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 ovals denote actual characters, while rectangles denote syntactic categories which have their own

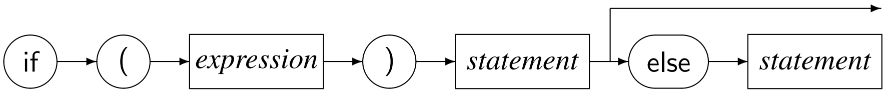

Figure 2.1: Syntax diagram

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:

  C

  ```
  /*                    Comment should end here
  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:

  C

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

  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.

Fortran

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

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 "=":

Fortran

```
do10i = 1.100
```

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?

3 Programming Environments

A language is a set of rules for writing programs which are no more than sequences of symbols. This chapter will review the components of a *programming environment*—the set of tools used to transform symbols into executable computations:

Editor An editor is a tool used to create and modify *source* files, which are files of characters that comprise a program in a language.

Compiler A compiler translates the symbols in a source file into an *object module*, which contains machine code instructions for a specific computer.

Librarian A librarian maintains collections of object files called libraries.

Linker A linker collects the object files of the components of the program and *resolves* external references from one component to another in order to form an *executable* file.

Loader A loader copies the executable file from the disk into memory, and initializes the computer before executing the program.

Debugger A debugger is a tool that enables the programmer to control the execution of a program at the level of individual statements, so that bugs can be diagnosed.

Profiler A profiler measures the relative amount of time spent in each component of a program. The programmer can then concentrate on improving the efficiency of *bottlenecks*: components responsible for most of the execution time.

Testing tools Testing tools automate aspects of the software testing process by creating and running tests, and analyzing the test results.

Configuration tools Configuration tools automate the creation of programs and track modifications to source files.

Interpreter An interpreter directly executes the source code of a program, rather than translating the source into an object file.

A programming environment can be constructed from separate tools; alternatively, many vendors sell *integrated* programming environments, which are systems containing most or all of these tools. The advantage of an integrated environment is that the user interface is extremely simple: each tool is initiated using a single key or menu selection, rather than typing file names and parameters.

3.1 Editor

Every programmer has his/her favorite general-purpose editor. Even so, you may wish to consider using a *language-sensitive* editor, which is an editor that can create an entire language construct such as an if-statement with one key-press. The advantage of a language-sensitive editor is that syntax errors are prevented. However, touch-typists may find that it is easier to type language constructs rather than to search for the menu selection.

The reference manual for a language may specify a recommended layout for the source code: indentation, line breaks, upper/lower case. These rules do not affect the correctness of the program, but for the benefit of future generations that will read your program, such conventions should be respected. If you have failed to follow the conventions while writing the program, you can use a tool called a *pretty-printer* which rewrites the source code in the recommended format. Since a pretty-printer can inadvertently introduce mistakes, it is preferable to respect conventions from the beginning.

3.2 Compiler

A programming language without a compiler (or interpreter) may be of great theoretical interest, but you cannot execute a program in the language on a computer. The relationship between languages and compilers is so close that the distinction can become blurred, and one often hears such nonsense as:

> Language L1 is more efficient than language L2.

What *is* true is that compiler C1 may emit more efficient code than compiler C2, or that a construct of L1 may be easier to compile efficiently than a corresponding construct of L2. One of the aims of this text is to show the relation between language constructs and resulting machine code after compilation.

The structure of a compiler is shown in Figure 3.1. The *front-end* of a compiler "understands" the program by analyzing its syntax and semantics according to the rules of the language. The syntax analyzer is responsible for transforming sequences of characters into abstract syntactical entities called *tokens*. For example, the character "=" in C is transformed into the assignment operator, unless it is followed by another "="; in that case the two characters together "==" are transformed into the equality operator. The semantics analyzer is responsible for assigning a meaning to these abstract entities. For example, in the following program the semantic analyzer assigns a global address to the first i, and a parameter offset to the second i:

C

```
static int i;
void proc(int i) { . . . }
```

The output of the front-end of the compiler is an abstract representation of the program called the *intermediate representation*. From it you could reconstruct the source program except for the names of the identifiers and the physical format of lines, spaces, comments, etc.

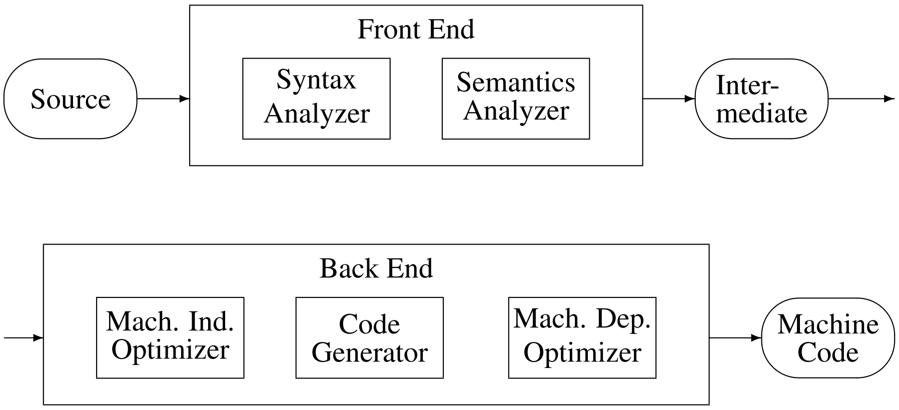

Figure 3.1: The structure of a compiler

Research into compilers has advanced to the point where a front-end can be automatically generated from a *grammar* (a formal description) of the language. Readers who are interested in designing a programming language are strongly advised to study compilation in depth, so that the language can be easy to compile using automated techniques.

The *back-end* of a compiler takes the intermediate representation and emits *machine code* for a specific computer. Thus the front-end is *language-dependent* while the back-end is *computer-dependent*. A compiler vendor can produce a family of compilers from some language L to a set of widely differing computers C1, C2, ..., by writing a set of back-ends which use the intermediate representation of a common front-end. Similarly, a computer vendor can create a high-quality back-end for a computer C, and then support a large number of languages L1, L2, ..., by writing front-ends that translate the source of each language to the common intermediate code. In this case, it truly makes no sense to ask which language is most efficient on the computer.

Associated with the code generator is the *optimizer* which attempts to improve the code to make it more efficient. Several classes of optimization are possible:

- Optimization on the intermediate representation, for example, common subexpression extraction:

  ```
  a = f1(x+y) + f2(x+y);
  ```

  Rather than compute the expression x+y twice, it can be computed once and stored in a temporary variable or register. Optimizations such as this do not depend on a specific target computer and can be done before code generation. This means that even part of the back-end can be shared by compilers for different computers.

- Computer-specific optimization. Optimizations such as storing intermediate results in registers rather than in memory must clearly be done during code generation, because the number and type of registers differ from one computer to another.

- *Peephole optimization* is usually done on the generated instructions, though sometimes it can be done on the intermediate representation. This optimization technique attempts to replace short sequences of instructions with a single, more efficient instruction. For example, the C expression n++ might be compiled into:

```
load     R1,n
add      R1,#1
store    R1,n
```

but a peephole optimizer for a specific computer might be able to replace the three instructions with one that directly increments a memory word:[1]

```
incr     n
```

Optimizers must be used with care. Since by definition an optimizer modifies a program, it may be difficult to debug using a source debugger, because the order of the statements as executed may be different from the order in the source code. It is usually necessary to turn off the optimizer when debugging. Furthermore, the complexity of an optimizer means that it is more likely to contain bugs than any other component of the compiler. An optimizer bug is difficult to diagnose because the debugger is designed to work on source code, not on optimized (that is, modified) object code. In no circumstances should a program be tested without using the optimizer and then distributed after optimization. Finally, the optimizer may make assumptions that are incorrect in any given situation. For example, when using memory-mapped I/O devices, a variable may have a value assigned to it twice without an intervening read:

C

```
transmit_register = 0x70;
/* Wait 1 second */
transmit_register = 0x70;
```

The optimizer will assume that the second assignment is redundant and will remove it from the generated object code.

3.3 Librarian

Object modules can be stored as individual files, or a set of modules can be stored in a single file called a library. Libraries can be provided with the compiler, purchased separately or created by the programmer.

Many of the constructs in a programming language are not implemented by compiled code within the program, but by calls to procedures that are stored in a library provided by the compiler vendor. As languages grow larger, there is a trend towards placing more functionality into "standard" libraries that are an indivisible part of the language. Since the library is just a structured collection of types and subprograms and does not contain new language constructs, it simplifies the tasks of both the student who must learn the language and the implementor of a compiler.

[1] In fact C was first implemented on the DEC PDP-11 computer which has increment memory and decrement memory instructions. These instructions were the motivation for the corresponding operators in C.

The essential set of procedures needed for initialization, memory management, expression evaluation, etc., is called the *run-time system.* It is essential that a programmer be familiar with the run-time system of the compiler being used: innocent-looking language constructs may in fact invoke time-consuming procedures in the run-time system. For example, if high-precision arithmetic is implemented by library procedures, then changing all integers to long integers will significantly increase execution time.

The Java Libraries

The trend in programming language design is to limit the scope of the language by providing functionality in standard libraries. For example, write is a statement in the Pascal language with a special syntax, while in Ada there are no I/O statements; instead, I/O is supported through packages in the standard library.

The standard Ada libraries supply computational facilities such as I/O, character and string processing, mathematical functions, and system interfaces. C++ also supports container classes such as stacks and queues. Similarly, Java contains basic libraries called java.lang, java.util and java.io, which are part of the language specification.

In addition to the language specification, there is a specification for the Application Programming Interface (API) that all Java implementations are supposed to support. The API consists of three libraries: java.applet, java.awt and java.net.

java.applet contains support for creating and executing applets and for creating multimedia applications. The *Abstract Window Toolkit* (AWT) is a library for creating graphical user interfaces (GUI): windows, dialog boxes and bit-mapped graphics. The library for network communications provides the essential interface for locating and transferring data on the net.

3.4 Linker

It is perfectly possible to write a program that is several thousand lines long in a single file or module. However, large software systems, especially those that are developed by teams, require that the software be decomposed into modules (Chapter 13). If a call is made to a procedure not in the current module, the compiler has no way of knowing the address of the procedure. Instead a note of the *external reference* is made in the object module. If a language allows access to global variables from several modules, external references must be created for each such access. When all modules have been compiled, the linker resolves these references by searching for definitions of procedures and variables that have been *exported* from a module for non-local use.

Modern programming practice calls for extensive use of modules to decompose a program. A side-effect of this practice is that compilations are usually short and fast, while the linker is called upon to link hundreds of modules with thousands of external references. The efficiency of the linker can be critical to a software team's performance in the final stages of a project: even a small change to one source module will require a time-consuming link.

One solution to this problem is to link subsystems together and only then resolve the links between the subsystems. Another solution is to use *dynamic linking* if available on the system. In dynamic linking, external references are not resolved; instead the first call to a procedure is trapped by the operating system and resolved. Dynamic linking may be combined with *dynamic loading*: not only are the references unresolved, but a module is not even loaded until one of its exported procedures is needed. Of course dynamic linking or loading entails additional overhead at run-time, but it is a powerful method of adapting systems to changing requirements without relinking.

3.5 Loader

As its name implies, the loader loads a program into memory and initializes it for execution. On older computers, the loader was non-trivial because it had to *relocate* programs. An instruction such as `load 140` referred to an absolute memory address and had to be fixed according to the actual address where the program was loaded. Modern computers address code and data relative to registers. A register will be allocated to point to each code and data area, so all the loader has to do is to copy the executable program into memory and initialize a set of registers. The instruction `load 140` now means "load the value located in the address obtained by adding 140 to the contents of the register that points to the data area".

3.6 Debugger

Debuggers support three functions:

Tracing The execution of a program step-by-step so that the programmer can precisely follow the order in which statements are executed.

Breakpoints A means to specify that a program will run until a specific line in the program has been reached. A special form of a breakpoint is a *watchpoint*, which causes the program to run until a specific memory location is accessed.

Examine/modify data The ability to examine and modify the value of any variable at any point in the computation.

Symbolic debuggers work in terms of symbols of the source code (variable and procedure names) rather than in terms of absolute machine addresses. A symbolic debugger requires the cooperation of the compiler and linker to create tables relating symbols and their addresses.

Modern debuggers are extremely powerful and flexible, to the point where they can be abused as a substitute for thinking. Often, just trying to explain a procedure to another programmer can locate a bug faster than days of tracing.

Some problems are not easily solved even using a debugger. For example, dynamic data structures such as lists and trees cannot be examined as a whole; instead you must manually traverse each link. More serious are problems like smearing memory (see Section 5.4), which are caused by bugs that are far removed from the area where the symptoms appear. Debuggers are used for investigating symptoms such as “zero-divide in procedure p1”, and are of limited use in solving such bugs.

Finally, some systems cannot be “debugged” as such: you cannot create a medical emergency at will just to debug the software for a heart monitor; you cannot send a team of programmers along on a spaceflight to debug the flight control program. Such systems must be tested using special hardware and software to simulate the program inputs and outputs; the software is thus never tested and debugged in actual use! Critical software systems motivate research into language constructs improving program reliability and formal methods of program verification.

3.7 Profiler

It is often said that more bugs are caused by attempts at efficiency than by all other causes. What makes this saying so tragic is that most attempts at improving efficiency achieve nothing, or at most, improvements that are not commensurate with the effort invested. Throughout this book we will discuss the relative efficiency of various program constructs, but this information should be used only if three conditions are met:

- The current performance of the program is unacceptable.
- There is no better way to improve efficiency. In general, choosing a more efficient algorithm will give better results than improving the programming of an existing one (see Section 6.5 for an example).
- The cause of the inefficiency can be identified.

Identifying inefficiencies is extremely difficult without the aid of a measuring tool. The reason is that there are so many orders of magnitude between time intervals that we instinctively understand (seconds), and the time intervals in which computers work (micro- or nano-seconds). A function which seems difficult to us may have little effect on the overall execution time of a program.

A profiler samples the instruction pointer of the computer at regular intervals and then builds a histogram displaying the percentage of execution time that can be attributed to each procedure or statement. Very often the result will surprise the programmer by identifying bottlenecks that are not at all evident. Investing effort in optimizing a program without the use of a profiler is totally unprofessional.

Even with a profiler, improving the efficiency of a program can be difficult. One reason is that much of the execution time is being spent in externally provided components such as database or windowing subsystems, which are often designed more for flexibility than for performance.

3.8 Testing tools

Testing a large system can take as much time as the initial programming and debugging. Software tools have been developed to automate several aspects of testing. One tool is a *coverage analyzer* which keeps track of which statements have been tested. However, such a tool does nothing to help create and execute the tests.

More sophisticated tools execute predefined tests and then compare the output to a specification. Tests may also be generated automatically by *capturing* input from an external source such as a user pressing keys on a keyboard. The captured input sequence can then be run repeatedly. Whenever a new version of a software system is to be released, previous tests should be rerun. Such *regression testing* is essential because the assumptions underlying a program are so interconnected, that any modification may cause bugs even in modules where "nothing has changed".

3.9 Configuration tools

Configuration tools are used to automate bureaucratic tasks associated with software. A *make* tool creates an executable file from source code by invoking the compiler, linker, etc. In a large project, it can be difficult to keep track of exactly which files need to be recompiled, in which order, and with which parameters, and it is very easy to fix a bug and then cause another one by using an outdated object module. A make tool automatically ensures that a new module is correctly created with a minimum amount of recompilation.

A *source control* or *revision control* tool is used to maintain a record of all changes to the source code modules. This is important because in a large project it is often necessary to "back off" a modification that causes unforeseen problems, or to be able to trace a modification to a specific version or programmer. In addition, a program may be delivered in different versions to different customers; without a software tool, we would have to fix a common bug in all the versions. A source control tool simplifies these tasks because it only maintains changes (called *deltas*) from the original version, and can easily reconstruct any previous version from the changes.

3.10 Interpreters

An *interpreter* is a program that directly executes the source code of a program. The advantage of an interpreter over a compiler is that it can be extremely convenient to use, because it does not require that a sequence of tools be invoked: compiler, linker, loader, etc. Interpreters are also easy to write because they need not be computer-specific; since they directly execute the program they have no machine code output. Thus an interpreter written in a standardized language is portable. An additional simplification comes from the fact that traditionally interpreters do not attempt any optimization.

In fact, it can be very difficult to distinguish an interpreter from a compiler. Very few interpreters actually execute the source code of a program; instead they translate (that is, compile) the source into the machine code of an imaginary machine, and then execute the abstract machine code (Figure 3.2).

Suppose now that someone invents a computer whose machine code is exactly this abstract code; alternatively, suppose that someone writes a set of macros that replace the abstract machine code

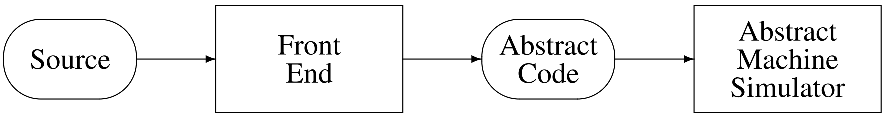

Figure 3.2: The structure of an interpreter

by the actual machine code of an existing computer. In either case, the so-called interpreter has become a compiler without changing a single line of the program.

The original Pascal compiler was written to produce machine code for a specific machine (CDC 6400). Soon afterwards, Niklaus Wirth created a compiler that produced code, called *P-Code*, for an abstract stack-based machine. By writing an interpreter for P-Code, or by compiling the P-Code into machine code for a specific machine, it is possible to create a Pascal interpreter or compiler with relatively little effort. The compiler for P-Code was a decisive factor in making Pascal the widespread language that it is today.

The logic programming language Prolog (see Chapter 17) was considered at first to be a language that was only suitable for interpretation. David Warren produced the first true compiler for Prolog by defining an abstract machine (the *Warren Abstract Machine*, or *WAM*) that manipulated the basic data structures required to execute programs in the language. Both compiling Prolog to WAM programs and compiling WAM programs into machine code are not too difficult; Warren's achievement was to define the WAM at the right intermediate level between the two. Much research on compilation of logic programming languages has been based on the WAM.

There does not seem to be much practical value in arguing about the difference between a compiler and interpreter. When comparing programming environments, the emphasis should be on reliability, correct translation, high performance, efficient code generation, good debugging facilities and so on, and not on the implementation techniques used to create the environment.

3.11 The Java model

You can write a compiler for the Java language exactly as you would for any other imperative, object-oriented language. The Java model, however, is based on the concept of an interpreter (see Figure 3.2, reproduced in modified form as Figure 3.3) like the Pascal interpreters we discussed in Section 3.10.

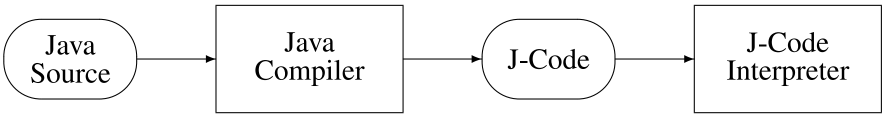

Figure 3.3: An interpreter for Java

In Java, however, the arrow from the J-Code to the J-Code interpreter does not represent simply the flow of data between components of a software development environment. Instead, the J-Code can be packaged in an *applet*[2] that can be transmitted across the communications system of a network. The receiving computer executes the J-Code using an interpreter called a *Java Virtual Machine (JVM)*. The JVM is usually embedded within a *net browser*, which is a program for retrieving and displaying information obtained from a network. When the browser identifies that an applet has been received, it invokes the JVM to execute the J-Code. Furthermore, the Java model includes standard libraries for graphical user interfaces, multimedia and network communications, which are mapped by each implementation of the JVM onto the facilities of the underlying operating system. This is a significant extension of the concept of a virtual machine beyond the simple computation of Pascal P-Code.

Note that you can also write an ordinary program in Java which is called an *application*. An application is not subject to the security limitations discussed below. Unfortunately, there are a few irritating differences between the programming of an applet and an application.

Performance

Too good to be true? Of course. The Java model suffers from the same performance problems that affect any model based on interpretation of an abstract machine code. For relatively simple programs running on powerful personal computers and workstations, this will not be a serious problem, but performance may set a limit on the applicability of the Java model.

One solution is to include, *within* the browser on the receiving end, a compiler that translates the abstract J-Code into machine code for the receiving computer. In fact, the computer can carry out this translation simultaneously, or nearly so, with the reception of the J-Code; this is called *on-the-fly compilation*. At worst, performance will improve the *second* time you execute the applet you have loaded.

However, this solution is only a partial one. It would not be practical or economical to bundle a sophisticated optimizing compiler within each browser for each combination of computer and operating system. Nevertheless, for many types of applications, the Java model will be very successful for developing portable software.

Security

Suppose that you download an applet and execute it on your computer. How do you know that the applet is not going to erase your hard disk? Considering the extensive damage that has been caused by viruses maliciously appended to otherwise valid programs, you will appreciate how dangerous it is to download arbitrary programs from a remote location. The Java model employs several strategies to eliminate (or at least reduce!) the possibility that an applet, downloaded from a remote location, will damage the software and data on the receiving computer:

Strong type-checking As in Ada, the idea is to eliminate accidental or malicious damage to data caused by out-of-bound array indices, arbitrary or dangling pointers, and so on. We will elaborate on this in the following sections.

[2] Applet is a new term derived from "application".

J-Code verification The interpreter on the receiving computer verifies that the stream of bytes received from the remote computer actually consists of legal J-Code instructions. This ensures that the secure semantics of the model is actually what is executed, and that the interpreter cannot be "spoofed" into causing damage.

Applet restrictions An applet is not allowed to perform certain operations on the receiving computer, such as writing or erasing files. This is the most problematical aspect of the security model, because we would like to write applets that can do anything that an ordinary program can.

Clearly, the success of Java depends on the degree to which the security model is strict enough to prevent malicious use of the JVM, while at the same time retaining enough flexibility to enable useful programs to be built.

Language independence

The astute reader will have noticed that the previous section has been written without reference to the Java programming language! This is intentional, because the Java model is both valid and useful even if the source code of the applets is written in some other language. For example, there are compilers that translate Ada 95 into J-Code. However, the Java language was designed together with the JVM and the language semantics closely match the capabilities of the model.

3.12 Exercises

1. Study your compiler's documentation and list the optimizations that it performs. Write programs and check the resulting object code for the optimizations.

2. What information does the debugger need from the compiler and linker?

3. Run a profiler and study how it works.

4. How can you write your own simple testing tool? What influence does automated testing have on program design?

5. AdaS is an interpreter for a subset of Ada written in Pascal. It works by compiling the source into P-Code and then executing the P-Code. Study the AdaS program and write a description of the P-Code machine.