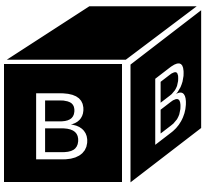


Chapter 2

Languages and Grammars

Two attributes of a programming language are its syntax and semantics. A computer language's syntax is the set of rules that a program listing must obey to be declared a valid program of the language. Its semantics is the meaning or logic behind the valid program. When you begin your study of the Component Pascal programming language in the next chapter you will need to know the language's syntax to be able to write programs that the computer will accept.

Syntax and semantics

Three common techniques to describe a language's syntax are:

Techniques for describing a language's syntax

- Grammars
- Regular expressions
- Finite state machines

This chapter introduces grammars. A variation of a grammar is used to describe the syntax of Component Pascal. Space limitations preclude a presentation of regular expressions and finite state machines. Later sections present finite state machines in a context other than describing a language's syntax.

Languages

Every language has an alphabet. Formally, an *alphabet* is a finite, nonempty set of characters.

An alphabet

Example 2.1 The alphabet for the language of real numbers that are not written in scientific notation is the set

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, .}

When you write a real number, such as -23.7, you use only the characters from the alphabet. If you attempt to write a real number using some other character not in the alphabet, such as -2y.7, then the sequence of characters that you write cannot be a valid real number. ∎

Example 2.2 Another example of a language is the language of expressions that you are familiar with from algebra. Examples of some valid algebraic expressions are:

a × b	c × (x + y)	c × y + x
(x – y) × (x + y)	(x – y) / (x + y)	(– (– (– ((b)))))

The expression a @ b is not valid because the character @ is not in the alphabet of the language of algebraic expressions. The alphabet for the language of algebraic expressions using only lowercase variable letters is:

{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, (,), +, –, ×, /}

The alphabet specifies which characters are legal to use in the language. To make a sentence in the language you join two or more characters together to form a string. The operation of joining them together is called *concatenation*. In Example 2.1 the string -23.7 is the concatenation of the individual characters -, 2, 3, ., and 7.

Concatenation

Concatenation applies not only to individual characters in an alphabet to construct a string, but also to strings to construct bigger strings. In Example 2.2 the string c × y is concatenated with the string + x to produce the string c × y + x.

The *length* of a string is the number of characters in the string. The string c × y + x has a length of five. The string of length zero, called the empty string, is denoted by the Greek letter ε to distinguish it from the English characters in an alphabet. Its concatenation properties are

The length of a string

$$\varepsilon x = x\varepsilon = x$$

The empty string

where x is a string. The empty string is useful for describing syntax rules.

In mathematics terminology, ε is the identity element for the concatenation operation. In general, an *identity element*, *i*, for an operation is one that does not change a value, *x*, when *x* is operated on by *i*.

Identity elements

Example 2.3 One is the identity element for multiplication because

$$1 \cdot x = x \cdot 1 = x$$

If *T* is an alphabet, the *closure* of *T*, denoted *T**, is the set of all possible strings formed by concatenating elements from *T*. *T** is extremely large. For example, if *T* is the set of characters and punctuation marks of the English alphabet, *T** includes all the sentences in the collected works of Shakespeare, in the English Bible, and in all the English encyclopedias ever published. It includes all strings of those characters ever printed in all the libraries in all the world throughout history, and then some. Not only does it include all those meaningful strings, it includes meaningless ones as well.

The closure of an alphabet

Example 2.4 Here are some elements of *T** for the English alphabet:

To be or not to be, that is the question.
Go fly a kite.
Here over highly toward?
alkeu jfoj ,9nm20mfq23jk l?x!jeo

Example 2.5 Some elements of *T** where *T* is the alphabet of the language for real numbers are

```
-2894.01
24
+78.3.80
--234--
6
```

You can easily construct many other elements of *T** with any of the alphabets in the previous examples. Because strings can be infinitely long, the closure of any alphabet has an infinite number of elements.

What is a language? In the examples of *T** that were just presented, some of the strings are in the language and some are not. In Example 2.4, the first two strings are valid English sentences; that is, they are in the language. The last two strings are not in the language. A *language* is a subset of the closure of its alphabet. Of the infinite number of strings you can construct from concatenating strings of characters from its alphabet, only some will be in the language.

A language

Example 2.6 Consider the following two elements of *T** where *T* is the alphabet for Example 2.2.

$a \times b$ $\quad\quad$ $c \times ((x + y)$

The first element of *T** is in the language of algebraic expressions, but the second is not, because it has a syntax error. It is illegal to have a left parenthesis without a matching right parenthesis.

Grammars

To define a language, you need a way to specify which of the many elements of *T** are in the language and which are not. A grammar is a system that specifies how you can concatenate the characters of alphabet *T* to form a legal string in a language. Formally, a grammar contains four parts:

The four parts of a grammar

- *N*, a nonterminal alphabet
- *T*, a terminal alphabet
- *P*, a set of rules of production
- *S*, the start symbol, an element of *N*

An element from the nonterminal alphabet, *N*, represents a group of characters from the terminal alphabet, *T*. A nonterminal symbol is sometimes a single descriptive word that begins with an uppercase letter to distinguish it from a terminal symbol. You see the terminals when you read the language. The rules of production use the nonterminals to describe the structure of the language, which may not be readily apparent when you read the language.

Example 2.7 In the English language, the nonterminals include Verb, Adverb,

Noun, Adjective, Preposition, and Subject among others. A valid English sentence is

Computer science is fun.

The word is is a Verb. Because Verb is a nonterminal you do not see it in the sentence. In other words, even though is is a Verb you would never see the sentence

Computer science Verb fun.

Every grammar has a special nonterminal called the start symbol, *S*. Notice that *N* is a set, but *S* is not. *S* is one of the elements of set *N*. The start symbol, along with the rules of production, *P*, enables you to decide whether a string of terminals is a valid sentence in the language. If, starting from *S*, you can generate the string of terminals using the rules of production, then the string is a valid sentence.

The start symbol

A grammar for identifiers

The Component Pascal programming language has a rule for naming things. The rule is that the first character of the name must be a letter or underscore character and the remaining characters, if any, can be letters, or digits, or underscores in any combination. The name is called a Component Pascal identifier. Grammar A in Figure 2.1 specifies these rules for a Component Pascal identifier. Even though an identifier can use any uppercase or lowercase letter, or digit, or the underscore character, to keep the example small this grammar permits only the letters a, b, and c and the digits 1, 2, and 3.

N = {Identifier, Letter, Digit}
T = {a, b, c, 1, 2, 3}
P = the productions
1. Identifier → Letter
2. Identifier → Identifier Letter
3. Identifier → Identifier Digit
4. Letter → a
5. Letter → b
6. Letter → c
7. Digit → 1
8. Digit → 2
9. Digit → 3

S = Identifier

Figure 2.1
Grammar A for Component Pascal identifiers.

This grammar has three nonterminals, namely, Identifier, Letter, and Digit. The start symbol is Identifier, one of the elements from the set of nonterminals. The rules of production are of the form

The rules of production

$A \rightarrow w$

where *A* is a nonterminal and *w* is a string of terminals and nonterminals. The symbol → means "produces." You should read production rule number 3 in this grammar as, "An identifier produces an identifier followed by a digit."

Derivations

The grammar specifies the language by a process called a *derivation*. To derive a valid sentence in the language, you begin with the start symbol and substitute for nonterminals from the rules of production until you get a string of terminals.

Example 2.8 Here is a derivation of the identifier cab3 from Grammar A:

Identifier	⇒ Identifier Digit	Rule 3
	⇒ Identifier 3	Rule 9
	⇒ Identifier Letter 3	Rule 2
	⇒ Identifier b 3	Rule 5
	⇒ Identifier Letter b 3	Rule 2
	⇒ Identifier a b 3	Rule 4
	⇒ Letter a b 3	Rule 1
	⇒ c a b 3	Rule 6

Next to each derivation step is the production rule on which the substitution is based. For example, Rule 2,

Identifier → Identifier Letter

was used to substitute for Identifier in the derivation step

Identifier 3 ⇒ Identifier Letter 3

The symbol ⇒ means "derives in one step." You should read this derivation step as "Identifier followed by 3 derives in one step Identifier followed by Letter followed by 3."

Closure of the derivation operation

Analogous to the closure operation on an alphabet is the closure of the derivation operation. The symbol ⇒* means "derives in zero or more steps." You can summarize the previous eight derivation steps as

Identifier ⇒* c a b 3

This derivation proves that cab3 is a valid identifier, because it can be derived from the start symbol, Identifier. A language specified by a grammar consists of all the strings derivable from the start symbol using the rules of production. The grammar provides an operational test for membership in the language. If it is impossible to derive a string, the string is not in the language.

A grammar for signed integers

Grammar B in Figure 2.2 defines the language of signed integers, where d represents a decimal digit. The start symbol is I, which stands for integer. F is the first character, which is an optional sign, and M is the magnitude.

N = {I, F, M}
T = {+, -, d}
P = the productions
1. I → F M
2. F → +
3. F → -
4. F → ε
5. M → d M
6. M → d

S = I

Figure 2.2
Grammar B for signed integers.

Sometimes the rules of production are not numbered and are combined on one line to conserve space on the printed page. You can write the rules of production for this grammar as

I → F M
F → + | - | ε
M → d | d M

where the vertical bar, |, is the alternation operator and is read as "or." Read the last line as "M produces d, or d followed by M."

Example 2.9 Here are some derivations of valid signed integers in this grammar:

I ⇒ F M
⇒ F d M
⇒ F d d M
⇒ F d d d
⇒ - d d d

I ⇒ F M
⇒ F d M
⇒ F d d
⇒ d d

I ⇒ F M
⇒ F d M
⇒ F d d M
⇒ F d d d M
⇒ F d d d d
⇒ + d d d d ∎

Note how the last step of the second derivation used the empty string to derive dd from Fdd. It used the production F → ε and the fact that εd = d. This production rule with the empty string is a convenient way to express the fact that a positive or negative sign in front of the magnitude is optional.

Some illegal strings from this grammar are ddd+, +-ddd, and ddd+dd. Try to derive these strings from the grammar to convince yourself that they are not in the language. Can you informally prove from the rules of production that each of these strings is not in the language?

The productions in both of the example grammars have recursive rules in which a nonterminal is defined in terms of itself. Rule 3 of Grammar A defines an Identifier in terms of an Identifier as

Identifier → Identifier Digit

and Rule 5 of Grammar B defines M in terms of M as

M → d M

Recursive rules produce languages with an infinite number of legal sentences. To derive an identifier, you can keep substituting Identifier Digit for Identifier as long as you like to produce an arbitrarily long identifier. Like all recursive definitions, there must be an additional nonrecursive rule to provide the basis for the definition. Otherwise the sequence of substitutions for the nonterminal could never stop. The nonrecursive rule M → d provides the basis for M in Grammar B.

A context sensitive grammar

The production rules for the previous grammars always contained a single nonterminal on the left side. Grammar C in Figure 2.3 has some production rules with both a terminal and nonterminal on the left side.

N = {A, B, C}
T = {a, b, c}
P = the productions
1. A → a A B C
2. A → a b C
3. C B → B C
4. b B → b b
5. b C → b c
6. c C → c c

S = A

Figure 2.3
Grammar C, a context sensitive grammar.

Example 2.10 Here is a derivation of a string of terminals with this grammar:

A ⇒ a A B C	Rule 1
⇒ a a A B C B C	Rule 1
⇒ a a a b C B C B C	Rule 2
⇒ a a a b B C C B C	Rule 3
⇒ a a a b B C B C C	Rule 3
⇒ a a a b B B C C C	Rule 3
⇒ a a a b b B C C C	Rule 4
⇒ a a a b b b C C C	Rule 4
⇒ a a a b b b c C C	Rule 5
⇒ a a a b b b c c C	Rule 6
⇒ a a a b b b c c c	Rule 6

An example of a substitution in this derivation is using Rule 5 in the step aaabbbCCC ⇒ aaabbbcCC. Rule 5 says that you can substitute c for C, but only if the C has a b to the left of it. ∎

In the English language, to quote a phrase out of context means to quote it without regard to the other phrases that surround it. Rule 5 is an example of a context-

sensitive rule. It does not permit the substitution of C by c unless C is in the proper context, namely, immediately to the right of a b.

Context sensitive grammars

Loosely speaking, a *context-sensitive grammar* is one in which the production rules may contain more than just a single nonterminal on the left side. In contrast, grammars that are restricted to a single nonterminal on the left side of every production rule are called context-free. (The precise theoretical definitions of context-sensitive and context-free grammars are more restrictive than these definitions. For the sake of simplicity, this chapter will use the previous definitions, although you should be aware that a more rigorous description of the theory would not define them as we have here.)

Some other examples of valid strings in the language specified by this grammar are abc, aabbcc, and aaaabbbbcccc. Two examples of invalid strings are aabc and cba. You should derive these valid strings and also try to derive the invalid strings to prove their invalidity to yourself. Some experimentation with the rules should convince you that the language is the set of strings that begins with one or more a's, followed by an equal number of b's, followed by the same number of c's. Mathematically, this language, *L*, can be written

$$L = \{a^n b^n c^n \mid n > 0\}$$

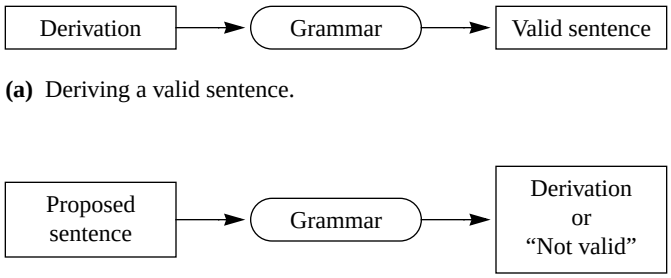
which you should read as "The language *L* is the set of strings $a^n b^n c^n$ such that *n* is greater than 0." The notation a^n means the concatenation of *n* a's.

The parsing problem

Deriving valid strings from a grammar is fairly straightforward. You can arbitrarily pick some nonterminal on the right side of the current intermediate string and select rules for the substitution repeatedly until you get a string of terminals. Such random derivations can give you many sample strings from the language.

Suppose we turn the problem around, however, and start with some given string of characters from the language's alphabet that is supposed to represent a valid sentence. You must determine if the string of terminals is indeed valid. But, the only way to determine if a string is valid is to derive it from the start symbol of the grammar. So, you must attempt such a derivation. If you succeed, you know the string is a valid sentence. The problem of determining whether or not a given string of terminal characters is valid for a specific grammar is called parsing, and is illustrated schematically in Figure 2.4.

Parsing a given string is more difficult than deriving an arbitrary valid string. The parsing problem is a form of searching. The parsing algorithm must search for just the right sequence of substitutions to derive the proposed string. Not only must it find the derivation if the proposed string is valid, but it must also admit the possibility that the proposed string may not be valid. If you look for a lost diamond ring in your room and do not find it, that does not mean the ring is not in your room. It may simply mean that you did not look in the right place. Similarly, if you try to find a derivation for a proposed string and do not find it, how do you know that such a derivation does not exist?



(a) Deriving a valid sentence.

(b) The parsing problem.

Figure 2.4
The difference between deriving an arbitrary sentence and parsing a proposed sentence.

A grammar for algebraic expressions

To see some of the difficulty a parser may encounter, consider Grammar D in Figure 2.5, which describes an algebraic expression. Nonterminal E represents the expression. T represents a term and F represents a factor.

N = {E, T, F}
T = {+, ×, (,), a}
P = the productions
1. E → E + T
2. E → T
3. T → T × F
4. T → F
5. F → (E)
6. F → a

S = E

Figure 2.5
Grammar D, a grammar for algebraic expressions.

Suppose you are given the string of terminals

(a × a) + a

and the production rules of this grammar and are asked to parse the proposed string. The correct parse is

E ⇒ E + T	Rule 1
⇒ T + T	Rule 2
⇒ F + T	Rule 4
⇒ (E) + T	Rule 5
⇒ (T) + T	Rule 2
⇒ (T × F) + T	Rule 3
⇒ (F × F) + T	Rule 4
⇒ (a × F) + T	Rule 6
⇒ (a × a) + T	Rule 6

$\Rightarrow (a \times a) + F$	Rule 4
$\Rightarrow (a \times a) + a$	Rule 6

The reason this could be difficult is that you might make a bad decision early in the parse that looks plausible at the time, but which leads to a dead end. For example, you might spot the "(" in the string that you were given and choose Rule 5 immediately. Your attempted parse might be

$E \Rightarrow T$	Rule 2
$\Rightarrow F$	Rule 4
$\Rightarrow (E)$	Rule 5
$\Rightarrow (T)$	Rule 2
$\Rightarrow (T \times F)$	Rule 3
$\Rightarrow (F \times F)$	Rule 4
$\Rightarrow (a \times F)$	Rule 6
$\Rightarrow (a \times a)$	Rule 6

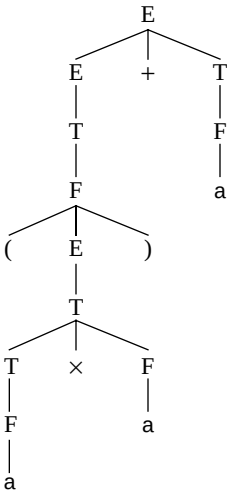


Figure 2.6
The syntax tree for the parse of $(a \times a) + a$ in Grammar D.

Until now, you have seemingly made progress toward your goal of parsing the original expression, because the intermediate string looks more like the original string at each successive step of the derivation. Unfortunately, now you are stuck, because there is no way to get the + a part of the original string. After reaching this dead end, you may be tempted to conclude that the proposed string is invalid, but that would be a mistake. Just because you cannot find a derivation, does not mean that such a derivation does not exist.

One interesting aspect of a parse is that it can be represented as a tree. The start symbol is the root of the tree. Each interior node of the tree is a nonterminal, and each leaf is a terminal. The children of an interior node are the symbols from the right side of the production rule substituted for the parent node in the derivation. The tree is called a syntax tree, for obvious reasons. Figure 2.6 shows the syntax tree for $(a \times a) + a$ with Grammar D, and Figure 2.7 shows it for dd with Grammar B.

Extended Backus-Naur form

The technique of using a grammar to specify the syntax rules of a programming language is sometimes called Backus-Naur Form (BNF) after John Backus and Peter Naur who developed it in the late 1950's. Backus was instrumental in the design of the Fortran language as was Naur for the Algol 60 language. An extended version of the system for specifying language syntax has come into use that is called Extended Backus-Naur Form (EBNF). The production rules are a bit simpler when written with EBNF because there is no need for the empty string ε, and there are usually not so many recursive production rules.

A minor difference in notation is that the equals sign = is sometimes used in place of the right arrow → when writing the production rules. An alternate notation is to use the two colons followed by the equals sign ::= to signify the same thing. More significantly, however, EBNF adds the following three operations:

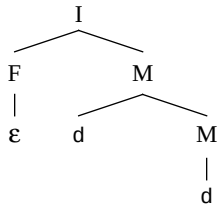


Figure 2.7
The syntax tree for the parse of dd in Grammar B.

The three operations of EBNF

- Alternation—select one of several alternatives
- Optional—include zero or one time
- Repetition—include zero or more times

The alternation symbol is the vertical bar |. The vertical bar was used previously to select between two complete productions of the grammar. However, in EBNF the bar is used within a single production rule. The notation for specifying that a symbol is optional is to enclose it in square brackets [ ]. To specify that it can be included even more than once enclose it in curly braces { }.

Example 2.11 The second and third production rules of Grammar A are recursive. That is, Identifier appears on both sides of the production arrow and is thus defined in terms of itself.

1. Identifier → Letter
2. Identifier → Identifier Letter
3. Identifier → Identifier Digit

As shown in Example 2.8 in the derivation of cab3, the effect of the recursive definition is to allow an unlimited number of letters or digits in the identifier. From the first production rule, Identifier must begin with Letter. These three production rules can be written in EBNF with one rule as

Identifier → Letter {Letter | Digit}

In English, you should read this as, "An identifier is a Letter followed by zero or more occurrences of a Letter or a Digit." ∎

Example 2.12 The production rules of Grammar B included one with the empty string to signify that the leading + or - sign is optional. The six production rules

1. I → F M
2. F → +
3. F → -
4. F → ε
5. M → d M
6. M → d

can be conveniently written in only one EBNF rule as

I → [+ | -] d { d }

which you should read in English as, "An I is an optional + or -, followed by one d, followed by zero or more occurrences of d." ∎

Component Pascal syntax

The Component Pascal syntax is specified by EBNF and is given in Appendix A. It is also available in the BlackBox on-line documentation under Component Pascal Language Report.

There are a few notational differences from the above examples. The production arrow → is written as an equals sign =. A more significant difference follows from the problem that the square brackets [ ], curly braces { } and vertical bar | used in the EBNF system are all valid characters in the language. So there must be some way to distinguish whether, for example, a square bracket [is an EBNF optional operator or a Component Pascal terminal symbol. The report makes the distinction by enclosing the symbol in double quotes " " if it is a Component Pascal terminal. Other terminals in the language are the words written in all uppercase letters.

Example 2.13 Referring to the production rules in Appendix A, here is a derivation that proves the string of terminals

```
IF alpha < 3 THEN DoBeta END
```

is a valid Statement. The derivation assumes that alpha and DoBeta have previously been shown to be valid Ident's, and that 3 has previously been shown to be a valid Integer.

Statement ⇒ IF Expr THEN StatementSeq END
⇒ IF SimpleExpr Relation SimpleExpr THEN StatementSeq END
⇒ IF Term Relation SimpleExpr THEN StatementSeq END
⇒ IF Factor Relation SimpleExpr THEN StatementSeq END
⇒ IF Designator Relation SimpleExpr THEN StatementSeq END
⇒ IF Qualident Relation SimpleExpr THEN StatementSeq END
⇒ IF Ident Relation SimpleExpr THEN StatementSeq END
⇒* IF alpha Relation SimpleExpr THEN StatementSeq END
⇒ IF alpha < SimpleExpr THEN StatementSeq END
⇒ IF alpha < Term THEN StatementSeq END
⇒ IF alpha < Factor THEN StatementSeq END
⇒ IF alpha < Number THEN StatementSeq END
⇒ IF alpha < Integer THEN StatementSeq END
⇒* IF alpha < 3 THEN StatementSeq END
⇒ IF alpha < 3 THEN Statement END
⇒ IF alpha < 3 THEN Designator END
⇒ IF alpha < 3 THEN Qualident END
⇒ IF alpha < 3 THEN Ident END
⇒* IF alpha < 3 THEN doBeta END

Figure 2.8 shows the corresponding syntax tree for this derivation. The dashed line from Ident to alpha indicates that more than one derivation step is hidden in the tree and corresponds to ⇒* in the above derivation. ∎

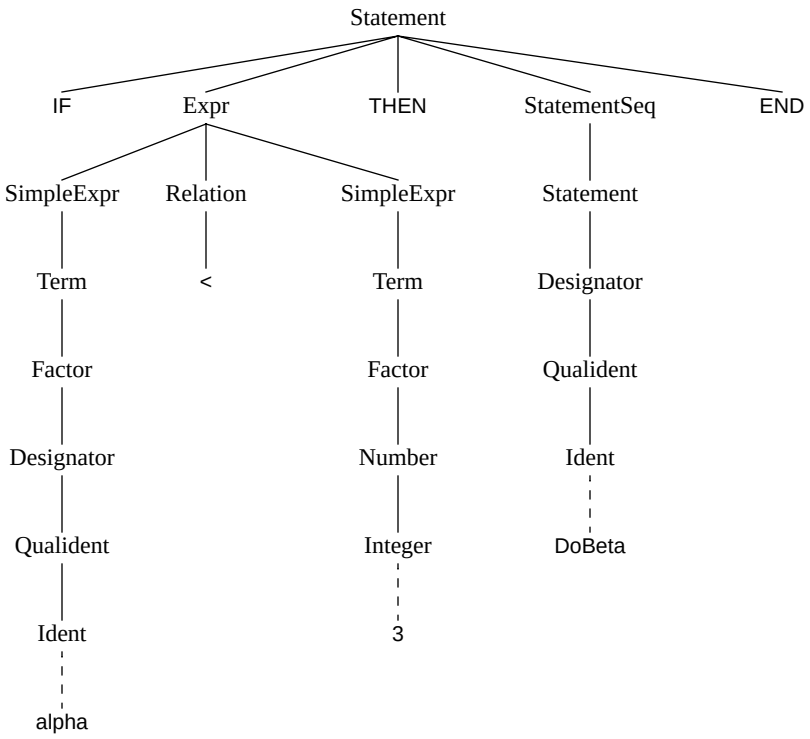


Figure 2.8
The syntax tree for the derivation in Example 2.13.

Unlike Grammar C, the production rules in Appendix A always have a single nonterminal on the left side. So it would appear from the production rules that the Component Pascal language is context free. However, it is not. The language report contains additional rules that must be followed to write a valid Component Pascal program. Following the grammar rules is a necessary but not sufficient condition for writing a valid program. That is, if you write a valid program you must conform to the grammar rules. But if you follow the grammar rules it does not automatically follow that you have written a valid program.

Exercises

1. What is the identity element for the addition operation on integers?

2. Derive the following strings with Grammar A in Figure 2.1 and draw the corresponding syntax tree.

(a) `abc123` **(b)** `a1b2c3` **(c)** `a321bc`

3. Derive the following strings with Grammar B in Figure 2.2 and draw the corresponding syntax tree.

(a) `-d` **(b)** `+ddd` **(c)** `d`

4. Derive the following strings with Grammar C in Figure 2.3.

(a) abc **(b)** aabbcc **(c)** aaaabbbbcccc

5. For each of the following strings, state whether it can be derived from the rules of Grammar D in Figure 2.5. If it can, draw the corresponding syntax tree.

(a) a + (a) **(b)** a × (+ a) **(c)** a × (a + a)
(d) a × (a + a) × a **(e)** a – a **(f)** (((a)))

6. For the grammar of Component Pascal in Appendix A, draw the syntax tree for StatementSeq from the following strings, assuming that *S1*, *S2*, *S3* and *S4* are each valid Statements and *C1* and *C2* are each valid Exprs.

(a)
```
IF C1 THEN
    S1
END ;
S2
```

(b)
```
IF C1 THEN
    S1 ;
    IF C2 THEN
        S2
    ELSE
        S3
    END
END ;
S4
```

(c)
```
IF C1 THEN
    IF C2 THEN
        S1
    ELSE
        S2
    END ;
    S3
ELSE
    S4
END
```

(d)
```
S1 ;
WHILE C1 DO
    IF C2 THEN
        S2
    END ;
    S3
END
```

7. For the Component Pascal grammar in Appendix A, draw the syntax tree for Statement.

(a) Alpha := 1
(b) Alpha := Alpha * 3
(c) Alpha := (Beta < 1)
(d) Alpha := ((Beta < 1) or (Gamma > 24))
(e) Alpha (Beta)
(f) Alpha (Beta, 24)