Chapter 2
How to Describe a Programming Language

A programming language is an artificial formalism in which algorithms can be expressed. For all its artificiality, though, this formalism remains a *language*. Its study can make good use of the many concepts and tools developed in the last century in linguistics (which studies both natural and artificial languages). Without going into great detail, this chapter poses the problem of what it means to "give" (define or understand) a programming language and which tools can be used in this undertaking.

2.1 Levels of Description

In a study which has now become a classic in linguistics, Morris [6] studied the various levels at which a description of a language can occur. He identified three major areas: *grammar*, *semantics* and *pragmatics*.

Grammar is that part of the description of the language which answers the question: which phrases are correct? Once the alphabet of a language has been defined as a first step (in the case of natural language, for example, the Latin alphabet of 22 or 26 letters, the Cyrillic alphabet, etc.), the *lexical* component, which uses this alphabet, identifies the sequence of symbols constituting the *words* (or *tokens*) of the language defined. When alphabet and words have been defined, the *syntax* describes which sequences of words constitute legal phrases. Syntax is therefore a relation between signs. Between all possible sequences of words (over a given alphabet), the syntax chooses a subset of sequences to form phrases of the language proper.[1]

[1] In linguistics, obviously, things are more complicated. In addition to the lexical and the syntactic levels, there is also a morphological level which is distinct from the two previous ones. At the morphological level, the different forms assumed by words (or phrases) as a function of their grammatical function are defined. For example in the lexicon (that is, the dictionary, the thesaurus of the language in question), we find the word "bear" with the associated lexical value given by the image of the animal that everybody knows. At the morphological level, on the other hand, the word is the convertible into the root "bear" and the morpheme "-s" which signals the plural. The natural language's phonetic aspects are also present but they are of no interest to us here.

M. Gabbrielli, S. Martini, *Programming Languages: Principles and Paradigms,*
Undergraduate Topics in Computer Science,
DOI 10.1007/978-1-84882-914-5_2,

Semantics is that part of the description of the language which seeks to answer the question "what does a correct phrase mean?" Semantics, therefore, attributes a *significance* to every correct phrase. In the case of natural languages, the process of attribution of meaning can be very complex; in the case of artificial languages the situation is rather simpler. It is not difficult to assume, in this case, that semantics is a relation between signs (correct sentences) and meanings (autonomous entities existing independently of the signs that are used to describe them). For example, the meaning of a certain program could be the mathematical function computed by that program. The semantic description of that language will be constructed using techniques allowing us, when given a program, to fix the function the program computes.

It is at the third level that the principal actor makes its appearance on the scene, the person who uses a certain language. *Pragmatics* is that part of a language description which asks itself "how do we use a meaningful sentence?" Sentences with the same meaning can be used in different ways by different users. Different linguistic contexts can require the use of different sentences; some are more elegant, some are antiquated, or more dialect-based than others. Understanding these linguistic mechanisms is no less important than knowing the syntax and semantics.

In the case of programming languages, we can add a fourth level to the three classical ones: the *implementation* level. Given that the languages that interest us are procedural languages (that is, languages whose correct phrases specify actions), it remains for us to describe "how to execute a correct sentence, in such a way that we respect the semantics". A knowledge of the semantics is, in general, enough for the language user, but the software designer (and more importantly the language designer) is also interested in the process with which operative phrases implement the state under consideration. It is precisely this which is described by the language implementation.

We can give a fairly rudimentary example which we hope will serve our purposes. Let us consider the natural language used to express recipes in cooking. The syntax determines the correct sentences with which a recipe is expressed. The semantics is about explaining "what is" a recipe, independent of its (specific) execution. Pragmatics studies how a cook ("that cook") interprets the various sentences of the recipe. In the end, the implementation describes the way (where, and with what ingredients) the kitchen recipe transforms into the dish that the semantics prescribes.

In the next sections, we analyse the role performed by the four levels when they are applied to programming languages.

2.2 Grammar and Syntax

We have already said that the grammar of a language first establishes the alphabet and lexicon. Then by means of a syntax, it defines those sequences of symbols corresponding to well-formed phrases and sentences (or to "sentences" in short). Clearly, at least from the viewpoint of natural language, the definition of the (finite) alphabet is immediate. Even the lexicon can be defined, at least to a first approximation, in

a simple fashion. We will be content, for the time being, with a finite vocabulary; we can simply list the words of interest to us. This is certainly what happens with natural languages, given that dictionaries are finite volumes![2]

How do we describe the syntax? In a natural language, it is the same natural language which is used, in general, to describe its own syntax (classical examples are grammar textbooks for various languages). Also a syntax for a programming language is often described using natural language, particularly for the older languages. Natural language, though, often introduces ambiguity in its syntactic description and, in particular, it is of no use in the process of translating (compiling) a program in a high-level language into another (generally, at a lower level) language.

Linguistics, on the other hand, through the work in the 1950s of the American linguist Noam Chomsky, has developed techniques to describe syntactic phenomena in a formal manner. This description uses formalisms designed specifically to limit the ambiguity that is always present in natural language. These techniques, known as generative grammar, are not of much use in describing the syntax of natural languages (which are too complex and highly sophisticated). Instead, they are a fundamental tool for describing the syntax of programming languages (and particularly their compilation, as we will briefly see in Sect. 2.4).

Example 2.1 We will describe a simple language. It is the language of palindromic strings, composed of the symbols a and b.[3] Let us therefore begin by fixing the *alphabet*, $A = \{a, b\}$. We must now select, from all strings over A (that is finite sequences of elements of A), the palindromic strings. The simplest way to do this is to observe that there is a simple recursive definition of a palindromic string. Indeed, we can say (this is the basis of the induction) that a and b are palindromic strings. If, then, s is any string that we know already to be palindromic, then so are asa and bsb (this is the induction step).

It is not difficult to convince ourselves that this definition captures all and only the palindromic strings of odd length over A. It remains to account for even-length strings, such as $abba$. To include these as well, we add the fact that the empty string (that is the string which contains no symbol from the alphabet) is also a palindromic string to the base case of the inductive definition. Now our definition categorises all and only the palindromic strings over the alphabet A. If a string really is a palindromic string, for example $aabaa$, or $abba$, there exists a sequence of applications of the inductive rule just mentioned which will construct it. On the other hand, if a string is not a palindromic string (for example $aabab$), there is no way to construct it inductively using the rules just introduced.

Context-Free Grammars, which we introduce in the next section, are a notation for the concise and precise expression of recursive definitions of strings such as

[2] In programming languages, the lexicon can also be formed from an infinite set of words. We will see below how this is possible.

[3] A string is palindromic if it is identical to its mirror image. That is, the string is the same when read from left to right or right to left. A famous example in Latin is, ignoring spaces, the riddle "in girum imus nocte et consumimur igni".

the one we have just seen. The inductive definition of palindromic strings can be expressed in grammatical form as:

$$
\begin{aligned}
P &\rightarrow \\
P &\rightarrow \mathbf{a} \\
P &\rightarrow \mathbf{b} \\
P &\rightarrow \mathbf{a}P\mathbf{a} \\
P &\rightarrow \mathbf{b}P\mathbf{b}
\end{aligned}
$$

In these rules, P stands for "any palindromic string", while the arrow $\rightarrow$ is read as "can be". The first three lines will be immediately recognised as the basis of the inductive definition, while the last two form the induction step.

2.2.1 Context-Free Grammars

The example just given is an example of a *context-free grammar*, a fundamental device for the description of programming languages. We begin by introducing a little notation and terminology. Having fixed a finite (or denumerable) set A, which we call the *alphabet*, we denote by A^* the set of all *finite strings over* A (that is finite length sequences of elements of A; the $*$ operator is called *Kleene's star*). Let us immediately observe that, according to the definition, the sequence of length zero also belongs to A^*—this is the empty string, denoted by ϵ.

A *formal language over the alphabet* A is nothing more than a subset of A^*. A formal grammar serves to identify a certain subset of strings from all those possible over a given alphabet.[4]

Definition 2.1 (Context-Free Grammar) A context-free grammar is a quadruple (NT, T, R, S) where:

1. NT is a finite set of symbols (non-terminal symbols, or variables, or syntactic categories).
2. T is a finite set of symbols (terminal symbols).
3. R is a finite set of *productions* (or *rules*), each of which is composed of an expression of the form:

   $$V \rightarrow w$$

   where V (the *head* of the production) is a single non-terminal symbol and w (the body) is a string composed of zero or more terminal or non-terminal symbols (that is w is a string over $T \cup NT$).

[4] In formal languages, a sequence of terminal symbols which appears in a language is usually called a "word" of the language. We do not use this terminology; instead we speak of strings, to avoid ambiguity with words and phrases of a language (natural or artificial), in the sense in which they are normally understood.

4. S is an element of NT (the *initial symbol*).

According to this definition, therefore, the grammar in the example shown in Example 2.1 is composed of the quadruple $(\{P\}, \{a, b\}, R, P)$, where R is the set of productions used in the example.

Let us observe that, according to the definition, a production can have an empty body, that is, it can be composed of the null symbol, or, more properly, can be formed from the *empty string*, ϵ. We will see in a later section how a given grammar defines a language over the alphabet formed from its terminal symbols. Let us begin however with an example.

Example 2.2 Let us see a simple grammar that describes arithmetic expressions (over the operators $+$, $*$, $-$, both unary and binary). The atomic elements of these expressions are simple identifiers formed from finite sequences of the symbols a and b.

We define the grammar $G = (\{E, I\}, \{\mathbf{a}, \mathbf{b}, +, *, -, (,)\}, R, E)$, where R is the following set of productions:

$$
\begin{aligned}
&1.\ E \to I \\
&2.\ E \to E + E \\
&3.\ E \to E * E \\
&4.\ E \to E - E \\
&5.\ E \to -E \\
&6.\ E \to (E)
\end{aligned}
$$

$$
\begin{aligned}
&7.\ I \to \mathbf{a} \\
&8.\ I \to \mathbf{b} \\
&9.\ I \to I\mathbf{a} \\
&10.\ I \to I\mathbf{b}
\end{aligned}
$$

Unlike the grammar in Example 2.1, this one has more than one non-terminal symbol corresponding, in an informal way, to an expression (E) or to an identifier (I). Note, once more, how the productions are a synthetic way to express recursive definitions. In this case, we are dealing with two definitions (which have been graphically separated), of which one (the one for E) uses in its base case the non-terminal symbol inductively defined using the other recursive definition.

BNF In the context of programming languages, context-free grammars were used for the first time in the definition of the language Algol60. In the report that introduced Algol60, the grammar that defines the language is described using a notation that is slightly different from the one we have just introduced. This notation, (considered, among other things, to be usable with a reduced character set, that does not include arrow, cursives, upper case, etc.) goes under the name of *Backus Naur normal form* (BNF), named after two authoritative members of the Algol committee (John Backus who had previously directed the Fortran project and had written the compiler for it—the first high-level language, and Peter Naur). In BNF:

Fig. 2.1 A derivation of the string $\mathbf{ab} * (\mathbf{a} + \mathbf{b})$

$$
\begin{aligned}
E &\Rightarrow_3 E * E \\
&\Rightarrow_1 I * E \\
&\Rightarrow_{10} I\mathbf{b} * E \\
&\Rightarrow_7 \mathbf{ab} * E \\
&\Rightarrow_6 \mathbf{ab} * (E) \\
&\Rightarrow_2 \mathbf{ab} * (E + E) \\
&\Rightarrow_1 \mathbf{ab} * (I + E) \\
&\Rightarrow_7 \mathbf{ab} * (\mathbf{a} + E) \\
&\Rightarrow_1 \mathbf{ab} * (\mathbf{a} + I) \\
&\Rightarrow_8 \mathbf{ab} * (\mathbf{a} + \mathbf{b})
\end{aligned}
$$

- The arrow "$\rightarrow$" is replaced by "::=".
- The non-terminal symbols are written between angle brackets (for example $\langle Exp \rangle$ and $\langle Ide \rangle$ could be the two non-terminal symbols of the grammar of Example 2.2).
- Productions with the same head are grouped into a single block using vertical bar ("|") to separate the productions. In Example 2.2, the productions for E could therefore be expressed as follows:

$$\langle E \rangle ::= \langle I \rangle \mid \langle E \rangle + \langle E \rangle \mid \langle E \rangle * \langle E \rangle \mid \langle E \rangle - \langle E \rangle \mid -\langle E \rangle \mid (\langle E \rangle).$$

Moreover, some notations mix symbols (for example, use of the vertical bar, the arrow and non-terminal symbols written in italic upper case).

Derivations and languages A grammar inductively defines a set of strings. We can make explicit the operative aspect of this inductive definition using the concept of derivation.

Example 2.3 We can ensure that the string $\mathbf{ab} * (\mathbf{a} + \mathbf{b})$ is correct according to the grammar of Example 2.2, reading the productions as "rewriting rules", and by repeatedly applying them. We use a bigger arrow ("$\Rightarrow$") to denote the operation of string rewriting. We can then proceed as follows. We start with the initial symbol, E, and rewrite it using a production (which we are allowed to select). Let us, for example, use production (3), and we obtain the rewriting $E \Rightarrow E * E$. Now we concentrate on the right-hand side of this production. We have two E's that can be rewritten (expanded) independently of each other. Let us take the one on the left and apply production (1) to obtain $E * E \Rightarrow I * E$. We can now choose whether we expand I or E (or the other way round). Figure 2.1 shows the rewriting that we have just started, developed until the string $\mathbf{ab} * (\mathbf{a} + \mathbf{b})$ has been derived. The production used is represented by each $\Rightarrow$s' subscript.

We can capitalise on this example in the following definition.

Definition 2.2 (Derivation) Having fixed a grammar, $G = (NT, T, R, S)$, and assigned two strings, v and w over $NT \cup T$, we say that w is immediately derived

Fig. 2.2 A derivation tree for the string $\mathbf{ab} * (\mathbf{a} + \mathbf{b})$

$$
\begin{aligned}
E &\Rightarrow_3 E * E \\
&\Rightarrow_6 E * (E) \\
&\Rightarrow_2 E * (E + E) \\
&\Rightarrow_1 E * (E + I) \\
&\Rightarrow_8 E * (E + \mathbf{b}) \\
&\Rightarrow_1 E * (I + \mathbf{b}) \\
&\Rightarrow_7 E * (\mathbf{a} + \mathbf{b}) \\
&\Rightarrow_1 I * (\mathbf{a} + \mathbf{b}) \\
&\Rightarrow_{10} I\mathbf{b} * (\mathbf{a} + \mathbf{b}) \\
&\Rightarrow_7 \mathbf{ab} * (\mathbf{a} + \mathbf{b})
\end{aligned}
$$

from v (or: v is rewritten in a single step into w), if w is obtained from v by substituting the body of a production of R whose head is V for a non-terminal symbol, V, in v. In this case, we will write $v \Rightarrow w$.

We say that w is derived from v (or: v is rewritten to w) and we write $v \Rightarrow^* w$, if there exists a finite (possibly empty) sequence of immediate derivations $v \Rightarrow w_0 \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w$.

Using the notation just introduced, and using the grammar for expressions that we have used so far we can write, for example $E * E \Rightarrow^* \mathbf{ab} * (\mathbf{a} + I)$. Particularly interesting are those derivations where on the left of the $\Rightarrow^*$ symbol there is the grammar's initial symbol and on the right is a string solely composed of *terminal* symbols. In a certain sense, these are maximal derivations which cannot be extended (by rewriting a non-terminal) either on the left or right. Following the intuition that has led us to the introduction of grammars, these derivations are the ones that give us the correct strings as far as the grammar is concerned.

Definition 2.3 (Generated Language) The language generated by a grammar $G = (NT, T, R, S)$ is the set $\mathscr{L}(G) = \{w \in T^* \mid S \Rightarrow^* w\}$.

Note that this definition, in accordance with everything we said at the start of Sect. 2.2.1, defines precisely a language over T^*.

Derivation Trees The derivation of a string is a sequential process. In it, there are steps that must be performed in a certain order. For example, in Fig. 2.1, it is clear that the first $\Rightarrow_{10}$ must follow the first $\Rightarrow_3$ because production (10) rewrites the non-terminal symbol I which does not exist in the initial string (which is composed of only the initial symbol E) which is introduced by production (3). But there are some steps whose order could be exchanged. In the derivation in Fig. 2.1, for example, each time that it is necessary to rewrite a non-terminal symbol, the leftmost is always chosen. We could imagine, on the other hand, concentrating first on the rightmost non-terminal symbol, thereby obtaining the derivation shown in Fig. 2.2.

The two derivations that we have given for the string $\mathbf{ab} * (\mathbf{a} + \mathbf{b})$ are, in a highly intuitive way, equivalent. Both reconstruct the structure of the string in the same way (in terms of non-terminal and terminal strings), while they differ only in the

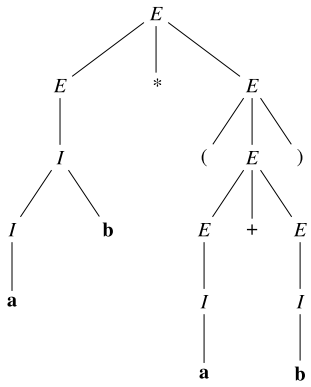


Fig. 2.3 A derivation tree for the string $\mathbf{ab} * (\mathbf{a} + \mathbf{b})$

order with which the productions are applied. This fact is made explicit in Fig. 2.3 which represents the derivation of the string $\mathbf{ab} * (\mathbf{a} + \mathbf{b})$ in the form of a tree.

Definition 2.4 Given a grammar, $G = (NT, T, R, S)$, a derivation tree (or parse tree) is an ordered tree in which:

1. Each node is labelled with a symbol in $NT \cup T \cup \{\epsilon\}$.
2. The root is labelled with S.
3. Each interior node is labelled with a symbol in NT.
4. If a certain node has the label $A \in NT$ and its children are $m_1, \ldots, m_k$ labelled respectively with $X_1, \ldots, X_k$, where $X_i \in NT \cup T$ for all $i \in [1, k]$, then $A \to X_1 \ldots X_k$ is a production of R.
5. If a node has label ϵ, then that node is the unique child. If A is its parent, $A \to \epsilon$ is a production in R.

It is easy to see that, a derivation tree can be obtained from any derivation. It is enough to start at the root (labelled with the initial symbol) and successively adding a level of children corresponding to the production used during the derivation. By applying this procedure to the derivation of Fig. 2.1, the derivation tree of Fig. 2.3 is obtained. Let us now apply this construction procedure to the derivation tree in Fig. 2.2. Again, the tree in Fig. 2.3 is obtained. The two derivations result in the same tree, and this fact corresponds to the intuition that the two derivations were substantially equivalent.

Derivation trees are one of the most important techniques in the analysis of programming language syntax. The structure of the derivation tree indeed expresses, by means of its subtrees, the logical structure that the grammar assigns to the string.

Trees

The concept of tree is of considerable importance in Computer Science and is also used a great deal in common language (think of a genealogical tree, for example). A tree is an information structure that can be defined in different ways and that exists in different "spaces". For our purposes, we are interested only in ordered, rooted trees (or simply *trees*) which we can define as follows. A (rooted, ordered) tree is a finite set of elements called *nodes*, such that if it is not empty, a particular node is called *the root* and the remaining nodes, if they exist, are partitioned between the elements of an (ordered) n-tuple $\langle S_1, S_2, \ldots, S_n \rangle$, $n \geq 0$, where each S_i, $i \in [1, N]$ is a tree.

Intuitively, therefore, a tree allows us to group nodes into levels where, at level 0, we have the root, at level 1 we have the roots of the trees $S_1, S_2, \ldots, S_n$ and so on.

Another (equivalent) definition of tree is often used. It is probably more significant for the reader who is familiar with genealogical trees, botanical classifications, taxonomies, etc. According to this definition, a tree is a particular case of a graph: a (rooted, ordered) tree is therefore a pair $T = (N, A)$, where N is a finite set of *nodes* and A is a set of ordered pairs of nodes, called *arcs*, such that:

- The number of arcs is equal to one less than the number of nodes.
- T is *connected*, that is, for each pair of nodes, $n, m \in N$, there exists a sequence of distinct nodes $n_0, n_1, \ldots, n_k$ such that $n_0 = n$ and $n_k = m$ and the pair (n_i, n_{i+1}) is an arc, for $i = 0, \ldots, k-1$.
- A (unique) node, r, is said to be the *root* and the nodes are ordered by level according to the following inductive definition. The root is at level 0; if a node n is at level i, and there exists the arc $(n, m) \in A$ then node m is at level $i + 1$.
- Given a node, n, the nodes m such that there exists an arc $(n, m) \in A$ are said to be the *children* of n (and n is said to be their *parent*); for every node $n \in N$, a total order is established on the set of all the children of n.

Using a curious mix of botanical, mathematical and "familiar" terms, nodes with the same parent are said to be *siblings* while nodes without children are said to be *leaves*. The root is the only node without a parent.

For example, the tree in Fig. 2.4 corresponds to the following derivation:

$$
\begin{aligned}
E &\Rightarrow_2 E + E \\
&\Rightarrow_3 E * E + E \\
&\Rightarrow_1 I * E + E \\
&\Rightarrow_7 \mathbf{a} * E + E \\
&\Rightarrow_1 \mathbf{a} * I + E \\
&\Rightarrow_8 \mathbf{a} * \mathbf{b} + E \\
&\Rightarrow_1 \mathbf{a} * \mathbf{b} + I \\
&\Rightarrow_7 \mathbf{a} * \mathbf{b} + \mathbf{a}
\end{aligned}
$$

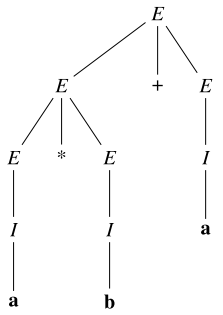


Fig. 2.4 A derivation tree for the string $\mathbf{a} * \mathbf{b} + \mathbf{a}$

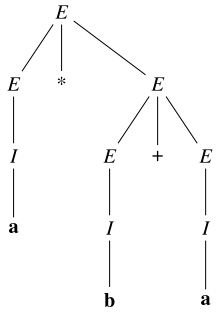


Fig. 2.5 Another derivation tree for the string $\mathbf{a} * \mathbf{b} + \mathbf{a}$

In the tree in Fig. 2.4, we can see that the subexpression $\mathbf{a} * \mathbf{b}$ appears as a left child of a sum. This expresses the fact that the string $\mathbf{a} * \mathbf{b} + \mathbf{a}$ must be interpreted as "first multiply **a** by **b** and add the result to **a**", given that in order to compute the sum, the operand present in the left subtree is needed.

Ambiguity Let us consider the following derivation (again using the grammar of Example 2.2):

$$\begin{aligned} E &\Rightarrow_3 E * E \\ &\Rightarrow_1 I * E \\ &\Rightarrow_7 \mathbf{a} * E \\ &\Rightarrow_2 \mathbf{a} * E + E \\ &\Rightarrow_1 \mathbf{a} * I + E \\ &\Rightarrow_8 \mathbf{a} * \mathbf{b} + E \\ &\Rightarrow_1 \mathbf{a} * \mathbf{b} + I \\ &\Rightarrow_7 \mathbf{a} * \mathbf{b} + \mathbf{a} \end{aligned}$$

If we construct the corresponding derivation tree, we obtain the tree in Fig. 2.5. Comparing the trees in Figs. 2.4 and 2.5, it can be seen that we have two different trees producing the same string. Reading the leaves of the two trees from left to right, in both cases, we obtain the string $\mathbf{a} * \mathbf{b} + \mathbf{a}$. Let us observe, however, that the two trees are radically different. The second assigns a structure to the same string that is quite different (and therefore reveals a different precedence implicit in the arithmetic operators). If we want to use derivation trees to describe the logical

structure of a string, we are in a bad position. The grammar of Example 2.2 is incapable of assigning a unique structure to the string in question. According to how the derivation is constructed, the precedence of the two arithmetic operators differs. Let us now see a formal definition of these concepts.

First, let us be clear about what it means to read or visit the leaves of a tree "from left to right". Visiting a tree consists of following a path through the tree's nodes so that each node is visited exactly once. There are different ways to visit nodes (depth-, breadth-first and symmetric), which, when applied to the same tree, produce different sequences of nodes. If only the leaves are considered, however, each and every one of these methods produces the same result for which we can use the following definitions:

Definition 2.5 Let $T = (N, A)$, be a non-empty, ordered tree with root r. The result of the left-to-right traversal of T is the sequence of nodes (leaves) obtained by the following recursive definition:

- If r has no children, the result of the traversal is r.
- If r has k children, $m_1, \ldots, m_k$, let $T_1, \ldots, T_k$ be the subtrees of T such that T_i has m_i as its root (T_i therefore contains m_i and all of that part of T which is underneath this node). The result of the traversal is the sequence of nodes obtained by visiting T_1 to T_k in turn, from left to right.

A this point, we can say what it means for a string to admit a derivation tree.

Definition 2.6 We say that a string of characters admits a derivation tree T if it is the result of a left-to-right traversal of T.

Finally, we can give the definition that interests us.

Definition 2.7 (Ambiguity) A grammar, G, is *ambiguous* if there exists at least one string of $\mathscr{L}(G)$ which admits more than one derivation tree.

We remark that ambiguity comes not from the existence of many derivations for the same string (a common and innocuous property) but from the fact that (at least) one string has more than one derivation *tree*.

An ambiguous grammar is useless as description of a programming language because it cannot be used to translate (compile) a program in a unique fashion. Fortunately, given an ambiguous grammar it is often possible to transform it into another, unambiguous, grammar that generates the same language.[5] Techniques for grammar disambiguation are outside the scope of this book. By way of example, Fig. 2.6 shows an unambiguous grammar whose generated language coincides with the one in Example 2.2.

[5]There exist pathological cases of languages which are generated only by ambiguous grammars. These languages have no relevance to programming languages.

Fig. 2.6 An unambiguous grammar for the language of expressions

$$G' = (\{E, T, A, I\}, \{\mathbf{a}, \mathbf{b}, +, *, -, (,)\}, R', E)$$

$$\begin{aligned} E &\rightarrow T \mid T + E \mid T - E \\ T &\rightarrow A \mid A * T \\ A &\rightarrow I \mid -A \mid (E) \\ I &\rightarrow \mathbf{a} \mid \mathbf{b} \mid I\mathbf{a} \mid I\mathbf{b} \end{aligned}$$

We have a grammar which interprets the structure of an expression according to the usual concept of arithmetic operator precedence. Unary minus ("-") has the highest precedence, followed by $*$, followed in their turn by $+$ and binary $-$ (which have the same precedence). The grammar interprets, then, a sequence of operators at the same level of precedence by association to the right. For example, the string $\mathbf{a} + \mathbf{b} + \mathbf{a}$ will be assigned the same structure as the string $\mathbf{a} + (\mathbf{b} + \mathbf{a})$. The absence of ambiguity is paid for by increased complexity. There are more non-terminals and the intuitive interpretation of the grammar is therefore more difficult.

The need to provide unambiguous grammars explains the contortions that appear in the definitions of some programming languages. For example, Fig. 2.7 shows an extract of the official grammar for Java's conditional command (its `if`) (the non-terminal symbols are printed in italic, while the terminal symbols in this extract are `if`, `else` and brackets).

This grammar is interesting for two reasons. First, note that what are formally considered single symbols (terminal or non-terminal) in context-free grammars are here represented by words composed of a number of characters. For example, `if` represents a single terminal symbol and, analogously, *IfThenElseStatement* represents a non-terminal symbol.

This happens because, in the definition of a programming language, it is preferential to use meaningful words (`if`, `then`, `else`) which can, up to certain limits, suggest an intuitive meaning, rather than symbols which would be harder to understand by the language user. In other words, as we will better see in the next chapters, that the use of (meaningful) symbolic names definitely makes programming easier. Analogously, for non-terminal symbols, they make the grammar easier to understand. It is definitely better to use names such as *Statement* rather than single symbols.

The second interesting aspect of the grammar is its complex nature. This complexity is necessary to resolve the ambiguity of strings (programs) such as the one exemplified by the following skeleton program:

```
if (expression1) if (expression2) command1;
   else command2;
```

Java, like a great many other programming languages allows conditional commands both with an `else` branch and without it. When an `if` command without `else` is combined with an `if` with an `else` branch (as in the case of the program appearing above), it is necessary to determine which of the two `if`s is the owner of the single `else`. The grammar in Fig. 2.7 is an unambiguous one which makes

Statement ::= ... | *IfThenStatement* | *IfThenElseStatement* |
StatementWithoutTrailingSubstatement
StatementWithoutTrailingSubstatement ::= ... | *Block* | *EmptyStatement* |
ReturnStatement
StatementNoShortIf ::= ... | *StatementWithoutTrailingSubstatement* |
IfThenElseStatementNoShortIf
IfThenStatement ::= `if` (*Expression*) *Statement*
IfThenElseStatement ::=
`if` (*Expression*) *StatementNoShortIf* `else` *Statement*
IfThenElseStatementNoShortIf ::=
`if` (*Expression*) *StatementNoShortIf* `else` *StatementNoShortIf*

Fig. 2.7 Grammar for Java conditional commands

"an `else` clause belong to the innermost `if` to which it might possibly belong" (from the Java definition [3]). In simple words, the `else` is paired with the second `if` (the one which tests `expression2`). Intuitively, this happens because, once an `if` command with an `else` is introduced by use of the non-terminal symbol *IfThenElseStatement*, the command in the *then* branch will be unable to hold an `if` command without an `else`, as can be seen by inspection of the rules which define the non-terminal symbol *StatementNoShortIf*.

2.3 Contextual Syntactic Constraints

The syntactic correctness of a phrase of a programming language sometimes depends on the *context* in which that phrase occurs. Let us consider, for example, the assignment `I = R+3;`. Assuming that the productions in the grammar that are used permit the correct derivation of this string, it might be, though, incorrect at the exact point in the program at which it occurs. For example, if the language requires the declaration of variables, it is necessary for programs to contain the declarations of `I` and `R` before the assignment. If, then, the language is typed, it would not accept the assignment if the type of the expression `R+3` is not compatible[6] with that of the variable `I`.

Strings that are correct with respect to the grammar, therefore, can be legal only *in a given context*. There are many examples of these contextual syntactic constraints:

- An identifier must be declared before use (Pascal, Java).
- The number of actual parameters to a function must be the same as the formal parameters (C, Pascal, Java, etc.).

[6] All these concepts will be introduced below. For now, an intuitive understanding of these concepts will suffice.

- In the case of assignment, the type of an expression must be compatible with that of the variable to which it is assigned (C, Pascal, Java, etc.).
- Assignments to the control variable of a `for` loop are not permitted (Pascal).
- Before using a variable, there must have been an assignment to it (Java).
- A method can be redefined (*overridden*) only by a method with the same signature (Java) or with a compatible signature (Java5).
- ...

These are, to be sure, syntactic constraints. However, their contextual nature makes it impossible to describe them using a context-free grammar (the name of this class of grammars was chosen purpose). A book on formal languages is required to go into detail on these issues; here, it suffices to note that there exist other types of grammar, called contextual grammars.[7] These grammars permit the description of cases such as this. These grammars are difficult to write, to process and, above all, there are no automatic techniques for efficient generation of translators, such as exist, on the other hand, for context-free grammars. This suggests, therefore, that use of grammars should be limited to the non-contextual description of syntax and then to express the additional contextual constraints using natural language or using formal techniques such as transition systems (which we will see in Sect. 2.5 when we consider semantics).

The term *static semantic constraint* is used in programming language jargon to denote the contextual constraints that appear in the syntactic aspects of a language. In the jargon, "syntax" in general means "describable using a context-free grammar", "static semantics" means "describable with verifiable contextual constraints on a static basis using the program text", while "dynamic semantics" (or simply "semantics") refers to everything that happens when a program is executed.

The distinction between context-free and context-dependent syntax (that is, static semantics) is established precisely by the expressive power of context-free grammars. The distinction between syntax and semantics is not always clear. Some cases are clear. Let us assume, for example, that we have a language whose definition establishes that should a division by 0 happen during execution, the abstract machine for the language *must* report it with an explicit error. This is clearly a dynamic semantic constraint. But let us assume, now, that the definition of another language is specified as follows:

> A program is syntactically incorrect when a division by 0 *can happen*.

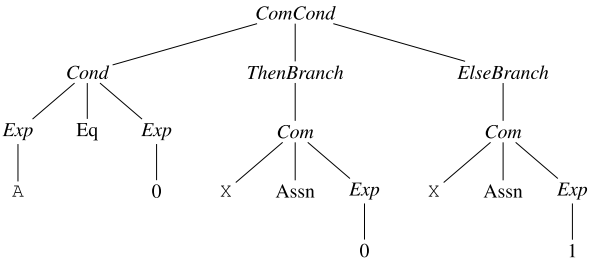
The program in Fig. 2.8 would be syntactically incorrect in this hypothetical language because there exists a sequence of executions (those in which the `read` command assigns the value 0 to `A`) which causes division by zero. Does the constraint that we have stated above (which certainly cannot be expressed in a free grammar) belong to static or dynamic semantics? It is clear that it refers to a dynamic event

[7] In extreme synthesis, in a contextual grammar, a production can take the form (which is more general than in a context-free grammar) $uAv \rightarrow uwv$, where u, v and w are strings over $T \cup NT$. Fundamental to this production, the non-terminal symbol A can be rewritten to w only if it appears in a certain context (the one described by u and v).

Fig. 2.8 A program that can cause a division by 0

```
int A, B;
read(A);
B = 10/A;
```

Fig. 2.9 An abstract syntax tree



(division by zero) but the constraint, to be meaningful (that is, that it really does exclude some programs) would have to be detected statically (as we have done for the simple program given above). A person can decide that it is syntactically incorrect without executing it but only from looking at the code and reasoning about it). More important than its classification, is understanding whether we have a verifiable constraint or not. Otherwise stated, is it really possible to implement an abstract machine for such a language, or does the very nature of the constraint imply that there can exist no abstract machine which is able to check it in a static fashion for an arbitrary program?

These are questions that we cannot immediately answer, even though they are very important, because they concern what we can, and cannot, do with an abstract machine. We will concern ourselves with these questions in the next chapter (Chap. 3).

2.4 Compilers

The moment has arrived at which to see in outline how the syntactic description of a language can be used automatically to translate a program. We know from Chap. 1 that such an automatic translator is called a *compiler*, whose general logical structure is shown in Fig. 2.10. It is characterised by a cascaded series of phases. The various phases, starting with a string representing the program in the *source* language, generate various internal intermediate representations until a string in the *object* language is generated. Note that, as seen in the previous chapter, in this context, "object language" does not necessarily equate to "machine code" or to "low-level language". It is merely a language towards which the translation is directed. In what follows, we briefly describe the different phases of compilation. We do not pretend to explain how to construct a compiler (see [1]), but just to fix some ideas useful for our study of programming languages.

Lexical analysis The aim of lexical analysis is to read the symbols (characters) forming the program sequentially from the input and to group these symbols into

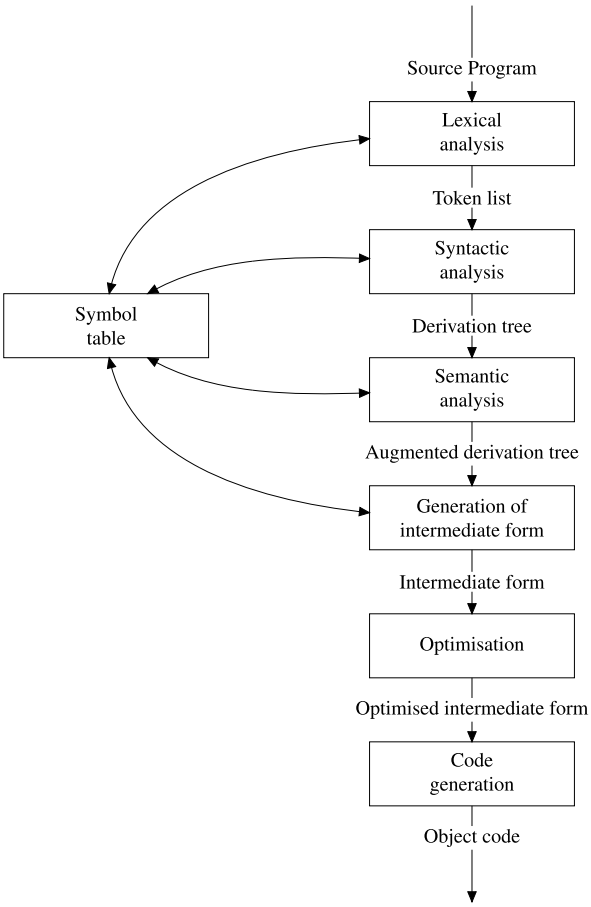


Fig. 2.10 Organisation of a compiler

meaningful logical units, which we call *tokens*, (which are analogous, for our programming language, to the words in the dictionary of a natural language). For example, the lexical analyser of C or Java, when presented with the string `x = 1 + foo++;` will produce 7 tokens: the identifier `x`, the assignment operator `=`, the number `1`, the addition operator `+`, the identifier `foo`, the auto increment operator `++` and finally the command termination token `;`. Particularly important tokens are the reserved words of the language (such as `for`, `if`, `else`, etc.), operators, open and close brackets (such as `{` and `}` in C or Java, but also those such as `begin` and `end` in Pascal). Lexical analysis (or *scanning*) is a very simple operation which scans the input text of the source program from left to right, taking a single pass to recognise tokens. No check is yet made on the sequence of tokens such as, for example, checking that brackets are correctly balanced. As discussed in detail in the box, the technical tool used for lexical analysis is a particular class of generative grammars (regular grammars). The use of a grammar to describe the elements of the lexicon is due both to the necessity of efficiently recognising these elements, and to

Abstract and Concrete Syntax

The grammar of a programming language defines the language as a set of strings. This set of strings corresponds in a natural way to the set of their derivation trees. These trees are much more interesting than the strings. First, they abstract from the specific lexical characteristic of the tokens. It can also happen that lexically different structures can result in the same tree. The tree depicted in Fig. 2.9 could correspond to the Pascal string:

```
if A=0 then X:=0 else X:=1
```

or to the Java string:

```
if (A==0) X=0; else X=1;
```

As we have already repeatedly observed, derivation trees are interesting because they express the canonical structure that can be assigned to the string.

Not all derivation trees correspond to legal programs. We know that static semantic analysis has the task of selecting those trees satisfying the contextual constraints of the language. The set of trees resulting from this process constitutes the *abstract syntax* of language. This is a much more convenient way of thinking of a language if we are interested in its manipulation (and not merely in writing correct programs in it).

the fact that, unlike the case of natural language lexicons, lexicons for programming languages can be formed from an infinite set of words and therefore a simple list, so the one used in a normal dictionary is inadequate (think, for example, of the possibility of defining identifiers as sequences of characters of arbitrary length starting with a particular letter).

Syntactic analysis Once the list of tokens has been constructed, the syntactic analyser (or *parser*) seeks to construct a derivation tree for this list. This is, clearly, a derivation tree in the grammar of the language. Each leaf of this tree must correspond to a token from the list obtained by the scanner. Moreover, these leaves, read from left to right, must form a correct phrase (or a sequence of terminal symbols) in the language. We already know that such trees represent the logical structure of the program which will be employed by the successive phases of compilation.

It can happen that the parser is unable to construct a derivation tree. This happens when the input string is not correct with reference to the language's grammar. In such a case, the parser reports the errors it has encountered and compilation is aborted. In reality, lexical and syntactic analysis are interleaved in a more direct fashion than appears from these notes (in particular, the two phases are not sequential but the scanner produces a token for each invocation of the parser); more details can be obtained from one of the many texts on compilers.

Regular Grammars

The difference between lexical and syntactic analysis can be made precise using a formal classification of grammars. If a grammar has all productions of the form $A \to bB$ (or in the form $A \to Bb$), where A and B are non-terminal symbols (B can also be absent or coincide with A) and b is a single terminal symbol, the grammar is said to be *regular*. In Example 2.2, the (sub-) grammar based on the non-terminal symbol I is a regular grammar (while the subgrammar for E is not).

The expressive power of regular grammars is highly limited. In particular, using a regular grammar, it is not possible to "count" an arbitrary number of characters. As a consequence, it is not possible to express the balancing of syntactic structures such as brackets using a regular grammar.

Technically, lexical analysis is the first phase of translation which checks that the input string can be decomposed into tokens, each of which is described by a regular grammar (in Example 2.2, lexical analysis would have recognised the sequences of **a** and **b** as instances of the non-terminal symbol I).

Once the sequence of tokens has been obtained, syntactic analysis takes place using a properly context-free grammar which uses the tokens produced by the preceding lexical analysis as its terminal symbols.

Semantic analysis The derivation tree (which represents the syntactic correctness of the input string) is subjected to checks of the language's various context-based constraints. As we have seen, it is at this stage that declarations, types, number of function parameters, etc., are processed. As these checks are performed, the derivation tree is *augmented* with the information derived from them and new structures are generated. By way of an example, every token that corresponds to a variable identifier will be associated with its type, the place of declaration and other useful information (for example its *scope*, see Chap. 4). To avoid the useless duplication of this information, it is, in general, collected into structures external to the derivation tree. Among these structures, the one in which information about identifiers is collected is called the symbol table. The symbol table plays an essential role in successive phases.

At the cost of boring the reader, so that we do not confuse it with what we call semantics in Sect. 2.5, let us note that the term semantic analysis is an historical relic and that it is concerned with context-based syntactic constraints.

Generation of intermediate forms An appropriate traversal of the augmented derivation tree allows an initial phase of code generation. It is not yet possible to generate code in the object language, given that there are many optimisations left to do and they are independent of the specific object language. Moreover, a compiler is often implemented to generate code for a whole series of object languages (for example, machine code for different architectures), not just one. It is useful, therefore, to concentrate all choices relating to a specific language in a single phase and

to generate code to an intermediate form, which is designed to be independent of both the source and the object languages.

Code optimisation The code obtained from the preceding phases by repeatedly traversing the derivation tree is fairly inefficient. There are many optimisations that can be made before generating object code. Typical operations that can be performed are:

- Removal of useless code (*dead code removal*). That is, removal of pieces of code that can never be executed because there is no execution sequence that can reach them.
- *In-line expansion* of function calls. Some function (procedure) calls can be substituted by the body of the associated function, making execution faster. It also makes other optimisations possible.
- Subexpression factorisation. Some programs compute the same value more than once. If, and when, this fact is discovered by the compiler, the value of the common subexpression can be calculated once only and then stored.
- Loop optimisations. Iterations are the places where the biggest optimisations can be performed. Among these, the most common consists of removing from inside a loop the computation of subexpressions whose value remains constant during different iterations.

Code generation Starting with optimised intermediate form, the final object code is generated. There follows, in general, a last phase of optimisation which depends upon the specific characteristics of the object language. In a compiler that generates machine code, an important part of this last phase is register assignment (decisions as to which variables should be stored in which processor registers). This is a choice of enormous importance for the efficiency of the final program.

2.5 Semantics

As is intuitively to be expected, the description of the semantics of a programming language is somewhat more complex than its syntactic description. This complexity is due both to technical problems, as well to the need to mediate between two opposing issues: the need for exactness as well as for flexibility. As far as exactness is concerned, a precise and unambiguous description is required of what must be expected from every syntactically correct construct so that every user knows *a priori* (that is before program execution), and in a manner independent of the architecture, what will happen at runtime. This search for exactness, however, must not preclude different implementations of the same language, all of which are correct with respect to the semantics. The semantic specification must therefore also be flexible, that is, it must not anticipate choices that are to be made when the language is implemented (and which therefore do not play a part in the definition of the language itself).

It would not be difficult to achieve exactness at the expense of flexibility. It is enough to give the semantics of a language using a specific compiler on a specific

architecture. The official meaning of a program is given by its execution on this architecture after translation using that compiler. Apart form the difficulty of speaking in general about the semantics of the language's constructs, this solution has no flexibility. Although we have constructed what we claim to be a semantics, there exists only one implementation (the official one) and all the other implementations are completely equivalent. But to what level of detail is the canonical implementation normative? Is the computation time of a particular program part of its definition? Is the reporting of errors? How can typically architecture-dependent input/output commands be ported to a different architecture? When the implementation technology of the physical machine changes, how is the semantics, which we have incautiously defined in terms of a specific machine, affected?

One of the difficulties of semantic definition is really that of finding the happy medium between exactness and flexibility, in such a way as to remove ambiguity, still leaving room for implementation. This situation suggests using formal methods to describe the semantics. Methods of this kind have existed for a long time for the artificial languages of mathematical logic and they have been adapted by computer scientists for their own ends. Yet, some semantic phenomena make formal description complex and not easily usable by anyone who does not have the appropriate skills. It is for this reason that the majority of official programming language definitions use natural language for their semantic description. This does not detract from the fact that formal methods for semantics are very often used in the preparatory phases of the design of a programming language, or to describe some of its particular characteristics, where it is paramount to avoid ambiguity at the cost of simplicity.

Formal methods for semantics divide into two main families: *denotational* and *operational* semantics.[8] Denotational semantics is the application to programming languages of techniques developed for the semantics of logico-mathematical languages. The meaning of a program is given by a function which expresses the input/output behaviour of the program itself. The domain and codomain of this function are suitable mathematical structures, such as the environment and the memory (store), that are internal to the language. Refined mathematical techniques (continuity, fixed points, etc.) allow the appropriate treatment of phenomena such as iteration and recursion (see [8]).

In the operational approach, on the other hand, there are no external entities (for example, functions) associated with language constructs. Using appropriate methods, an operational semantics specifies the behaviour of the abstract machine. That is, it formally defines the interpreter, making reference to an abstract formalism at a much lower level. The various operational techniques differ (sometimes profoundly) in their choice of formalism; some semantics use formal automata, others use systems of logico-mathematical rules, yet others prefer transition systems to specify the state transformations induced by a program.

[8] For completeness, we should also talk about algebraic and axiomatic semantics, but simplicity and conciseness require us to ignore them.

$$
\begin{array}{lll}
\mathit{Num} & ::= & 1 \mid 2 \mid 3 \mid \ldots \\
\mathit{Var} & ::= & X_1 \mid X_2 \mid X_3 \mid \ldots \\
 & & \\
\mathit{AExp} & ::= & \mathit{Num} \mid \mathit{Var} \mid (\mathit{AExp} + \mathit{AExp}) \mid (\mathit{AExp} - \mathit{AExp}) \\
\mathit{BExp} & ::= & \mathbf{tt} \mid \mathbf{ff} \mid (\mathit{AExp} == \mathit{AExp}) \mid \neg \mathit{BExp} \mid (\mathit{BExp} \wedge \mathit{BExp}) \\
 & & \\
\mathit{Com} & ::= & \mathbf{skip} \mid \mathit{Var} := \mathit{AExp} \mid \mathit{Com}; \mathit{Com} \mid \\
 & & \quad \mathbf{if}\ \mathit{BExp}\ \mathbf{then}\ \mathit{Com}\ \mathbf{else}\ \mathit{Com} \mid\ \mathbf{while}\ \mathit{BExp}\ \mathbf{do}\ \mathit{Com}
\end{array}
$$

Fig. 2.11 Syntax of a simple imperative language

This is not a book in which an exhaustive treatment of these semantic techniques can be given. We content ourselves with a simple example of techniques based on transition systems to give a semantics for a rudimentary imperative language. This technique, called SOS (Structured Operational Semantics, [7]), is highly versatile and has been effectively used in giving the semantics to some languages including Standard ML (an influential functional language).

Figure 2.11 shows the grammar of a simple language. For reasons of readability (and so that we can obtain a concise semantics), we have a grammar with infinite productions for the *Num* and *Var* non-terminal symbols. The explicit presence of brackets around each binary operator application eliminates possible problems of ambiguity.

We need to introduce some notation. We will use n to denote an arbitrary *Num* (numeric constant); using X, we will denote an arbitrary *Var* (variable). We write a for an arbitrary *AExp* (arithmetic expression). We use b to denote an arbitrary *BExp* (boolean expression, where **tt** and **ff** denote the values *true* and *false*, respectively). We write c for arbitrary *Comm* (commands). We use, when needed, subscripts to distinguish between objects of the same syntactic category (for example, we will write a_1, a_2, etc., for *AExp*s).

State The semantics of a command in our language uses a simple memory model which stores values of *Var*. In this model, a *state* is a finite sequence of pairs of the form (X, n) which we can read as "in the current state, the variable X has the value n" (in our little language, the value will always be an integer but we can easily imagine more complex situations). Given a command c (that is a derivation tree that is correct according to the grammar in Fig. 2.11), its reference state is given by a sequence of pairs which includes all *Var*s which are named in c. We denote an arbitrary state by σ or τ, with or without subscripts.

We need to define some operations on states: modification of an existing state and the retrieval of a variable's value in the current state. To this end, given a state, σ, a *Var* X and a value v, we write $\sigma[X \leftarrow v]$ to denote a new state that is the same as σ but differs from it by associating X with the value v (and thereby losing any previous association to X). Given a state, σ, and a variable, X, we write $\sigma(X)$ for the value that σ associates with X; this value is undefined if X does not appear in the domain of σ (σ is therefore a partial function).

Example 2.4 Let us fix $\sigma = [(X, 3), (Y, 5)]$, we have $\sigma[X \leftarrow 7] = [(X, 7), (Y, 5)]$. We also have $\sigma(Y) = 5$ and $\sigma[X \leftarrow 7](X) = 7$; $\sigma(W)$ is undefined.

Transitions Structured operational semantics can be seen as an elegant way of defining the functioning of an abstract machine without going into any details about its implementation.[9] This functioning is expressed in terms of the elementary computational steps of the abstract machine. The formal way in which structured operational semantics defines the meaning of a program, c, is in terms of a *transition* which expresses a step in the transformation (of the state and/or of the program itself) induced by the execution of the command. The simplest form of transition is:

$$\langle c, \sigma \rangle \rightarrow \tau,$$

where c is a command, σ the *starting state* and τ the *terminal state*. The interpretation that we give to this notation is that if we start the execution of c in the state σ, the execution terminates (in a single step) with state τ. For example, the transition which defines the **skip** command is

$$\langle \textbf{skip}, \sigma \rangle \rightarrow \sigma.$$

It can be seen that we have a command that does nothing. Starting in any state, σ, **skip** terminates leaving the state unchanged.

In more complex cases, a *terminal situation* will be reached not in a single large step, but rather in many little steps which progressively transform the state (starting with σ); the command, c, is progressively executed a part at a time until the whole has been "consumed". These little steps are expressed by transitions of the form:

$$\langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle.$$

For example, one of the transitions which define the conditional command will be:

$$\langle \textbf{if tt then } c_1 \textbf{ else } c_2, \sigma \rangle \rightarrow \langle c_1, \sigma \rangle,$$

This means that if the boolean condition is true, the command in the **then** branch must be executed. Some transitions, finally, are *conditional*: if some command, c_1, has a transition, then the command c has another transition. Conditional transitions take the form of a *rule*, expressed as a fraction:

$$\frac{\langle c_1, \sigma_1 \rangle \rightarrow \langle c_1', \sigma_1' \rangle \qquad \langle c_2, \sigma_2 \rangle \rightarrow \langle c_2', \sigma_2' \rangle}{\langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle}.$$

We read this rule in the following way. If the command, c_1, starting in state σ_1, can perform a computational step that transforms itself into command c_1' in state

[9] Using terminology which we will use again in the final chapters of this book, we can say that it is a *declarative* description of the language's abstract machine.

$$\langle X, \sigma \rangle \to \langle \sigma(X), \sigma \rangle$$

$$\frac{}{\langle (n+m), \sigma \rangle \to \langle p, \sigma \rangle} \text{ where } p = n + m \qquad \frac{}{\langle (n-m), \sigma \rangle \to \langle p, \sigma \rangle} \text{ where } p = n - m \text{ e } n \geq m$$

$$\frac{\langle a_1, \sigma \rangle \to \langle a', \sigma \rangle}{\langle (a_1 + a_2), \sigma \rangle \to \langle (a' + a_2), \sigma \rangle} \qquad \frac{\langle a_2, \sigma \rangle \to \langle a'', \sigma \rangle}{\langle (a_1 + a_2), \sigma \rangle \to \langle (a_1 + a''), \sigma \rangle}$$

$$\frac{\langle a_1, \sigma \rangle \to \langle a', \sigma \rangle}{\langle (a_1 - a_2), \sigma \rangle \to \langle (a' - a_2), \sigma \rangle} \qquad \frac{\langle a_2, \sigma \rangle \to \langle a'', \sigma \rangle}{\langle (a_1 - a_2), \sigma \rangle \to \langle (a_1 - a''), \sigma \rangle}$$

Fig. 2.12 Semantics of arithmetic expressions

σ_1', and if the command c_2, starting in σ_2, can perform a computational step and transform itself into the command c_2' in state σ_2', then the command, c, starting in the state σ can perform a computational step and transform itself into the command c' in state σ'. It is clear that a specific rule will express a number of meaningful relationships between c, c_1 and c_2 (and their respective states). In general, c_1 and c_2 will be subcommands of c.[10]

Expression semantics Figure 2.12 shows the rules for the semantics of arithmetic expressions. The first three rules are terminal rules (that is, the computation to the right of the arrow is in a form to which no more transitions can be applied). In order of presentation, they define the semantics of a variable, of addition in the case in which both of the summands are constants, of subtraction in the case in which both its arguments are constants and its result is a natural number. Note that no semantics is given for expressions such as $5 - 7$. The second group of four rules is formed of conditional rules. The first pair defines the semantics of sums and expresses the fact that to calculate the value of a sum, it is necessary first to evaluate its two arguments separately. Note how the semantics specifies *no order of evaluation* for the arguments of an addition. Subtraction is handled in a similar fashion.

Figure 2.13 shows the semantics of logical expressions and adopts the same ideas as for arithmetic expressions. Note that in this figure, *bv* denotes a boolean value (**tt** or **ff**).

Command semantics It can be seen how the state, σ, always remains unaltered during the evaluation of an expression and how it is only used to obtain the value of a variable. This changes for the semantics of commands as shown in Fig. 2.14. Note how the rules in the Figure are essentially of two types: those which, for every command, express the actual computational step for that command (this holds for rules ($c1$), ($c2$), ($c4$), ($c6$), ($c7$) and ($c9$)), as well as those which serve just to make

[10]It will not have escaped the attentive reader that what we have called conditional rules are, in reality, inductive rules forming part of the definition of transition relations.

$$\langle (n == m), \sigma \rangle \to \langle \mathbf{tt}, \sigma \rangle \quad \text{if } n = m \qquad \langle (n == m), \sigma \rangle \to \langle \mathbf{ff}, \sigma \rangle \quad \text{if } n \neq m$$

$$\langle (bv_1 \wedge bv_2), \sigma \rangle \to \langle bv, \sigma \rangle \quad \text{where } bv \text{ is the } \textit{and} \text{ of } bv_1 \text{ and } bv_2$$

$$\langle \neg \mathbf{tt}, \sigma \rangle \to \langle \mathbf{ff}, \sigma \rangle \qquad \langle \neg \mathbf{ff}, \sigma \rangle \to \langle \mathbf{tt}, \sigma \rangle$$

$$\frac{\langle a_1, \sigma \rangle \to \langle a', \sigma \rangle}{\langle (a_1 == a_2), \sigma \rangle \to \langle (a' == a_2), \sigma \rangle} \qquad \frac{\langle a_2, \sigma \rangle \to \langle a'', \sigma \rangle}{\langle (a_1 == a_2), \sigma \rangle \to \langle (a_1 == a''), \sigma \rangle}$$

$$\frac{\langle b_1, \sigma \rangle \to \langle b', \sigma \rangle}{\langle (b_1 \wedge b_2), \sigma \rangle \to \langle (b' \wedge b_2), \sigma \rangle} \qquad \frac{\langle b_2, \sigma \rangle \to \langle b'', \sigma \rangle}{\langle (b_1 \wedge b_2), \sigma \rangle \to \langle (b_1 \wedge b''), \sigma \rangle}$$

$$\frac{\langle b, \sigma \rangle \to \langle b', \sigma \rangle}{\langle \neg b, \sigma \rangle \to \langle \neg b', \sigma \rangle}$$

Fig. 2.13 Semantics of boolean expressions

$$\langle \mathbf{skip}, \sigma \rangle \to \sigma \quad (c1)$$

$$\langle X := n, \sigma \rangle \to \sigma[X \leftarrow n] \quad (c2) \qquad \frac{\langle a, \sigma \rangle \to \langle a', \sigma \rangle}{\langle X := a, \sigma \rangle \to \langle X := a', \sigma \rangle} \quad (c3)$$

$$\frac{\langle c_1, \sigma \rangle \to \sigma'}{\langle c_1; c_2, \sigma \rangle \to \langle c_2, \sigma' \rangle} \quad (c4) \qquad \frac{\langle c_1, \sigma \rangle \to \langle c_1', \sigma' \rangle}{\langle c_1; c_2, \sigma \rangle \to \langle c_1'; c_2, \sigma' \rangle} \quad (c5)$$

$$\langle \mathbf{if\ tt\ then}\ c_1\ \mathbf{else}\ c_2, \sigma \rangle \to \langle c_1, \sigma \rangle \quad (c6)$$
$$\langle \mathbf{if\ ff\ then}\ c_1\ \mathbf{else}\ c_2, \sigma \rangle \to \langle c_2, \sigma \rangle \quad (c7)$$

$$\frac{\langle b, \sigma \rangle \to \langle b', \sigma \rangle}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma \rangle \to \langle \mathbf{if}\ b'\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma \rangle} \quad (c8)$$

$$\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \langle \mathbf{if}\ b\ \mathbf{then}\ c; \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{else\ skip}, \sigma \rangle \quad (c9)$$

Fig. 2.14 Semantics of commands

the computation of a subcommand (or of a subexpression) progress. The semantic description of our language is now complete.

Computations A *computation* is a sequence of transitions which cannot be extended by another transition. Moreover, each transformation in a computation must be described by some rule.

Example 2.5 Let us consider the following program, c:

$$X := 1;\ \textbf{while}\ \neg(X == 0)\ \textbf{do}\ X := (X - 1)$$

Let us fix a state which includes all the variables mentioned in the program, for example, $\sigma = [(X, 6)]$. We can calculate the computation of c in σ as follows. To abbreviate the notation, we write c' to denote the iterative command **while ¬(X == 0) do** $X := (X - 1)$. It is not difficult to see that the computation generated by c is the following:

$$\begin{aligned}
&\langle c, \sigma\rangle\\
&\rightarrow \langle c', \sigma[X \leftarrow 1]\rangle\\
&\rightarrow \langle \textbf{if}\ \neg(X == 0)\ \textbf{then}\ X := (X - 1); c'\ \textbf{else skip}, \sigma[X \leftarrow 1]\rangle\\
&\rightarrow \langle \textbf{if}\ \neg(1 == 0)\ \textbf{then}\ X := (X - 1); c'\ \textbf{else skip}, \sigma[X \leftarrow 1]\rangle\\
&\rightarrow \langle \textbf{if}\ \neg\textbf{ff then}\ X := (X - 1); c'\ \textbf{else skip}, \sigma[X \leftarrow 1]\rangle\\
&\rightarrow \langle \textbf{if tt then}\ X := (X - 1); c'\ \textbf{else skip}, \sigma[X \leftarrow 1]\rangle\\
&\rightarrow \langle X := (X - 1); c', \sigma[X \leftarrow 1]\rangle\\
&\rightarrow \langle X := (1 - 1); c', \sigma[X \leftarrow 1]\rangle\\
&\rightarrow \langle X := 0; c', \sigma[X \leftarrow 1]\rangle\\
&\rightarrow \langle c', \sigma[X \leftarrow 0]\rangle\\
&\rightarrow \langle \textbf{if}\ \neg(X == 0)\ \textbf{then}\ X := (X - 1); c'\ \textbf{else skip}, \sigma[X \leftarrow 0]\rangle\\
&\rightarrow \langle \textbf{if}\ \neg(0 == 0)\ \textbf{then}\ X := (X - 1); c'\ \textbf{else skip}, \sigma[X \leftarrow 0]\rangle\\
&\rightarrow \langle \textbf{if}\ \neg\textbf{tt then}\ X := (X - 1); c'\ \textbf{else skip}, \sigma[X \leftarrow 0]\rangle\\
&\rightarrow \langle \textbf{if ff then}\ X := (X - 1); c'\ \textbf{else skip}, \sigma[X \leftarrow 0]\rangle\\
&\rightarrow \langle \textbf{skip}, \sigma[X \leftarrow 0]\rangle\\
&\rightarrow \sigma[X \leftarrow 0]
\end{aligned}$$

The computation in the example just discussed is a *terminated* computation, in the sense that, after a certain number of transitions, a situation is reached in which no other transition is possible. It can be seen that, moreover, the definition of computation that we have given above does *not* require that a computation be finite but only that *it cannot be extended*. There is, therefore, the possibility that there are infinite computations, as the following example demonstrates:

Example 2.6 Consider the following program, d:

$$X := 1;\ \textbf{while}\ (X == 1)\ \textbf{do skip}$$

Assume we are in the state $\tau = [(X, 0)]$. Let d' be the command **while** $(X == 1)$ **do skip**. We have:

$$\begin{aligned}
&\langle d, \tau\rangle\\
&\rightarrow \langle d', \tau[X \leftarrow 1]\rangle\\
&\rightarrow \langle \textbf{if}\ (X == 1)\ \textbf{then skip}\ ; d'\ \textbf{else skip}, \tau[X \leftarrow 1]\rangle\\
&\rightarrow \langle \textbf{if}\ (1 == 1)\ \textbf{then skip}\ ; d'\ \textbf{else skip}, \tau[X \leftarrow 1]\rangle
\end{aligned}$$

$$\rightarrow \langle \textbf{if tt then skip} \,;\, d' \,\textbf{else skip}, \tau[X \leftarrow 1]\rangle$$
$$\rightarrow \langle \,\textbf{skip} \,;\, d', \tau[X \leftarrow 1]\rangle$$
$$\rightarrow \langle d', \tau[X \leftarrow 1]\rangle$$
$$\rightarrow \ldots$$

There exist, therefore, two fundamentally different types of computations: *finite* ones (also called *terminating*) and *divergent* ones (that is, infinite ones which correspond, intuitively, to looping programs).

2.6 Pragmatics

If a precise description of the syntax and semantics of a programming language is (and must be) given, the same is not true of the pragmatics of the language. Let us recall that, for our purposes, the pragmatics of a language answers the question "what is the purpose of this construct?" or "what use is a certain command?" It is clear, therefore, that the pragmatics of a programming language is not established once and for all during its definition (that is, the definition of its syntax and semantics). On the contrary, it evolves with the use of the language. All suggestions about programming style are part of pragmatics. For example, there is the principle that jumps (`gotos`) should be avoided at all possible times. The choice of the most appropriate mode for passing parameters to a function is also a pragmatic question, as is the choice between bounded and unbounded iteration.

In a sense, the pragmatics of a programming language coincides with software engineering (the discipline which studies methods for the design and production of software) and is not much studied there. For many other aspects, on the other hand, clarifying the purpose and use of constructs is an essential part of the study of a programming language. It is for this that we will often refer below to pragmatics, possibly without making it explicit that we are so doing.

2.7 Implementation

The final level of programming language description will not be considered in this book in details. As we amply saw in the previous chapter, implementing a language means writing a compiler for it, as well as implementing an abstract machine for the compiler's object language; or to write an interpreter and implement the abstract machine in the language in which the interpreter is written. Alternatively, as happens in practice, a mix of both techniques is employed. Once more, this is not a book in which we can be exhaustive about these matters. But it would not be correct to present the constructions of programming languages without some reference to their implementation cost. Even without specifying the set of constructions in an interpreter, for each construct, we should always ask: "How is it to be implemented?", "At what cost?" The answer to these questions will also help us better to understand

the linguistic aspect (because a certain construct is formed in a certain way), as well as the pragmatic one (how can a certain construct best be used).

2.8 Chapter Summary

The chapter introduced the fundamental techniques for the description and implementation of a programming language.

- The distinction between syntax, semantics, pragmatics and implementation. Each of these disciplines describes a crucial aspect of the language.
- Context-free grammars. A formal method essential for the definition of a language's syntax.
- Derivation trees and ambiguity. Derivation trees represent the logical structure of a string. The existence of a unique tree for every string permits a canonical interpretation to be assigned to it.
- Static semantics. Not all syntactic aspects of a language are describable using grammars: Static semantic checks are concerned with eliminating from legal programs those strings which, though correct as far as the grammar is concerned, do not respect additional contextual constraints.
- The organisation of a compiler.
- Structured operational semantics. A formal method for describing the semantics of programming languages based on transition systems.

2.9 Bibliographical Notes

The literature on the topics of this chapter is enormous, even considering only introductory material. We limit ourselves to citing [4], the latest edition of a classic text on formal languages used by generations of students. For compiler-construction methods, we refer the reader to [1] and [2]. An introduction to operational semantics is [5], which also deals with denotational semantics; at a more advanced level, see [8], which also deals with denotational semantics.

2.10 Exercises

1. Consider the grammar G'', obtained from that in Fig. 2.6 by substituting the productions for the non-terminal symbol T with the following:

$$T \to A \mid E * T.$$

   Show that G'' is ambiguous.
2. Give the obvious ambiguous grammar for the conditional command `if`, `then`, `else`.

3. Using the grammar fragment in Fig. 2.7 as a reference point, construct a derivation tree for the string

```
if (expression1) if (expression2) command1
   else command2
```

   Assume that the following derivations exist:
   Expression $\Rightarrow^*$ `expression1`,
   Expression $\Rightarrow^*$ `expression2`,
   StatementWithoutTrailingSubstatement $\Rightarrow^*$ `command`,
   StatementWithoutTrailingSubstatement $\Rightarrow^*$ `command`.

4. Define a grammar that will generate all the pairs of balanced curly brackets (braces).
5. Define a grammar that generates the language $\{a^n b^m \mid n, m \geq 1\}$ using only productions of the form $N \rightarrow t\,M$ or $N \rightarrow t$, where N and M are any non-terminal symbol and t is any terminal symbol. Try to give an intuitive explanation of why there exists no grammar with these characteristics which generates the language $\{a^n b^n \mid n \geq 1\}$.
6. Modify the rule of Fig. 2.12 so as to describe a right-to-left evaluation of arithmetic expressions.
7. Calculate the computation of the command

$$X := 1;\ \textbf{while}\ \neg(X == 3)\ \textbf{do}\ X := (X + 1)$$

   starting with a chosen state.
8. In Example 2.5, the last transition has as its right member only a state (and not a pair composed of a command and a state). Is this always the case in a finite computation? (Hint: consider the command $X := 0;\ X := (X - 1)$, starting with any state whatsoever which includes X.)
9. State what the computation corresponding to the following command is:

$$X := 1; \quad X := (X - 1); \quad X := (X - 1); \quad X := 5$$

10. Considering Exercises 8 and 9, state a criterion which allows the division of finite computations into those which are correct and those which terminate because of an error.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, 1988.
2. A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, Cambridge, 1998.
3. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 3/E*. Addison Wesley, Reading, 2005. Available online at http://java.sun.com/docs/books/jls/index.html.

4. J. E. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, 2001.
5. C. Laneve. *La Descrizione Operazionale dei Linguaggi di Programmazione*. Franco Angeli, Milano, 1998 (in Italian).
6. C. W. Morris. Foundations of the theory of signs. In *Writings on the Theory of Signs*, pages 17–74. Mouton, The Hague, 1938.
7. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
8. G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, 1993.