5 LANGUAGE SPECIFICATION

A study of the syntax and semantics of programming languages may be made at many levels, and is an important part of modern Computer Science. One can approach it from a very formal viewpoint, or from a very informal one. In this chapter we shall mainly be concerned with ways of specifying the concrete syntax of languages in general, and programming languages in particular. This forms a basis for the further development of the syntax- directed translation upon which much of the rest of this text depends.

5.1 Syntax, semantics, and pragmatics

People use languages in order to communicate. In ordinary speech they use natural languages like English or French; for more specialized applications they use technical languages like that of mathematics, for example

$$\forall x \, \exists \varepsilon :: | x - \xi | < \varepsilon$$

We are mainly concerned with programming languages, which are notations for describing computations. (As an aside, the word "language" is regarded by many to be unsuitable in this context. The word "notation" is preferable; we shall, however, continue to use the traditional terminology.) A useful programming language must be suited both to *describing* and to *implementing* the solution to a problem, and it is difficult to find languages which satisfy both requirements - efficient implementation seems to require the use of low-level languages, while easy description seems to require the use of high-level languages.

Most people are taught their first programming language by example. This is admirable in many respects, and probably unavoidable, since learning the language is often carried out in parallel with the more fundamental process of learning to develop algorithms. But the technique suffers from the drawback that the tuition is incomplete - after being shown only a limited number of examples, one is inevitably left with questions of the "can I do this?" or "how do I do this?" variety. In recent years a great deal of effort has been spent on formalizing programming (and other) languages, and in finding ways to describe them and to define them. Of course, a formal programming language has to be described by using another language. This language of description is called the **metalanguage**. Early programming languages were described using English as the metalanguage. A precise specification requires that the metalanguage be completely unambiguous, and this is not a strong feature of English (politicians and comedians rely heavily on ambiguity in spoken languages in pursuing their careers!). Some beginner programmers find that the best way to answer the questions which they have about a programming language is to ask them of the compilers which implement the language. This is highly unsatisfactory, as compilers are known to be error-prone, and to differ in the way they handle a particular language.

Natural languages, technical languages and programming languages are alike in several respects. In each case the **sentences** of a language are composed of sets of **strings** of **symbols** or **tokens** or **words**, and the construction of these sentences is governed by the application of two sets of rules.

- **Syntax Rules** describe the *form* of the sentences in the language. For example, in English, the sentence "They can fish" is syntactically correct, while the sentence "Can fish they" is incorrect. To take another example, the language of binary numerals uses only the symbols 0 and 1, arranged in strings formed by concatenation, so that the sentence 101 is syntactically correct for this language, while the sentence 1110211 is syntactically incorrect.

- **Semantic Rules**, on the other hand, define the *meaning* of syntactically correct sentences in a language. By itself the sentence 101 has no meaning without the addition of semantic rules to the effect that it is to be interpreted as the representation of some number using a positional convention. The sentence "They can fish" is more interesting, for it can have two possible meanings; a set of semantic rules would be even harder to formulate.

The formal study of syntax as applied to programming languages took a great step forward in about 1960, with the publication of the *Algol 60 report* by Naur (1960, 1963), which used an elegant, yet simple, notation known as **Backus-Naur-Form** (sometimes called **Backus-Normal-Form**) which we shall study shortly. Simply understood notations for describing semantics have not been so forthcoming, and many semantic features of languages are still described informally, or by example.

Besides being aware of syntax and semantics, the user of a programming language cannot avoid coming to terms with some of the pragmatic issues involved with implementation techniques, programming methodology, and so on. These factors govern subtle aspects of the design of almost every practical language, often in a most irritating way. For example, in Fortran 66 and Fortran 77 the length of an identifier was restricted to a maximum of six characters - a legacy of the word size on the IBM computer for which the first Fortran compiler was written.

5.2 Languages, symbols, alphabets and strings

In trying to specify programming languages rigorously one must be aware of some features of **formal language theory**. We start with a few abstract definitions:

- A **symbol** or **token** is an atomic entity, represented by a character, or sometimes by a reserved or key word, for example `+ , ; END`.

- An **alphabet** *A* is a non-empty, but finite, set of symbols. For example, the alphabet of Modula-2 includes the symbols

  ```
  - / * a b c A B C BEGIN CASE END
  ```

  while that for C++ would include a corresponding set

  ```
  - / * a b c A B C { switch }
  ```

- A **phrase**, **word** or **string** "over" an alphabet *A* is a sequence $\sigma = a_1 a_2 ... a_n$ of symbols from *A*.

- It is often useful to hypothesize the existence of a string of length zero, called the **null string** or **empty word**, usually denoted by ϵ (some authors use λ instead). This has the property that if it is concatenated to the left or right of any word, that word remains unaltered.

  $$a \epsilon = \epsilon a = a$$

- The set of all strings of length *n* over an alphabet *A* is denoted by A^n. The set of all strings (including the null string) over an alphabet *A* is called its **Kleene closure** or, simply, **closure**, and is denoted by A^*. The set of all strings of length at least one over an alphabet *A* is called its **positive closure**, and is denoted by A^+. Thus

  $$A^* = A^0 \cup A^1 \cup A^2 \cup A^3 \ldots$$

- A **language** *L* over an alphabet *A* is a subset of A^*. At the present level of discussion this involves no concept of meaning. A language is simply a set of strings. A language consisting of a finite number of strings can be defined simply by listing all those strings, or giving a rule for their derivation. This may even be possible for simple infinite languages. For example, we might have

  $$L = \{ \ (\ [a+ \ )^n \ (\ b] \ )^n \mid n > 0 \ \}$$

  (the vertical stroke can be read "such that"), which defines exciting expressions like

  [*a* + *b*]
  [*a* + [*a* + *b*] *b*]
  [*a* + [*a* + [*a* + *b*] *b*] *b*]

5.3 Regular expressions

Several simple languages - but by no means all - can be conveniently specified using the notation of **regular expressions**. A regular expression specifies the form that a string may take by using the symbols from the alphabet *A* in conjunction with a few other **metasymbols**, which represent operations that allow for

- *Concatenation* - symbols or strings may be concatenated by writing them next to one another, or by using the metasymbol · (dot) if further clarity is required.

- *Alternation* - a choice between two symbols *a* and *b* is indicated by separating them by the metasymbol **|** (bar).

- *Repetition* - a symbol *a* followed by the metasymbol ***** (star) indicates that a sequence of zero or more occurrences of *a* is allowable.

- *Grouping* - a group of symbols may be surrounded by the metasymbols **(** and **)** (parentheses).

As an example of a regular expression, consider

$$1 \ (\ 1 \mid 0 \ )^* \ 0$$

This generates the set of strings, each of which has a leading 1, is followed by any number of 0’s or 1’s, and is terminated with a 0 - that is, the set

{ 10, 100, 110, 1000 ... }

If a semantic interpretation is required, the reader will recognize this as the set of strings representing non-zero even numbers in a binary representation,

Formally, regular expressions may be defined inductively as follows:

- A regular expression denotes a regular set of strings.
- Ø is a regular expression denoting the empty set.
- ε is a regular expression denoting the set that contains only the empty string.
- σ is a regular expression denoting a set containing only the string σ.
- If A and B are regular expressions, then (A) and $A \mid B$ and $A \cdot B$ and A^* are also regular expressions.

Thus, for example, if σ and τ are strings generated by regular expressions, $\sigma\tau$ and $\sigma \cdot \tau$ are also generated by a regular expression.

The reader should take note of the following points:

- As in arithmetic, where multiplication and division take precedence over addition and subtraction, there is a precedence ordering between these operators. Parentheses take precedence over repetition, which takes precedence over concatenation, which in turn takes precedence over alternation. Thus, for example, the following two regular expressions are equivalent

  his | hers   and   h (i | er) s

  and both define the set of strings   { his , hers }.

- If the metasymbols are themselves allowed to be members of the alphabet, the convention is to enclose them in quotes when they appear as simple symbols within the regular expression. For example, comments in Pascal may be described by the regular expression

  "(" "*" c^* "*" ")"   where $c \in A$

- Some other shorthand is commonly found. For example, the positive closure symbol $^+$ is sometimes used, so that a^+ is an alternative representation for $a\ a^*$. A question mark is sometimes used to denote an optional instance of a, so that a? denotes $a \mid \varepsilon$. Finally, brackets and hyphens are often used in place of parentheses and bars, so that [*a-eBC*] denotes ($a \mid b \mid c \mid d \mid e \mid B \mid C$).

- Regular expressions have a variety of algebraic properties, among which we can draw attention to

$A \mid B = B \mid A$	(commutativity for alternation)
$A \mid (B \mid C) = (A \mid B) \mid C$	(associativity for alternation)
$A \mid A = A$	(absorption for alternation)
$A \cdot (B \cdot C) = (A \cdot B) \cdot C$	(associativity for concatenation)
$A (B \mid C) = A B \mid A C$	(left distributivity)
$(A \mid B) C = A C \mid B C$	(right distributivity)
$A \varepsilon = \varepsilon A = A$	(identity for concatenation)

$A^* A^* = A^*$ (absorption for closure)

Regular expressions are of practical interest in programming language translation because they can be used to specify the structure of the tokens (like identifiers, literal constants, and comments) whose recognition is the prerogative of the scanner (lexical analyser) phase of a compiler.

For example, the set of integer literals in many programming languages is described by the regular expression

$(0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)^+$

or, more verbosely, by

$(0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) \cdot (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)^*$

or, more concisely, by

$[0\text{-}9]^+$

and the set of identifiers by a similar regular expression

$(a | b | c | \ldots | Z) \cdot (0 | 1 | \ldots | 9 | a | \ldots | Z)^*$

or, more concisely, by

$[a\text{-}zA\text{-}Z][a\text{-}zA\text{-}Z0\text{-}9]^*$

Regular expressions are also powerful enough to describe complete simple assembler languages of the forms illustrated in the last chapter, although the complete expression is rather tedious to write down, and so is left as an exercise for the zealous reader.

Exercises

5.1 How would the regular expression for even binary numbers need modification if the string 0 (zero) was allowed to be part of the language?

5.2 In some programming languages, identifiers may have embedded underscore characters. However, the first character may not be an underscore, nor may two underscores appear in succession. Write a regular expression that generates such identifiers.

5.3 Can you find regular expressions that describe the form of REAL literal constants in Pascal? In C++? In Modula-2?

5.4 Find a regular expression that generates the Roman representation of numbers from 1 through 99.

5.5 Find a regular expression that generates strings like "facetious" and "abstemious" that contain all five vowels, in order, but appearing only once each.

5.6 Find a regular expression that generates all strings of 0's and 1's that have an odd number of 0's and an even number of 1's.

5.7 Describe the simple assembler languages of the last chapter by means of regular expressions.

5.4 Grammars and productions

Most practical languages are, of course, rather more complicated than can be defined by regular expressions. In particular, regular expressions are not powerful enough to describe languages that manifest *self-embedding* in their descriptions. Self-embedding comes about, for example, in describing structured statements which have components that can themselves be statements, or expressions comprised of factors that may contain further parenthesized expressions, or variables declared in terms of types that are structured from other types, and so on.

Thus we move on to consider the notion of a **grammar**. This is essentially a set of rules for describing **sentences** - that is, choosing the subsets of A^* in which one is interested. Formally, a grammar G is a quadruple $\{ N, T, S, P \}$ with the four components

(a) N - a finite set of **non-terminal** symbols,
(b) T - a finite set of **terminal** symbols,
(c) S - a special **goal** or **start** or **distinguished** symbol,
(d) P - a finite set of **production rules** or, simply, **productions**.

(The word "set" is used here in the mathematical sense.) A sentence is a string composed entirely of terminal symbols chosen from the set T. On the other hand, the set N denotes the **syntactic classes** of the grammar, that is, general components or concepts used in describing sentence construction.

The union of the sets N and T denotes the **vocabulary** V of the grammar.

$$V = N \cup T$$

and the sets N and T are required to be disjoint, so that

$$N \cap T = \emptyset$$

where Ø is the empty set.

A convention often used when describing grammars in the abstract is to use lower-case Greek letters (α, β, γ, ...) to represent strings of terminals and/or non-terminals, capital Roman letters (A, B, C ...) to represent single non- terminals and lower case Roman letters (a, b, c ...) to represent single terminals. Each author seems to have his or her own set of conventions, so the reader should be on guard when consulting the literature. Furthermore, when referring to the types of strings generated by productions, use is often made of the closure operators. Thus, if a string α consists of zero or more terminals (and no non-terminals) we should write

$$\alpha \in T^*$$

while if α consists of one or more non-terminals (but no terminals)

$$\alpha \in N^+$$

and if α consists of zero or more terminals and/or non-terminals

$$\alpha \in (N \cup T)^* \quad \text{that is, } \alpha \in V^*$$

English words used as the names of non-terminals, like *sentence* or *noun* are often non-terminals. When describing programming languages, reserved or key words (like `END, BEGIN` and `CASE`) are inevitably terminals. The distinction between these is sometimes made with the use of different type face - we shall use *italic font* for non-terminals and `monospaced font` for terminals where it is necessary to draw a firm distinction.

This probably all sounds very abstruse, so let us try to enlarge a little, by considering English as a written language. The set *T* here would be one containing the 26 letters of the common alphabet, and punctuation marks. The set *N* would be the set containing syntactic descriptors - simple ones like *noun*, *adjective*, *verb*, as well as more complex ones like *noun phrase*, *adverbial clause* and *complete sentence*. The set *P* would be one containing syntactic rules, such as a description of a *noun phrase* as a sequence of *adjective* followed by *noun*. Clearly this set can become very large indeed - much larger than *T* or even *N*. The productions, in effect, tell us how we can *derive* sentences in the language. We start from the distinguished symbol *S*, (which is always a non-terminal such as *complete sentence*) and, by making successive substitutions, work through a sequence of so- called **sentential forms** towards the final string, which contains terminals only.

There are various ways of specifying productions. Essentially a production is a rule relating to a pair of strings, say γ and δ, specifying how one may be transformed into the other. Sometimes they are called **rewrite rules** or **syntax equations** to emphasize this property. One way of denoting a general production is

$$\gamma \rightarrow \delta$$

To introduce our last abstract definitions, let us suppose that σ and τ are two strings each consisting of zero or more non-terminals and/or terminals (that is, $\sigma,\ \tau \in V = (N \cup T)^*$).

- If we can obtain the string τ from the string σ by employing *one* of the productions of the grammar *G*, then we say that σ *directly produces* τ (or that τ *is directly derived from* σ), and express this as $\sigma \Rightarrow \tau$.

  That is, if $\sigma = \alpha\delta\beta$ and $\tau = \alpha\gamma\beta$, and $\delta \rightarrow \gamma$ is a production in *G*, then $\sigma \Rightarrow \tau$.

- If we can obtain the string τ from the string σ by applying *n* productions of *G*, with $n \geq 1$, then we say that σ *produces* τ *in a non-trivial way* (or that τ *is derived from* σ *in a non-trivial way*), and express this as $\sigma \Rightarrow^+ \tau$.

  That is, if there exists a sequence $\alpha_o, \alpha_1, \alpha_2, \ldots \alpha_k$ (with $k \geq 1$), such that

$$\begin{array}{ll} \sigma = \alpha_o, & \\ \alpha_{j-1} \Rightarrow \alpha_j & \quad (\text{for } 1 \leq j \leq k) \\ \alpha_k = \tau, & \end{array}$$

then $\sigma \Rightarrow^{+} \tau$.

- If we can produce the string τ from the string σ by applying *n* productions of *G*, with $n \geq 0$ (this includes the above and, in addition, the trivial case where $\sigma = \tau$), then we say that σ *produces* τ (or that τ *is derived from* σ), and express this $\sigma \Rightarrow^{*} \tau$.
- In terms of this notation, a **sentential form** is the goal or start symbol, or *any* string that can be derived from it, that is, any string σ such that $S \Rightarrow^{*} \sigma$.
- A grammar is called *recursive* if it permits derivations of the form $A \Rightarrow^{+} \omega_1 A \omega_2$, (where $A \in N$, and $\omega_1, \omega_2 \in V^{*}$. More specifically, it is called *left recursive* if $A \Rightarrow^{+} A \omega$ and *right recursive* if $A \Rightarrow^{+} \omega A$.
- A grammar is *self-embedding* if it permits derivations of the form $A \Rightarrow^{+} \omega_1 A \omega_2$, (where $A \in N$, and where $\omega_1, \omega_2 \in V^{*}$, but where ω_1 or ω_2 contain at least one terminal (that is $(\omega_1 \cap T) \cup (\omega_2 \cap T) \neq \emptyset$).
- Formally we can now define a language *L*(*G*) produced by a grammar *G* by the relation

$$L(G) = \{ w \mid w \in T^{*} ; S \Rightarrow^{*} w \}$$

5.5 Classic BNF notation for productions

As we have remarked, a production is a rule relating to a pair of strings, say γ and δ, specifying how one may be transformed into the other. This may be denoted $\gamma \rightarrow \delta$, and for simple theoretical grammars use is often made of this notation, using the conventions about the use of upper case letters for non-terminals and lower case ones for terminals. For more realistic grammars, such as those used to specify programming languages, the most common way of specifying productions for many years was to use an alternative notation invented by Backus, and first called Backus-Normal-Form. Later it was realized that it was not, strictly speaking, a "normal form", and was renamed Backus-Naur-Form. Backus and Naur were largely responsible for the *Algol 60 report* (Naur, 1960 and 1963), which was the first major attempt to specify the syntax of a programming language using this notation. Regardless of what the acronym really stands for, the notation is now universally known as **BNF**.

In classic BNF, a non-terminal is usually given a descriptive name, and is written in angle brackets to distinguish it from a terminal symbol. (Remember that non-terminals are used in the construction of sentences, although they do not actually appear in the final sentence.) In BNF, productions have the form

$$\textit{leftside} \rightarrow \textit{definition}$$

Here "$\rightarrow$" can be interpreted as "is defined as" or "produces" (in some texts the symbol ::= is used in preference to $\rightarrow$). In such productions, both *leftside* and *definition* consist of a string concatenated from one or more terminals and non-terminals. In fact, in terms of our earlier notation

$$leftside \in (N \cup T)^+$$

and

$$definition \in (N \cup T)^*$$

although we must be more restrictive than that, for *leftside* must contain at least one non-terminal, so that we must also have

$$leftside \cap N \neq \emptyset$$

Frequently we find several productions with the same *leftside*, and these are often abbreviated by listing the *definitions* as a set of one or more alternatives, separated by a vertical bar symbol "|".

5.6 Simple examples

It will help to put the abstruse theory of the last two sections in better perspective if we consider two simple examples in some depth.

Our first example shows a grammar for a tiny subset of English itself. In full detail we have

```
G = {N , T , S , P}
N = {<sentence> , <qualified noun> , <noun> , <pronoun> , <verb> , <adjective> }
T = { the , man , girl , boy , lecturer , he , she , drinks , sleeps ,
      mystifies , tall , thin , thirsty }
S = <sentence>
P = {  <sentence>       →   the <qualified noun> <verb>       (1)
                          | <pronoun> <verb>                 (2)
       <qualified noun> →  <adjective> <noun>                (3)
       <noun>           →  man | girl | boy | lecturer       (4, 5, 6, 7)
       <pronoun>        →  he | she                          (8, 9)
       <verb>           →  talks | listens | mystifies       (10, 11, 12)
       <adjective>      →  tall | thin | sleepy              (13, 14, 15)
    }
```

The set of productions defines the non-terminal <sentence> as consisting of either the terminal "the" followed by a <qualified noun> followed by a <verb>, or as a <pronoun> followed by a <verb>. A <qualified noun> is an <adjective> followed by a <noun>, and a <noun> is one of the terminal symbols "man" or "girl" or "boy" or "lecturer". A <pronoun> is either of the terminals "he" or "she", while a <verb> is either "talks" or "listens" or "mystifies". Here <sentence>, <noun>, <qualified noun>, <pronoun>, <adjective> and <verb> are non-terminals. These do not appear in any sentence of the language, which includes such majestic prose as

the thin lecturer mystifies
he talks
the sleepy boy listens

From a grammar, one non-terminal is singled out as the so-called **goal** or **start symbol**. If we want to *generate* an arbitrary sentence we start with the goal symbol and successively replace each non-terminal on the right of the production defining that non-terminal, until all non-terminals have been removed. In the above example the symbol <sentence> is, as one would expect, the goal symbol.

Thus, for example, we could start with <sentence> and from this derive the sentential form the <qualified noun> <verb>

In terms of the definitions of the last section we say that <sentence> *directly produces* "the <qualified noun> <verb>". If we now apply production 3 (<qualified noun> $\rightarrow$ <adjective> <noun>) we get the sentential form

the <adjective> <noun> <verb>

In terms of the definitions of the last section, "the <qualified noun> <verb>" directly produces "the <adjective> <noun> <verb>", while <sentence> has produced this sentential form in a non-trivial way. If we now follow this by applying production 14 (<adjective> $\rightarrow$ thin) we get the form

the thin <noun> <verb>

Application of production 10 (<verb> $\rightarrow$ talks) gets to the form

the thin <noun> talks

Finally, after applying production 6 (<noun> $\rightarrow$ boy) we get the sentence

the thin boy talks

The end result of all this is often represented by a tree, as in Figure 5.1, which shows a **phrase structure tree** or **parse tree** for our sentence. In this representation, the order in which the productions were used is not readily apparent, but it should now be clear why we speak of "terminals" and "non-terminals" in formal language theory - the leaves of such a tree are all terminals of the grammar; the interior nodes are all labelled by non-terminals.

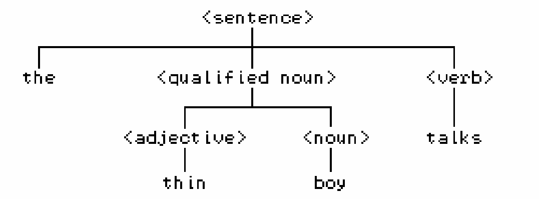


Figure 5.1 Parse tree for "the thin boy talks"

A moment's thought should reveal that there are many possible derivation paths from the goal or start symbol to the final sentence, depending on the order in which the productions are applied. It is convenient to be able to single out a particular derivation as being *the* derivation. This is generally called the **canonical derivation**, and although the choice is essentially arbitrary, the usual one is that where at each stage in the derivation the left-most non-terminal is the one that is replaced - this is called a **left canonical derivation**. (In a similar way we could define a **right canonical derivation**.)

Not only is it important to use grammars generatively in this way, it is also important - perhaps more so - to be able to take a given sentence and determine whether it is a valid member of the language - that is, to see whether it could have been obtained from the goal symbol by a suitable choice of derivations. When mere recognition is accompanied by the determination of the underlying tree structure, we speak of **parsing**. We shall have a lot more to say about this in later chapters; for the moment note that there are several ways in which we can attempt to solve the problem. A fairly natural way is to start with the goal symbol and the sentence, and, by reading the sentence from left to right, to try to deduce which series of productions must have been applied.

Let us try this on the sentence

the thin boy talks

If we start with the goal <sentence> we can derive a wide variety of sentences. Some of these will arise if we choose to continue by using production 1, some if we choose production 2. By reading no further than "the" in the given sentence we can be fairly confident that we should try production 1.

<sentence> → the <qualified noun> <verb>.

In a sense we now have a residual input string "thin boy talks" which somehow must match <qualified noun> <verb>. We could now choose to substitute for <verb> or for <qualified noun>. Again limiting ourselves to working from left to right, our residual sentential form <qualified noun> <verb> must next be transformed into <adjective> <noun> <verb> by applying production 3.

In a sense we now have to match "thin boy talks" with a residual sentential form <adjective> <noun> <verb>. We could choose to substitute for any of <adjective>, <noun> or <verb>; if we read the input string from the left we see that by using production 14 we can reduce the problem of matching a residual input string "boy talks" to the residual sentential form <noun> <verb>. And so it goes; we need not labour a very simple point here.

The parsing problem is not always as easily solved as we have done. It is easy to see that the algorithms used to parse a sentence to see whether it can be derived from the goal symbol will be very different from algorithms that might be used to generate sentences (almost at random) starting from the start symbol. The methods used for successful parsing depend rather critically on the way in which the productions have been specified; for the moment we shall be content to examine a few sets of productions without worrying too much about how they were developed.

In BNF, a production may define a non-terminal recursively, so that the same non-terminal may occur on both the left and right sides of the → sign. For example, if the production for <qualified noun> were changed to

<qualified noun> → <noun> | <adjective> <qualified noun> (3a, 3b)

this would define a <qualified noun> as either a <noun>, or an <adjective> followed by a <qualified noun> (which in turn may be a <noun>, or an <adjective> followed by a <qualified noun> and so on). In the final analysis a <qualified noun> would give rise to zero or more <adjective>s followed by a <noun>. Of course, a recursive definition can only be useful provided that there is some way of terminating it. The single production

<qualified noun> → <adjective> <qualified noun> (3b)

is effectively quite useless on its own, and it is the alternative production

<qualified noun> → <noun> (3a)

which provides the means for terminating the recursion.

As a second example, consider a simple grammar for describing a somewhat restricted set of algebraic expressions:

```
    G = {N , T , S , P}
    N = {<goal> , <expression> , <term> , <factor> }
    T = { a , b , c , - , * }
    S = <goal>
    P =
        <goal>         →  <expression>                           (1)
        <expression>   →  <term> | <expression> - <term>         (2, 3)
        <term>         →  <factor> | <term> * <factor>           (4, 5)
        <factor>       →  a | b | c                              (6, 7, 8)
```

It is left as an easy exercise to show that it is possible to derive the string *a* - *b* * *c* using these productions, and that the corresponding phrase structure tree takes the form shown in Figure 5.2.

A point that we wish to stress here is that the construction of this tree has, happily, reflected the relative precedence of the multiplication and subtraction operations - assuming, of course, that the symbols * and - are to have implied meanings of "multiply" and "subtract" respectively. We should also point out that it is by no means obvious at this stage how one goes about designing a set of productions that not only describe the syntax of a programming language but also reflect some semantic meaning for the programs written in that language. Hopefully the reader can foresee that there will be a very decided advantage if such a choice *can* be made, and we shall have more to say about this in later sections.

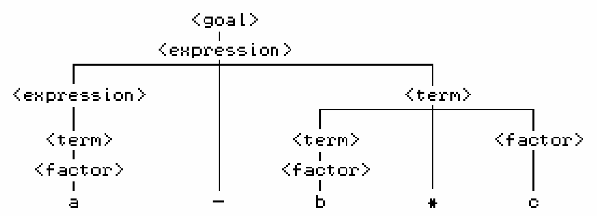


Figure 5.2 Parse tree for the expression a - b * c

Exercises

5.8 What would be the shortest sentence in the language defined by our first example? What would be the longest sentence? Would there be a difference if we used the alternative productions (3a, 3b)?

5.9 Draw the phrase structure trees that correspond to the expressions *a* - *b* - *c* and *a* * *b* * *c* using the second grammar.

5.10 Try to extend the grammar for expressions so as to incorporate the + and / operators.

5.7 Phrase structure and lexical structure

It should not take much to see that a set of productions for a real programming language grammar will usually divide into two distinct groups. In such languages we can distinguish between the productions that specify the **phrase structure** - the way in which the words or tokens of the language are combined to form components of programs - and the productions that specify the **lexical structure** or **lexicon** - the way in which individual characters are combined to form such words or tokens. Some tokens are easily specified as simple constant strings standing for themselves. Others are more generic - lexical tokens such as identifiers, literal constants, and strings are themselves specified by means of productions (or, in many cases, by regular expressions).

As we have already hinted, the recognition of tokens for a real programming language is usually done by a scanner (lexical analyser) that returns these tokens to the parser (syntax analyser) on demand. The productions involving only individual characters on their right sides are thus the productions used by a sub-parser forming part of the lexical analyser, while the others are productions used by the main parser in the syntax analyser.

5.8 ε-productions

The alternatives for the right-hand side of a production usually consist of a string of one or more terminal and/or non-terminal symbols. At times it is useful to be able to derive an empty string, that is, one consisting of no symbols. This string is usually denoted by ε when it is necessary to reveal its presence explicitly. For example, the set of productions

```
<unsigned integer>    →    <digit> <rest of integer>
<rest of integer>     →    <digit> <rest of integer> | ε
<digit>               →    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

defines <rest of integer> as a sequence of zero or more <digit>s, and hence <unsigned integer> is defined as a sequence of one or more <digit>s. In terms of our earlier notation we should have

$$\text{<rest of integer>} \to \text{<digit>}^{*}$$

or

$$\text{<unsigned integer>} \to \text{<digit>}^{+}$$

The production

$$\text{<rest of integer>} \to \varepsilon$$

is called a **null production**, or an ε-production, or sometimes a **lambda production** (from an alternative convention of using λ instead of ε for the null string). Applying a production of the form $L \to \varepsilon$ amounts to the erasure of the non-terminal L from a sentential form; for this reason such productions are sometimes called *erasures*. More generally, if for some string σ it is possible that

$$\sigma \Rightarrow^{*} \varepsilon$$

then we say that σ is *nullable*. A non-terminal L is said to be nullable if it has a production whose definition (right side) is nullable.

5.9 Extensions to BNF

Various simple extensions are often employed with BNF notation for the sake of increased readability and for the elimination of unnecessary recursion (which has a strange habit of confusing people brought up on iteration). Recursion is often employed in BNF as a means of specifying simple repetition, as for example

```
        <unsigned integer>  →  <digit> | <digit> <unsigned integer>
```

(which uses right recursion) or

```
        <unsigned integer>  →  <digit> | <unsigned integer> <digit>
```

(which uses left recursion).

Then we often find several productions used to denote alternatives which are very similar, for example

```
       <integer>            →  <unsigned integer> | <sign> <unsigned integer>
       <unsigned integer>   →  <digit> | <digit> <unsigned integer>
       <sign>               →  + | -
```

using six productions (besides the omitted obvious ones for <digit>) to specify the form of an <integer>.

The extensions introduced to simplify these constructions lead to what is known as **EBNF** (Extended BNF). There have been many variations on this, most of them inspired by the metasymbols used for regular expressions. Thus we might find the use of the Kleene closure operators to denote repetition of a symbol zero or more times, and the use of round brackets or parentheses ( ) to group items together.

Using these ideas we might define an integer by

```
       <integer>           →  <sign> <unsigned integer>
       <unsigned integer>  →  <digit> ( <digit> )*
       <sign>              →  + | - | ε
```

or even by

```
              <integer>    →   ( + | - | ε )  <digit> ( <digit> )*
```

which is, of course, nothing other than a regular expression anyway. In fact, a language that can be expressed as a regular expression can always be expressed in a single EBNF expression.

5.9.1 Wirth's EBNF notation

In defining Pascal and Modula-2, Wirth came up with one of these many variations on BNF which has now become rather widely used (Wirth, 1977). Further metasymbols are used, so as to express more succinctly the many situations that otherwise require combinations of the Kleene closure operators and the ε string. In addition, further simplifications are introduced to facilitate the automatic processing of productions by parser generators such as we shall discuss in a later section. In this notation for EBNF:

`Non-terminals`	are written as single words, as in *VarDeclaration* (rather than the <Var Declaration> of our previous notation)
`Terminals`	are all written in quotes, as in "`BEGIN`" (rather than as themselves, as in BNF)
`|`	is used, as before, to denote alternatives
`(   )`	(parentheses) are used to denote grouping
`[   ]`	(brackets) are used to denote the optional appearance of a

	symbol or group of symbols
`{  }`	(braces) are used to denote optional repetition of a symbol or group of symbols
`=`	is used in place of the ::= or $\rightarrow$ symbol
`.`	is used to denote the end of each production
`(*  *)`	are used in some extensions to allow comments
ε	can be handled by using the [ ] notation
`spaces`	are essentially insignificant.

For example

```
Integer         =  Sign UnsignedInteger .
UnsignedInteger =  digit  {  digit  } .
Sign            =  [ "+" | "-" ] .
digit           =  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
```

The effect is that non-terminals are less "noisy" than in the earlier forms of BNF, while terminals are "noisier". Many grammars used to define programming language employ far more non-terminals than terminals, so this is often advantageous. Furthermore, since the terminals and non-terminals are textually easily distinguishable, it is usually adequate to give only the set of productions P when writing down a grammar, and not the complete quadruple $\{ N, T, S, P \}$.

As another example of the use of this notation we show how to describe a set of EBNF productions in EBNF itself:

```
EBNF        =  { Production } .
Production  =  nonterminal "=" Expression "." .
Expression  =  Term { "|" Term } .
Term        =  Factor { Factor } .
Factor      =      nonterminal | terminal | "[" Expression "]"
                | "(" Expression ")" | "{" Expression "}" .
nonterminal =  letter { letter } .
terminal    =  "'" character { character } "'" | '"' character { character } '"' .
character   =  (* implementation defined *)  .
```

Here we have chosen to spell *nonterminal* and *terminal* in lower case throughout to emphasize that they are lexical non-terminals of a slightly different status from the others like *Production*, *Expression*, *Term* and *Factor*.

A variation on the use of braces allows the (otherwise impossible) specification of a limit on the number of times a symbol may be repeated - for example to express that an identifier in Fortran may have a maximum of six characters. This is done by writing the lower and upper limits as sub- and super-scripts to the right of the curly braces, as for example

$$FortranIdentifier \rightarrow letter \ \{ \ letter \mid digit \ \}_0^5$$

5.9.2 Semantic overtones

Sometimes productions are developed to give semantic overtones. As we shall see in a later section, this leads more easily towards the possibility of extending or *attributing* the grammar to incorporate a formal semantic specification along with the syntactic specification. For example, in describing Modula-2, where expressions and identifiers fall into various classes at the static semantic level, we might find among a large set of productions:

```
ConstDeclarations    =  "CONST"
                           ConstIdentifier "=" ConstExpression ";"
                           { ConstIdentifier "=" ConstExpression ";" } .
ConstIdentifier      =  identifier .
```

```
 ConstExpression      =  Expression .
```

5.9.3 The British Standard for EBNF

The British Standards Institute has a published standard for EBNF (BS6154 of 1981). The BSI standard notation is noisier than Wirth's one: elements of the productions are separated by commas, productions are terminated by semicolons, and spaces become insignificant. This means that compound words like *ConstIdentifier* are unnecessary, and can be written as separate words. An example in BSI notation follows:

```
 Constant Declarations  =  "CONST",
                               Constant Identifier, "=", Constant Expression, ";",
                             { Constant Identifier, "=", Constant Expression, ";" }  ;
 Constant Identifier    =  identifier ;
 Constant Expression    =  Expression ;
```

5.9.4 Lexical and phrase structure emphasis

We have already commented that real programming language grammars have a need to specify phrase structure as well as lexical structure. Sometimes the distinction between "lexical" and "syntactic" elements is taken to great lengths. For example we might find:

```
 ConstDeclarations    =  constSym
                            ConstIdentifier equals ConstExpression semicolon
                            { ConstIdentifier equals ConstExpression semicolon } .
```

with productions like

```
  constSym            =  "CONST" .
  semicolon           =  ";" .
  equals              =  "=" .
```

and so on. This may seem rather long-winded, but there are occasional advantages, for example in allowing alternatives for limited character set machines, as in

```
  leftBracket         =  "["  |  "(." .
  pointerSym          =  "^"  |  "@" .
```

as is used in some Pascal systems.

5.9.5 Cocol

The reader will recall from Chapter 2 that compiler writers often make use of compiler generators to assist with the automated construction of parts of a compiler. Such tools usually take as input an augmented description of a grammar, one usually based on a variant of the EBNF notations we have just been discussing. We stress that far more is required to construct a compiler than a description of syntax - which is, essentially, all that EBNF can provide. In later chapters we shall describe the use of a specific compiler generator, Coco/R, a product that originated at the University of Linz in Austria (Rechenberg and Mössenböck, 1989, Mössenböck, 1990a,b). The name Coco/R is derived from "**Co**mpiler-**Co**mpiler/**R**ecursive descent. A variant of Wirth's EBNF known as Cocol/R is used to define the input to Coco/R, and is the notation we shall prefer in the rest of this text (to avoid confusion between two very similar acronyms we shall simply refer to Cocol/R as Cocol). Cocol draws a clear distinction between lexical and phrase structure, and also makes clear provision for describing the character sets from which lexical tokens are constructed.

A simple example will show the main features of a Cocol description. The example describes a calculator that is intended to process a sequence of simple four-function calculations involving decimal or hexadecimal whole numbers, for example `3 + 4 * 8 =` or `$3F / 7 + $1AF =`.

```
COMPILER Calculator

CHARACTERS
  digit      = "0123456789" .
  hexdigit   = digit + "ABCDEF" .

IGNORE CHR(1) .. CHR(31)

TOKENS
  decNumber  = digit { digit } .
  hexNumber  = "$" hexdigit { hexdigit } .

PRODUCTIONS
  Calculator = { Expression "=" } .
  Expression = Term { "+" Term  |  "-" Term } .
  Term       = Factor { "*" Factor |  "/" Factor } .
  Factor     = decNumber | hexNumber .
END Calculator.
```

The `CHARACTERS` section describes the set of characters that can appear in decimal or hexadecimal digit strings - the right sides of these productions are to be interpreted as defining sets. The `TOKENS` section describes the valid forms that decimal and hexadecimal numbers may take - but notice that we do not, at this stage, indicate how the values of these numbers are to be computed from the digits. The `PRODUCTIONS` section describes the phrase structure of the calculations themselves - again without indicating how the results of the calculations are to be obtained.

At this stage it will probably come as no surprise to the reader to learn that Cocol, the language of the input to Coco/R, can itself be described by a grammar - and, indeed, we may write this grammar in a way that it could be processed by Coco/R itself. (Using Coco/R to process its own grammar is, of course, just another example of the bootstrapping techniques discussed in Chapter 3; Coco/R is another good example of a self-compiling compiler). A full description of Coco/R and Cocol appears later in this text, and while the finer points of this may currently be beyond the reader’s comprehension, the following simplified description will suffice to show the syntactic elements of most importance:

```
COMPILER Cocol

CHARACTERS
  letter       = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit        = "0123456789" .
  tab          = CHR(9) .
  cr           = CHR(13) .
  lf           = CHR(10) .
  noQuote2     = ANY - ’"’ - cr - lf .
  noQuote1     = ANY - "’" - cr - lf .

IGNORE tab + cr + lf

TOKENS
  identifier   = letter { letter | digit } .
  string       = ’"’ { noQuote2 } ’"’ | "’" { noQuote1 } "’" .
  number       = digit { digit } .

PRODUCTIONS
  Cocol        = "COMPILER"  Goal
                    [ Characters ]
                    [ Ignorable ]
                    [ Tokens ]
                    Productions
                 "END" Goal "." .
  Goal         = identifier .

  Characters   = "CHARACTERS" { NamedCharSet } .
  NamedCharSet = SetIdent "=" CharacterSet "." .
  CharacterSet = SimpleSet { "+" SimpleSet | "-" SimpleSet } .
  SimpleSet    = SetIdent | string | SingleChar [ ".." SingleChar ] | "ANY" .
  SingleChar   = "CHR" "(" number ")" .
  SetIdent     = identifier .

  Ignorable    = "IGNORE" CharacterSet .

  Tokens       = "TOKENS" { Token } .
  Token        = TokenIdent "=" TokenExpr "."  .
```

```
    TokenExpr    = TokenTerm { "|" TokenTerm } .
    TokenTerm    = TokenFactor { TokenFactor } [ "CONTEXT" "(" TokenExpr ")" ] .
    TokenFactor  =   TokenSymbol | "(" TokenExpr ")" | "[" TokenExpr "]"
                   | "{" TokenExpr "}" .
    TokenSymbol  = SetIdent | string .
    TokenIdent   = identifier .

    Productions  = "PRODUCTIONS" { Production } .
    Production   = NonTerminal "=" Expression  "." .
    Expression   = Term { "|" Term } .
    Term         = Factor { Factor } .
    Factor       =   Symbol | "(" Expression ")" | "[" Expression "]"
                   | "{" Expression "}" .
    Symbol       = string | NonTerminal | TokenIdent .
    NonTerminal  = identifier .

  END Cocol.
```

The following points are worth emphasizing:

- The productions in the `TOKENS` section specify identifiers, strings and numbers in the usual simple way.

- The first production (for Cocol) shows the overall form of a grammar description as consisting of four sections, the first three of which are all optional (although they are usually present in practice).

- The productions for `CharacterSets` show how character sets may be given names (*SetIdents*) and values (of *SimpleSets*).

- The production for `Ignorable` allows certain characters - typically line feeds and other unimportant characters - to be included in a set that will simply be ignored by the scanner when it searches for and recognizes tokens.

- The productions for `Tokens` show how tokens (terminal classes) may be named (*TokenIdents*) and defined by expressions in EBNF. Careful study of the semantic overtones of these productions will show that they are not self-embedding - that is, one token may not be defined in terms of another token, but only as a quoted string, or in terms of characters chosen from the named character sets defined in the `CHARACTERS` section. This amounts, in effect, to defining these tokens by means of regular expressions, even though the notation used is not the same as that given for regular expressions in section 5.3.

- The productions for `Productions` show how we define the phrase structure by naming *NonTerminals* and expressing their productions in EBNF. Notice that here we *are* allowed to have self-embedding and recursive productions. Although terminals may again be specified directly as strings, we are not allowed to use the names of character sets as symbols in the productions.

- Although it is not specified by the grammar above, one non-terminal must have the same identifier name as the grammar itself to act as the goal symbol (and, of course, all identifiers must be "declared" properly).

- It is possible to write input in Cocol that is syntactically correct (in terms of the grammar above) but which cannot be fully processed by Coco/R because it does not satisfy other constraints. This topic will be discussed further in later sections.

We stress again that Coco/R input really specifies *two* grammars. One is the grammar specifying the non- terminals for the lexical analyser (`TOKENS`) and the other specifies non-terminals for the higher level phrase structure grammar used by the syntax analyser (`PRODUCTIONS`). However, terminals may also be implicitly declared in the productions section. So the following, in one sense, may appear to be equivalent:

```
COMPILER Sample   (* one *)

CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .

TOKENS
  ident = letter { letter } .

PRODUCTIONS
  Sample = "BEGIN" ident ":=" ident "END" .

END Sample .
--------------------------------------------------------------------

COMPILER Sample   (* two *)

CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .

TOKENS
  Letter = letter .

PRODUCTIONS
  Sample = "BEGIN" Ident ":=" Ident "END" .
  Ident  = Letter { Letter } .

END Sample .
--------------------------------------------------------------------

COMPILER Sample   (* three *)

CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .

TOKENS
  ident = letter { letter } .
  begin = "BEGIN" .
  end   = "END" .
  becomes = ":=" .

PRODUCTIONS
  Sample = begin ident becomes ident end .

END Sample .
```

Actually they are not quite the same. Since Coco/R always ignores spaces (other than in strings), the second one would treat the input

```
A C E := S P A D E
```

as the first would treat the input

```
ACE := SPADE
```

The best simple rule seems to be that one should declare under `TOKENS` any class of symbol that has to be recognized as a contiguous string of characters, and of which there may be several instances (this includes entities like identifiers, numbers, and strings) - as well as special character terminals (like `EOL`) that cannot be graphically represented as quoted characters. Reserved keywords and symbols like "`:=`" are probably best introduced as terminals implicitly declared in the `PRODUCTIONS` section. Thus grammar (1) above is probably the best so far as Coco/R is concerned.

Exercises

5.11 Develop simple grammars to describe each of the following

(a) A person’s name, with optional title and qualifications (if any), for example

S.B. Terry , BSc
Master Kenneth David Terry
Helen Margaret Alice Terry

(b) A railway goods train, with one (or more) locomotives, several varieties of trucks, and a guard’s van at the rear.

(c) A mixed passenger and goods train, with one (or more) locomotives, then one or more goods trucks, followed either by a guard’s van, or by one or more passenger coaches, the last of which should be a passenger brake van. In the interests of safety, try to build in a regulation to the effect that fuel trucks may not be marshalled immediately behind the locomotive, or immediately in front of a passenger coach.

(d) A book, with covers, contents, chapters and an index.

(e) A shopping list, with one or more items, for example

3 Practical assignments
124 bottles Castle Lager
12 cases Rhine Wine
large box aspirins

(f) Input to a postfix (reverse Polish) calculator. In postfix notation, brackets are not used, but instead the operators are placed after the operands.

For example,

```
infix expression             reverse Polish equivalent

6 + 9 =                      6 9 + =
(a + b) * (c + d)            a b + c d + *
```

(g) A message in Morse code.

(h) Unix or MS-DOS file specifiers.

(i) Numbers expressed in Roman numerals.

(j) Boolean expressions incorporating conjunction (OR), disjunction (AND) and negation (NOT).

5.12 Develop a Cocol grammar using only BNF-style productions that defines the rules for expressing a set of BNF productions.

5.13 Develop a Cocol grammar using only BNF-style productions that defines the rules for expressing a set of EBNF productions.

5.14 Develop an EBNF grammar that defines regular expressions as described in section 5.3.

5.15 What real practical advantage does the Wirth notation using [ ] and { } afford over the use of the Kleene closure symbols?

5.16 In yet another variation on EBNF ε can be written into an empty right side of a production explicitly, in addition to being handled by using the [ ] notation, for example:

```
            Sign = "+" | "-" | .  (* the ε or null is between the last | and . *)
```

Productions like this cannot be described by the productions for EBNF given in section 5.9.1. Develop a Cocol grammar that describes EBNF productions that *do* allow an empty string to appear implicitly.

5.17 The local Senior Citizens Association make a feature of Friday evenings, when they employ a mediocre group to play for dancing. At such functions the band perform a number of selections, interspersed with periods of silence which are put to other good use. The band have only four kinds of selection at present. The first of these consists of waltzes - such a selection always starts with a slow waltz, which may be followed by several more slow waltzes, and finally (but only if the mood of the evening demands it) by one or more fast waltzes. The second type of selection consists of several Rock'n'Roll numbers. The third is a medley, consisting of a number of tunes of any sort played in any order. The last is the infamous "Paul Jones", which is a special medley in which every second tune is "Here we go round the mulberry bush". During the playing of this, the dancers all pretend to change partners, in some cases actually succeeding in doing so. Develop a grammar which describes the form that the evening assumes.

5.18 Scottish pipe bands often compete at events called Highland Gatherings where three forms of competition are traditionally mounted. There is the so-called "Slow into Quick March" competition, in which each band plays a single Slow March followed by a single Quick March. There is the so-called "March, Strathspey and Reel" competition, where each band plays a single Quick March, followed by a single Strathspey, and then by a single Reel; this set may optionally be followed by a further Quick March. And there is also the "Medley", in which a band plays a selection of tunes in almost any order. Each tune fall into one of the categories of March, Strathspey, Reel, Slow March, Jig and Hornpipe but, by tradition, a group of one or more Strathspeys within such a medley is always followed by a group of one or more Reels.

Develop a grammar to describe the activity at a Highland Gathering at which a number of competitions are held, and in each of which at least one band performs. Competitions are held in one category at a time. Regard concepts like "March", "Reel" and so on as terminals - in fact there are many different possible tunes of each sort, but you may have to be a piper to recognize one tune from another.

5.19 Here is an extract from the index of my forthcoming bestseller "Hacking out a Degree":

abstract class 12, 45
abstraction, data 165
advantages of Modula-2 1-99, 100-500, Appendix 4
aegrotat examinations -- see unethical doctors
class attendance, intolerable 745
deadlines, compiler course -- see sunrise
horrible design (C and C++) 34, 45, 85-96
lectures, missed 1, 3, 5-9, 12, 14-17, 21-25, 28
recursion -- see recursion
senility, onset of 21-24, 105

subminimum 30
supplementary exams 45 - 49
wasted years 1996-1998

Develop a grammar that describes this form of index.

5.20 You may be familiar with the "make" facility that is found on Unix (and sometimes on MS-DOS) for program development. A "make file" consists of input to the `make` command that typically allows a system to be re-built correctly after possibly modifying some of its component parts. A typical example for a system involving C++ compilation is shown below. Develop a grammar that describes the sort of make files that you may have used in your own program development.

```
# makefile for maintaining my compiler
CFLAGS = -Wall
CC     = g++
HDRS   = parser.h scanner.h generator.h
SRCS   = compiler.cpp \
          parser.cpp scanner.cpp generator.cpp
OBJS   = compiler.o parser.o scanner.o generator.o

%.o: %.cpp $(HDRS)
        $(CC) -c $(CFLAGS) $<

all:    compiler

new:    clean compiler

cln:    $(OBJS)
        $(CC) -o cln $(CFLAGS) $(OBJS)

clean:
        rm *.o
        rm compiler
```

5.21 C programmers should be familiar with the use of the standard functions `scanf` and `printf` for performing input and output. Typical calls to these functions are

```
scanf("%d %s %c", &n, string, &ch);
printf("Total = %-10.4d\nProfit = %d\%%\n", total, profit);
```

in which the first argument is usually a literal string incorporating various specialized format specifiers describing how the remaining arguments are to be processed.

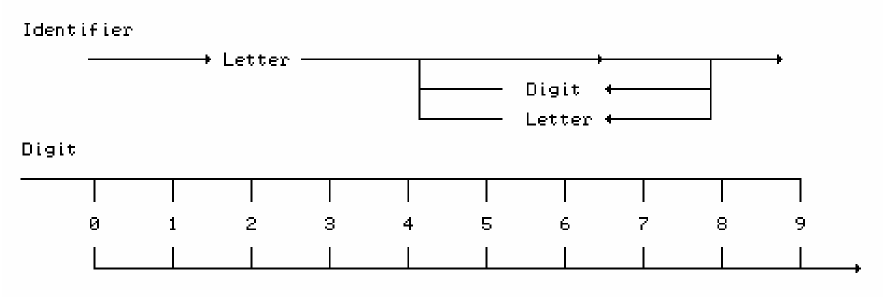
Develop a grammar that describes such statements as fully as you can. For simplicity restrict yourself to the situation where any arguments after the first refer to simple variables.

Further reading

The use of BNF and EBNF notation is covered thoroughly in all good books on compilers and syntax analysis. Particularly useful insight will be found in the books by Watt (1991), Pittman and Peters (1992) and Gough (1988).

5.10 Syntax diagrams

An entirely different method of syntax definition is by means of the graphic representation known as syntax diagrams, syntax charts, or sometimes "railroad diagrams". These have been used to define the syntax of Pascal, Modula-2 and Fortran 77. The rules take the form of flow diagrams, the possible paths representing the possible sequences of symbols. One starts at the left of a diagram, and traces a path which may incorporate terminals, or incorporate transfers to other diagrams if a word is reached that corresponds to a non-terminal. For example, an identifier might be defined by



with a similar diagram applying to `Letter`, which we can safely assume readers to be intelligent enough to draw for themselves.

Exercises

5.22 Attempt to express some of the solutions to previous exercises in terms of syntax diagrams.

5.11 Formal treatment of semantics

As yet we have made no serious attempt to describe the semantics of programs written in any of our "languages", and have just assumed that these would be self-evident to a reader who already has come to terms with at least one imperative language. In one sense this is satisfactory for our purposes, but in principle it is highly unsatisfactory not to have a simple, yet rigidly formal means of specifying the semantics of a language. In this section we wish to touch very briefly on ways in which this might be achieved.

We have already commented that the division between syntax and semantics is not always clear-cut, something which may be exacerbated by the tendency to specify productions using names with clearly semantic overtones, and whose sentential forms already reflect meanings to be attached to operator precedence and so on. When specifying semantics a distinction is often attempted between what is termed **static semantics** - features which, in effect, mean something that can be checked at compile-time, such as the requirement that one may not branch into the middle of a **procedure**, or that assignment may only be attempted if type checking has been satisfied - and **dynamic semantics** - features that really only have meaning at run-time, such as the effect of a branch statement on the flow of control, or the effect of an assignment statement on elements of storage.

Historically, attempts formally to specify semantics did not meet with the same early success as those which culminated in the development of BNF notation for specifying syntax, and we find that the semantics of many, if not most, common programming languages have been explained in terms of a natural language document, often regrettably imprecise, invariably loaded with jargon, and difficult to follow (even when one has learned the jargon). It will suffice to give two examples:

(a) In a draft description of Pascal, the syntax of the **with** statement was defined by

```
          <with-statement> ::=  with <record-variable-list> do <statement>
    <record-variable-list> ::=  <record-variable> { , <record-variable> }
    <variable-identifier>  ::=  <field-identifier>
```

with the commentary that

"The occurrence of a <record-variable> in the <record-variable-list> is a defining occurrence of its <field-identifier>s as <variable-identifier>s for the <with-statement> in which the <record-variable-list> occurs."

The reader might be forgiven for finding this awkward, especially in the way it indicates that within the <statement> the <field-identifier>s may be used as though they were <variable-identifier>s.

(b) In the same description we find the **while** statement described by

<while-statement> : : = **while** <Boolean-expression> **do** <statement>

with the commentary that

"The <statement> is repeatedly executed while the <Boolean-expression> yields the value TRUE. If its value is FALSE at the beginning, the <statement> is not executed at all. The <while- statement>

while *b* **do** *body*

is equivalent to

if *b* **then repeat** *body* **until not** *b*."

If one is to be very critical, one might be forgiven for wondering what exactly is meant by "beginning" (does it mean the beginning of the program, or of execution of this one part of the program). One might also conclude, especially from all the emphasis given to the effect when the <Boolean-expression> is initially FALSE, that in that case the <while-statement> is completely equivalent to an empty statement. This is not necessarily true, for evaluation of the <Boolean-expression> might require calls to a function which has side-effects; nowhere (at least in the vicinity of this description) was this point mentioned.

The net effect of such imprecision and obfuscation is that users of a language often resort to writing simple test programs to help them understand language features, that is to say, they use the operation of the machine itself to explain the language. This is a technique which can be disastrous on at least two scores. In the first place, the test examples may be incomplete, or too special, and only a half-truth will be gleaned. Secondly, and perhaps more fundamentally, one is then confusing an abstract language with one concrete implementation of that language. Since implementations may be error prone, incomplete, or, as often happens, may have extensions that do not form part of the standardized language at all, the possibilities for misconception are enormous.

However, one approach to formal specification, known as **operational semantics** essentially refines this ad-hoc arrangement. To avoid the problems mentioned above, the (written) specification usually describes the action of a program construction in terms of the changes in state of an abstract machine which is supposed to be executing the construction. This method was used to specify the language PL/I, using the metalanguage **VDL** (Vienna Definition Language). Of course, to understand such specifications, the reader has to understand the definition of the abstract machine, and not only might this be confusingly theoretical, it might also be quite unlike the actual machines which he or she has encountered. As in all semantic descriptions, one is simply shifting the problem of "meaning" from one area to another. Another drawback of this approach is that it tends to obscure the semantics with a great detail of what is essentially useful knowledge for the implementor of the language, but almost irrelevant for the user of the same.

Another approach makes use of **attribute grammars**, in which the syntactic description (in terms of EBNF) is augmented by a set of distinct attributes *V* (each one associated with a single terminal or non-terminal) and a set of assertions or predicates involving these attributes, each assertion being associated with a single production. We shall return to this approach in a later chapter, for it forms the basis of practical applications of several compiler generators, among them Coco/R.

Other approaches taken to specifying semantics tend to rely rather more heavily on mathematical logic and mathematical notation, and for this reason may be almost impossible to understand if the programmer is one of the many thousands whose mathematical background is comparatively weak. **Denotational semantics**, for example defines programs in terms of mappings into mathematical operations and constructs: a program is simply a function that maps its input data to its output data, and its individual component statements are functions that map an environment and store to an updated store. A variant of this, **VDM** (Vienna Definition Method), has been used in formal specifications of Ada, Algol-60, Pascal and Modula-2. These specifications are long and difficult to follow (that for Modula-2 runs to some 700 pages).

Another mathematically based method, which was used by Hoare and Wirth (1973) to specify the semantics of most of Pascal, uses so-called **axiomatic semantics**, and it is worth a slight digression to examine the notation used. It is particularly apposite when taken in conjunction with the subject of **program proving**, but, as will become apparent, rather limited in the way in which it specifies what a program actually seems to be doing.

In the notation, *S* is used to represent a statement or statement sequence, and letters like *P*, *Q* and *R* are used to represent **predicates**, that is, the logical values of Boolean variables or expressions. A notation like

$$\{ P \} \; S \; \{ Q \}$$

denotes a so-called **inductive expression**, and is intended to convey that if *P* is true before *S* is executed, then *Q* will be true after *S* terminates (assuming that it does terminate, which may not always happen).

P is often called the **precondition** and *Q* the **postcondition** of *S*. Such inductive expressions may be concatenated with logical operations like $\wedge$ (*and*) and $\neg$ (*not*) and $\Rightarrow$ (*implies*) to give expressions like

$$\{ P \} \; S_1 \; \{ Q \} \wedge \{ Q \} \; S_2 \; \{ R \}$$

from which one can infer that

$$\{ P \} \; S_1 \; ; S_2 \; \{ R \}$$

which is written more succinctly as a **rule of inference**

$$\frac{\{ P \}\ S_1\ \{ Q \} \wedge \{ Q \}\ S_2\ \{ R \}}{\{ P \}\ S_1 ; S_2\ \{ R \}}$$

Expressions like

$P \Rightarrow Q$ and $\{ Q \}\ S\ \{ R \}$

and

$\{ P \}\ S\ \{ Q \}$ and $Q \Rightarrow R$

lead to the **consequence rules**

$$\frac{P \Rightarrow Q \text{ and } \{ Q \}\ S\ \{ R \}}{\{ P \}\ S\ \{ R \}}$$

and

$$\frac{\{ P \}\ S\ \{ Q \} \text{ and } Q \Rightarrow R}{\{ P \}\ S\ \{ R \}}$$

In these rules, the top line is called the **antecedent** and the bottom one is called the **consequent**; so far as program proving is concerned, to prove the truth of the consequent it is necessary only to prove the truth of the antecedent.

In terms of this notation one can write down rules for nearly all of Pascal remarkably tersely. For example, the **while** statement can be described by

$$\frac{\{ P \wedge \mathrm{B} \}\ S\ \{ P \}}{\{ P \}\ \textbf{while}\ \mathrm{B}\ \textbf{do}\ S\ \{ P \wedge \neg \mathrm{B} \}}$$

and the **if** statements by

$$\frac{\{ P \wedge \mathrm{B} \}\ S\ \{ Q \} \text{ and } P \wedge \neg \mathrm{B} \Rightarrow Q}{\{ P \}\ \textbf{if}\ \mathrm{B}\ \textbf{then}\ S\ \{ Q \}}$$

$$\frac{\{ P \wedge \mathrm{B} \}\ S_1\ \{ Q \} \text{ and } \{ P \wedge \neg \mathrm{B} \}\ S_2\ \{ Q \}}{\{ P \}\ \textbf{if}\ \mathrm{B}\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{ Q \}}$$

With a little reflection one can understand this notation quite easily, but it has its drawbacks. Firstly, the rules given are valid only if the evaluation of *B* proceeds without side-effects (compare the discussion earlier). Secondly, there seems to be no explicit description of what the machine implementing the program actually does to alter its state - the idea of "repetition" in the rule for the **while** statement probably does not exactly strike the reader as obvious.

Further reading

In what follows we shall, perhaps cowardly, rely heavily on the reader’s intuitive grasp of semantics. However, the keen reader might like to follow up the ideas germinated here. So far as natural language descriptions go, a draft description of the Pascal Standard is to be found in the article by Addyman *et al* (1979). This was later modified to become the ISO Pascal Standard, known variously as ISO 7185 and BS 6192, published by the British Standards Institute, London (a copy is given as an appendix to the book by Wilson and Addyman (1982)). A most readable guide to the Pascal Standard was later produced by Cooper (1983). Until a standard for C++ is completed, the most precise description of C++ is probably the "ARM" (Annotated Reference Manual) by Ellis and Stroustrup (1990), but C++ has not yet stabilized fully (in fact the standard appeared shortly after this book was published). In his book, Brinch Hansen (1983) has a very interesting chapter on the problems he encountered in trying to specify Edison completely and concisely.

The reader interested in the more mathematically based approach will find useful introductions in the very readable books by McGettrick (1980) and Watt (1991). Descriptions of VDM and specifications of languages using it are to be found in the book by Bjorner and Jones (1982). Finally, the text by Pittman and Peters (1992) makes extensive use of attribute grammars.