# 2 Elements of Programming Languages

## 2.1 Syntax

Like ordinary languages, programming languages have syntax:

> The *syntax* of a (programming) language is a set of rules that define what sequences of symbols are considered to be valid expressions (programs) in the language.

The syntax (set of rules) is expressed using a formal notation.

The most widespread formal notation for syntax is *Extended Backus-Naur Form* (*EBNF*) In EBNF, we start with a highest level entity *program* and use rules to decompose entities until single characters are reached. For example, in C the syntax of an if-statement is given by the rule:

*if-statement* ::= **if (** *expression* **)** *statement* [ **else** *statement* ]

Names in italic represent syntactic categories, while names and symbols in boldface represent actual characters that must appear in the program. Each rule contains the symbol "::=" which means "is composed of". Various other symbols are used to make the rules more concise:

| | | | | | |
|---|---|---|---|---|---|
| [ ] | Optional | { } | Zero or more repetitions | \| | Or |

Thus the else-clause of an if-statement is optional. The use of braces is demonstrated by the (simplified) rule for the declaration of a list of variables:

*variable-declaration* ::= *type-specifier identifier* {**,** *identifier*}**;**

This is read: a variable declaration *is composed of* a type specifier followed by an identifier (the variable name), optionally followed by a sequence of identifiers preceded by commas, and terminated by a semicolon.

Syntax rules are easier to learn if they are given as syntax diagrams (Figure 2.1). Circles or

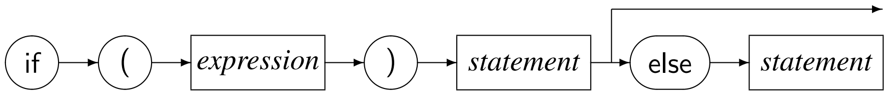

Figure 2.1: Syntax diagram

ovals denote actual characters, while rectangles denote syntactic categories which have their own diagram. A sequence of symbols constructed by tracing a path in the diagrams is a (syntactically) valid program.

While many programmers have a passionate attachment to the syntax of a specific language, this aspect of a language is perhaps the least important. It is easy to learn and to develop familiarity with any reasonable syntax; furthermore, bugs with syntax are caught by the compiler and rarely cause problems with a working program. We will restrict ourselves to noting several possible pitfalls with syntax that can cause run-time bugs:

- Be careful with limitations on the significance of the length of identifiers. If only the first 10 characters are significant, current_winner and current_width will represent the same identifier.

- Many languages are case-insensitive, that is, COUNT and count represent the same name. C is case-sensitive, so that the strings represent two distinct identifiers. In case-sensitive languages, a project should establish clear conventions on the use of case so that accidental typing mistakes do not cause errors. For example, one convention in C requires that everything be in lower-case, except defined constant names which are all in upper-case.

- There are two forms of comments: Fortran, Ada and C++ comments begin with a symbol (C, --, and //, respectively) and extend to the end of a line, while C[1] and Pascal comments have both begin- and end-symbols: /*...*/ in C, and (*...*) or {...} in Pascal. The second form is convenient for "commenting-out" unused code that may have been inserted for testing, but it is possible to omit an end-symbol causing a sequence of statements to be ignored:

  ```
  /*                      Comment should end here          C
  a = b+c;                Statement will be ignored
  /* ... */               Comment ends here
  ```

- Beware of similar but distinct symbols. If you have ever studied mathematics, you should be surprised that the familiar symbol "=" is used for the assignment operator in C and Fortran, while new symbols "==" in C and ".eq." in Fortran are used for the familiar equality operator. The tendency to write:

  ```
  if (a = b) ...                                           C
  ```

  is so common that many compilers will issue a warning even though the statement has a valid meaning.

- For its historical interest, we mention a notorious problem with the syntax of Fortran. Most languages require that words in a program be separated by one or more spaces (or other *whitespace* characters such as tabs), while in Fortran whitespace is ignored. Consider then the following statement, which specifies "loop until label 10 as the index i counts from 1 to 100":

[1]C++ also allows the use of C-style comments.

```
do 10 i = 1, 100
```

Fortran

If the comma is accidentally replaced by a period, the statement becomes a valid assignment statement, assigning 1.100 to the variable whose name is obtained by concatenating all characters before the "=":

```
do10i = 1.100
```

Fortran

This bug is reported to have caused a rocket to explode upon launch into space!

## 2.2 * Semantics

*Semantics* is the meaning of an expression (program) in a (programming) language.

While the syntax of languages is very well understood, semantics is much more difficult. To give an example from English, consider the sentences:

The pig is in the pen.
The ink is in the pen.

It requires quite a lot of general knowledge that has nothing to do with the English language to know that "pen" does not have the same meaning in the two sentences.

Formal notations for programming language semantics are beyond the scope of this book. We will restrict ourselves to a few sentences that give the basic idea. At any point in its execution, we can describe the *state* of a program which is composed of: (1) a pointer to the next instruction to be executed, and (2) the contents of the program's memory.[2] The semantics of a statement is given by specifying the change in state caused by the execution of the statement. For example, executing:

```
a := 25
```

will change the state $s$ to a new state $s'$ that is exactly like $s$ except that the memory location assigned to a contains 25.

For control statements, mathematical logic is used to describe the computation. Assume that we already know the meaning of two statements S1 and S2 in an arbitrary state $s$. This is denoted by the formulas $p(S1,s)$ and $p(S2,s)$, respectively. Then the meaning of the if-statement:

```
if C then S1 else S2
```

is given by the formula:

$$(C(s) \Rightarrow p(S1,s))\,\&\,(\neg C(s) \Rightarrow p(S2,s))$$

[2]More exactly a mapping from every variable to its value.

If C evaluates to *true* in state *s* then the meaning of the if-statement is the same as the meaning of S1; otherwise, C evaluates to *not true* and the meaning is the same as that of S2.

As you can imagine, specifying the semantics of loop statements and procedure calls with parameters can be very difficult. In this book, we will content ourselves with the informal explanations of the semantics of language constructs as commonly written in reference manuals:

> The condition following if is evaluated; if the result is True, the statement following then is executed, otherwise the statement following else is executed.

An additional advantage to a formalization of the semantics of programming languages is that it becomes possible to *prove* the correctness of a program. The effect of an execution of the program can be formalized by axioms that describe how the statement transforms a state that satisfies an input *assertion* (logical formula) to a state that satisfies an output assertion. The meaning of a program is obtained by building input and output assertions for the whole program based on the individual statements. The result is a proof that *if* the input data satisfies the input assertion, *then* the output data satisfies the output assertion.

Of course, the proof of correctness is only relative to the input and output assertions: it does you no good to prove that a program computes a square root if you need a program to compute a cube root! Nevertheless, program verification is a powerful method that has been used in systems that must be highly reliable. More importantly, studying verification will improve your ability to write correct programs, because you will learn to think in terms of the requirements for correctness. We also recommend studying and using the Eiffel programming language which includes support for assertions within the language (Section 11.5).

## 2.3 Data

When first learning a programming language there is a tendency to concentrate on actions: statements or commands. Only when the statements of the language have been studied and exercised, do we turn to study the support that the language provides for structuring data. In modern views of programming such as object-oriented programming, statements are seen as manipulating the data used to represent some object. Thus you should study the data structuring aspects of a language before the statements.

Assembly language programs can be seen as specifications of actions to be performed on *physical* entities such as registers and memory cells. Early programming languages continued this tradition of identifying language entities like *variables* with memory words, even though mathematical names like *integer* were attributed to these variables. In Chapters 4 and 9 we will explain why int and float are not mathematical, but rather physical representations of memory.

We now define the central concept of programming:

> A *type* is a set of *values* and a set of *operations* on those values.

The correct meaning of int in C is: int is a type consisting of a finite set of values (about 65,000 or 4 billion, depending on the computer), and a set of operations (denoted by $+$, ¡=, etc.) on these values. Modern programming languages like Ada and C++ are characterized by their ability to create new types. Thus, we are no longer restricted to the handful of types predefined by the inventor of the language; instead, we can create our own types that correspond more exactly to the problem that we are trying to solve.

The discussion of data types in this book will follow this approach, namely, define a set of values and the operations on these values. Only later will we discuss how such a type can be implemented on a computer. For example, an array is an indexed collection of elements with operations such as indexing. Note that the definition of a type varies with the language: assignment is an operation defined on arrays in Ada but not in C. Following the definition of an array type, the implementation of arrays as sequences of memory cells can be studied.

To conclude this section, we define the following terms that will be used when discussing data:

**Value** A value is an undefined primitive concept.

**Literal** A specific value is denoted in a program by a literal, which is a sequence of symbols, for example: `154`, `45.6`, `FALSE`, `'x'`, `"Hello world"`.

**Representation** A value is represented within a computer by a specific string of bits. For example, the character value denoted by `'x'` may be represented by the string of eight bits 0111 1000.

**Variable** A variable is a name given to the memory cell or cells that can hold the representation of a value of a specific type. The value may be changed during the execution of the program.

**Constant** A constant is a name given to the memory cell or cells that can hold the representation of a value of a specific type. The value may *not* be changed during the execution of the program.

**Object** An object is a variable or a constant.

Note that a variable must be defined to be of a specific type for the simple reason that the compiler must know how much memory to allocate! A constant is simply a variable that can not be modified. Until we discuss object-oriented programming, we will generally use the familiar term variable, rather than the precise term object to denote either a constant or a variable.

## 2.4 The assignment statement

Surprisingly, ordinary programming languages have only one statement that actually does anything: the assignment statement. All other statements such as if-statements and procedure calls exist only to control the sequence of execution of the assignment statements. Unfortunately, it is difficult to give a formal meaning to the assignment statement (as opposed to describing what it does when executed); in fact, you never encountered anything similar when you studied mathematics in high school and college. What you studied was *equations*:

$$ax^2 + bx + c = 0 \qquad\qquad \int \sin x\, dx = -\cos x$$

You transformed equations, you solved them and you evaluated them. Never did you modify them: if $x$ represented a number in one part of the equation, it represented the same number in the rest of the equation.

The lowly assignment statement is actually quite complex, executing three separate tasks:

1. Compute the value of the expression on the right-hand side of the statement.
2. Compute the expression on the left-hand side of the statement; the expression must evaluate to the address of a memory cell.
3. Copy the value obtained in step (1) to memory cells starting with the address obtained in step (2).

Thus the assignment statement:

```
a(i+1) = b + c;
```

despite its superficial resemblance to an equation specifies a complex computation.

## 2.5 Type checking

In the three-step description of assignment, the evaluation of the expression produces a value of a specific type, while the computation of the left-hand-side produces only the starting address of a block of memory. There is no guarantee that the address is associated with a variable of the same type as that of the expression; in fact there is not even a guarantee that the *size* of the value to be copied is the same as the size of the receiving variable.

> *Type checking* is a check that the type of the expression is compatible with the type of the target variable during assignment. This includes the assignment of an actual parameter to a formal parameter when a procedure is called.

Possible approaches to type checking are:

- Do nothing; it is the programmer's responsibility to ensure that the assignment is meaningful.
- Implicitly convert the value of the expression to the type required by the left-hand side.
- *Strong* type checking: refuse to execute the assignment if the types are not the same.

There is a clear trade-off between flexibility and reliability: the more type checking that is done the more reliable a program will be, but it will require more programming effort to define an appropriate set of types. In addition, provision must be made to bypass type checking if needed. Conversely, if little type checking is done it is easy to write a program, but it then becomes difficult to find bugs and to ensure the reliability of the program. A further disadvantage to type checking is that it may require run-time overhead to implement. Implicit type conversion can be as bad as, if not worse than, doing nothing because it gives a false sense of security that all is well.

Strong type checking can eliminate obscure bugs which are usually caused by such errors or misunderstandings. It is especially important in large software projects which are developed by teams of programmers; breakdowns in communications, personnel turnovers, etc. make it difficult to integrate such software without the constant checking done by strong type checking. In effect, strong type checking attempts to transform run-time errors into compile-time errors. Run-time errors can be extremely difficult to find, dangerous to the users of the software and expensive for the developer of the software in terms of delayed delivery and damaged reputation. The cost of a compile-time error is trivial; in fact, you probably don't even have to tell your boss that you made a compile-time error.

## 2.6 Control statements

Assignment statements are normally executed in the sequence in which they are written. Control statements are used to modify the order of execution. Assembly language programs use arbitrary jumps from one instruction address to another. By analogy, a programming language can include a goto-statement which jumps to a label on an arbitrary statement. Programs using arbitrary jumps are difficult to read, and hence to modify and maintain.

*Structured programming* is the name given to the programming style which restricts the use of control statements to those which yield well-structured programs that are easy to read and understand. There are two classes of well-structured control statements:

- Choice statements that select one alternative from two or more possible execution sequences: if-statements and case- or switch-statements.

- Loop statements that repeatedly execute a sequence of statements: for-statements and while-statements.

A good understanding of loop statements is particularly important for two reasons: (1) most of the execution time will (obviously) be spent within loop statements, and (2) many bugs are caused by incorrect coding at the beginning or end of a loop.

## 2.7 Subprograms

A *subprogram* is a unit consisting of data declarations and executable statements that can be invoked (*called*) repeatedly from different parts of a program. Subprograms (called *procedures*, *functions*, *subroutines* or *methods*) were originally used just to enable such reuse of a program segment. The more modern view is that subprograms are an essential element of program structure, and that every program segment that does some identifiable task should be placed in a separate subprogram.

When a subprogram is called, it is passed a sequence of values called *parameters*. Parameters are used to modify each execution of the subprogram, to send data to the subprogram, and to receive the results of a computation. Parameter passing mechanisms differ widely from one language to another, and programmers must fully understand their effect on the reliability and efficiency of a program.

## 2.8 Modules

The elements of the language discussed thus far are sufficient for writing *programs*; they are not sufficient for writing a *software system*: a very large program or set of programs developed by teams of programmers. Students often extrapolate from their talent at writing (small) programs, and conclude that writing a software system is no different, but bitter experience has shown that writing a large system requires additional methods and tools beyond mere programming. The term *software engineering* is used to denote the methods and tools for designing, managing and constructing software systems. In this text we limit ourselves to discussing support for large systems that can be given by programming languages.

You may have been told that a single subprogram should be limited to 40 or 50 lines, because a programmer cannot easily read and comprehend larger segments of code. By the same measure, it should be easy to understand the interactions of 40 or 50 subprograms. The obvious conclusion is that any program larger than 1600-2500 lines is hard to understand! Since useful programs can have tens of thousands of lines, and systems of hundreds of thousands of lines are not uncommon, it is clear that additional structures are needed to construct large systems.

If older programming languages are used, the only recourse is to "bureaucracy": sets of rules and conventions that prescribe to the team members how programs are to be written. Modern programming languages contain an additional structuring method for *encapsulating*[3] data and subprograms within larger entities called *modules*. The advantage of modules over bureaucracy is that the interfaces between modules can be checked during compilation to prevent errors and misunderstandings. Furthermore, the actual executable statements, and most (or all) of the data of a module, can be hidden so that they cannot be modified or used except as specified by the interface.

The are two potential difficulties with the practical use of modules:

- A powerful program development environment is needed to keep track of the modules and to check the interfaces.
- Modularization encourages the use of many small subprograms with the corresponding run-time overhead of the subprogram call.

Neither of these is now a problem: the average personal computer is more than adequate to run an environment for C++ or Ada, and modern computer architectures and compilation techniques minimize the overhead of calls.

[3]From the word "capsule", container.

The fact that a language has support for modules does not help us to decide what to put in a module. In other words, how do we decompose a software system into modules? Since the quality of the system is directly dependent on the quality of the decomposition, the competence of a software engineer should be judged on his/her ability to analyze a project's requirements and to create the best software structure to implement the project. It requires much experience to develop this ability; perhaps the best way is to study existing software systems.

Despite the fact that sound engineering judgement cannot be taught, there are certain principles that can be studied. One of the leading methods for guiding program decomposition is *object-oriented programming* (OOP) which builds on the concept of type discussed above. According to OOP, a module should be constructed for any real-world or abstract "object" that can be represented by a set of data and operations on that data. Chapters 14 and 15 contain a detailed discussion of language support for OOP.

## 2.9 Exercises

1. Translate (part of) the BNF syntax of C or Ada into syntax diagrams.
2. Write a program in Pascal or C that compiles and executes, but computes the wrong answer because of a comment that was not closed.
3. Even if Ada used the style of comments used in C and Pascal, bugs caused by not closing comments would be less frequent. Why?
4. In most languages, keywords like `begin` and `while` are reserved and may not be used as identifiers. Other languages like Fortran and PL/I do not have reserved keywords. What are the advantages and disadvantages of reserved words?