?
2. What is a token?
3. What is a nonterminal?
4. What does BNF stand for? What is its purpose?
5. Describe what the rules in lines 35-37 of the EWE BNF on page 26 mean. Answer this in some detail. Saying they define equates is not enough.
6. According to the EWE BNF, how many labels can an instruction have?
7. Given the grammar in example 2.6, derive the sentence $(4+5)*3$.
8. Draw a parse tree for the sentence $(4+5)*3$.
9. What kind of derivation does a top-down parser construct?
10. What would the abstract syntax tree for $(4+5)*3$ look like?
11. Describe how you might evaluate the abstract syntax tree of an expression to get a result? Write out your algorithm in English that describes how this might be done.
12. List four context-sensitive conditions in your favorite language.
13. Write a EWE program that prompts the user to enter three numbers and prints the max of the three numbers to the screen. Think about this before attempting to write it. It might be harder than you think at first.
14. Write a EWE program that prompts the user to enter a string and prints the reverse of that string to the screen.
15. Write a EWE program that prompts the user to enter a string and prints the string back to the screen with the first letter of each word upper cased.
16. Write a EWE program that asks the user to enter a number and prints either the square root of the number if it is an integer or the two integers the square root falls between if it is not an integer result. EWE does not operate on real numbers. It only works with integers and strings.
17. Using the EWE interpreter, write a program that prompts the user for a number and prints the factorial of that number.

2.16 Solutions to Practice Problems

These are solutions to the practice problems. You should only consult these answers after you have tried each of them for yourself first. Practice problems are meant to help reinforce the material you have just read so make use of them.

Solution to Practice Problem 2.1

Here is a correct version of the program. As you can see there are several things wrong with the original.

```

1   readInt(A)
2   readInt(B)
3   C := A - B
4   zero := 0
5   if C >= zero then goto pastwrtA
6       writeInt(A)
7       goto end
8   pastwrtA:
9       writeInt(B)
10  end:
11  halt
12  equ A M[0]    equ B M[1]    equ C M[2]    equ zero M[3]
```

Solution to Practice Problem 2.2

The easiest way to write EWE programs is to write in a language like Java or Python and then translate the code to EWE. Reverse any relational operators to make the translation (see the previous exercise). So for instance, a less than operator becomes greater or equal when translated into EWE. Here is a Python version of the program.

```

1 n = input("Enter a positive integer:")
2 sum = 0
3 for x in range(n+1):
4     sum = sum + x
5
6 print "The sum is", sum
```

And here is a EWE version.

```

1 readInt(n)
2 sum := 0
3 one := 1
4 x := 1
5 loop:
6     if x > n then goto end
7         sum := sum + x
8         x := x + one
```

42 Specifying Syntax

```
9   goto loop
10  end:
11  writeInt(sum)
12  halt
13  equ sum M[0]   equ one M[1]   equ x M[2]   equ n M[3]
```

If you think hard about this problem there is a simpler version that is about three lines long. You have to find the formula that computes the sum of the first n integers, though.

Solution to Practice Problem 2.3

You need to use indexed addressing to create a stack.

```
1  SP := 100
2  hundred := 100
3  zero := 0
4  one := 1
5  readloop:
6   readInt(x)
7   if x = zero then goto printloop
8   M[SP+0] := x
9   SP := SP + one
10  goto readloop
11  printloop:
12   SP := SP - one
13   if SP < hundred then goto end
14   x := M[SP+0]
15   writeInt(x)
16   goto printloop
17  end:
18  halt
19  equ SP M[0]   equ hundred M[1]   equ x M[3]
20  equ zero M[4]   equ one M[5]
```

Solution to Practice Problem 2.4

This is a left-most derivation of the expression.

$$\begin{aligned} E &\Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow 4 + T \Rightarrow 4 + T * F \Rightarrow 4 + F * F \Rightarrow 4 + (E) * F \Rightarrow \\ &4 + (E - T) * F \Rightarrow 4 + (T - T) * F \Rightarrow 4 + (F - T) * F \Rightarrow 4 + (a - T) * F \Rightarrow \\ &4 + (a - F) * F \Rightarrow 4 + (a - b) * F \Rightarrow 4 + (a - b) * x \end{aligned}$$

Solution to Practice Problem 2.5

This is a right-most derivation of the expression.

$$E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + z \Rightarrow T + z \Rightarrow T * F + z \Rightarrow T * y + z \Rightarrow F * y + z \Rightarrow x * y + z$$

Solution to Practice Problem 2.6

Exactly like the parse tree for any other derivation of $(5*x)+y$. There is only one parse tree for the expression given this grammar.

Solution to Practice Problem 2.7

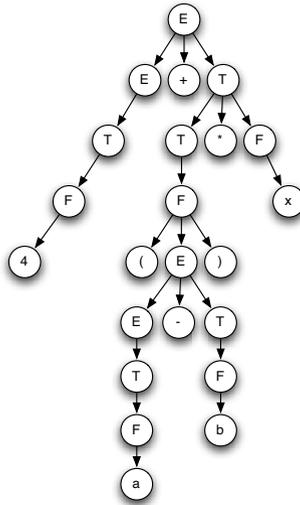


Fig. 2.4: The parse tree for practice problem 2.7

Solution to Practice Problem 2.8

1. $a+b-c$ is a valid simple expression.

44 Specifying Syntax

2. $-a$ or $b + c$ is a valid simple expression.
3. $b - - c$ is not a simple expression.

Solution to Practice Problem 2.9

In this problem we have a choice of putting the $*$ or the $+$ operator closer to the top of the tree. This will give us two different trees depending on which we choose.

Solution to Practice Problem 2.10

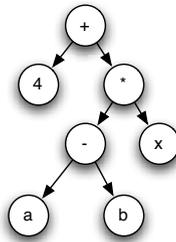


Fig. 2.5: The parse tree for practice problem 2.10

Solution to Practice Problem 2.11

```
1 void inordertraverse(BTNode root) {
2     if (root == nil) then return;
3
4     inordertraverse(root.getLeft());
5     System.out.println(root.getData() + " ");
6     inordertraverse(root.getRight());
7
8 }
```

The output would be $5 + x * y$. The traversal has thrown away the parentheses. If parens are needed the inorder traversal code could be modified to produce a fully parenthesized expression.

Solution to Practice Problem 2.12

The `println` statement would move to the last line of the function. The postorder output would be $5 x + y *$. No parens are needed in a postfix expression.

2.17 Additional Reading

This chapter introduces you to programming language syntax and reading syntactic descriptions. This is a worthwhile skill since you will undoubtedly come across new languages in your career as a computer scientist. There is certainly more that can be said about the topic of syntax of languages. Aho, Sethi, and Ullman [1] have written the widely recognized book on compiler implementation which includes material on syntax definition and parser implementation. There are many other good compiler references as well. The Chomsky hierarchy of languages is also closely tied to this topic. Many books on Discrete Structures in Computer Science introduce this topic and a few good books explore the Chomsky hierarchy more deeply including an excellent text by Peter Linz [21].



<http://www.springer.com/978-0-387-79421-1>

Programming Languages

An Active Learning Approach

Lee, K.D.

2008, XIV, 282 p., Hardcover

ISBN: 978-0-387-79421-1