CHAPTER ONE

Introduction

In this introductory chapter we start by reviewing the distinction between low-level and high-level programming languages. We then see what is meant by a programming language processor, and look at examples from different programming systems. We review the specification of the syntax and semantics of programming languages. Finally, we look at Triangle, a programming language that will be used as a case study throughout this book.

1.1 Levels of programming language

Programming languages are the basic tools of all programmers. A programming language is a formal notation for expressing algorithms. Now, an algorithm is an abstract concept, and has an existence independent of any particular notation in which it might be expressed. Without a notation, however, we cannot express an algorithm, nor communicate it to others, nor reason about its correctness.

Practicing programmers, of course, are concerned not only with expressing and analyzing algorithms, but also with constructing software that instructs machines to perform useful tasks. For this purpose programmers need facilities to enter, edit, translate, and interpret programs on machines. Tools that perform these tasks are called *programming language processors*, and are the subject of this book.

Machines are driven by programs expressed in ***machine code*** (or *machine language*). A machine-code program is a sequence of *instructions*, where each instruction is just a bit string that is interpreted by the machine to perform some defined operation. Typical machine-code instructions perform primitive operations like the following:

- Load an item of data from memory address 366.
- Add two numbers held in registers 1 and 2.
- Jump to instruction 13 if the result of the previous operation was zero.

In the very early days of computing, programs were written directly in machine code. The above instructions might be written, respectively, as follows:

- 0000 0001 0110 1110

- 0100 0000 0001 0010
- 1100 0000 0000 1101

Once written, a program could simply be loaded into the machine and run.

Clearly, machine-code programs are extremely difficult to read, write, and edit. The programmer must keep track of the exact address of each item of data and each instruction in storage, and must encode every single instruction as a bit string. For small programs (consisting of thousands of instructions) this task is onerous; for larger programs the task is practically infeasible.

Programmers soon began to invent symbolic notations to make programs easier to read, write, and edit. The above instructions might be written, respectively, as follows:

- `LOAD x`
- `ADD R1 R2`
- `JUMPZ h`

where `LOAD`, `ADD`, and `JUMPZ` are symbolic names for operations, `R1` and `R2` are symbolic names for registers, `x` is a symbolic name for the address of a particular item of data, and `h` is a symbolic name for the address of a particular instruction. Having written a program like this on paper, the programmer would prepare it to be run by manually translating each instruction into machine code. This process was called *assembling* the program.

The obvious next step was to make the machine itself assemble the program. For this process to work, it is necessary to standardize the symbolic names for operations and registers. (However, the programmer should still be free to choose symbolic names for data and instruction addresses.) Thus the symbolic notation is formalized, and can now be termed an ***assembly language***.

Even when writing programs in an assembly language, the programmer is still working in terms of the machine's instruction set. A program consists of a large number of very primitive instructions. The instructions must be written individually, and put together in the correct sequence. The algorithm in the mind of the programmer tends to be swamped by details of registers, jumps, and so on. To take a very simple example, consider computing the area of a triangle with sides a, b, and c, using the formula:

$$\sqrt{(s \times (s-a) \times (s-b) \times (s-c))}$$
where $s = (a + b + c) / 2$

Written in assembly language, the program must be expressed in terms of individual arithmetic operations, and in terms of the registers that contain intermediate results:

```
LOAD R1 a;  ADD R1 b;  ADD R1 c;  DIV R1 #2;
LOAD R2 R1;
LOAD R3 R1;  SUB R3 a;  MULT R2 R3;
LOAD R3 R1;  SUB R3 b;  MULT R2 R3;
```

```
LOAD R3 R1;  SUB R3 c;  MULT R2 R3;
LOAD R0 R2;  CALL sqrt
```

Programming is made very much easier if we can use notation similar to the familiar mathematical notation:

```
let s = (a+b+c)/2
in sqrt(s*(s-a)*(s-b)*(s-c))
```

Today the vast majority of programs are written in programming languages of this kind. These are called ***high-level languages***, by contrast with machine languages and assembly languages, which are ***low-level languages***. Low-level languages are so called because they force algorithms to be expressed in terms of primitive instructions, of the kind that can be performed directly by electronic hardware. High-level languages are so called because they allow algorithms to be expressed in terms that are closer to the way in which we conceptualize these algorithms in our heads. The following are typical of concepts that are supported by high-level languages, but are supported only in a rudimentary form or not at all by low-level languages:

- *Expressions:* An expression is a rule for computing a value. The high-level language programmer can write expressions similar to ordinary mathematical notation, using operators such as '+', '−', '*', and '/'.
- *Data types:* Programs manipulate data of many types: primitive types such as truth values, characters, and integers, and composite types such as records and arrays. The high-level language programmer can explicitly define such types, and declare constants, variables, functions, and parameters of these types.
- *Control structures:* Control structures allow the high-level language programmer to program selective computation (e.g., by if- and case-commands) and iterative computation (e.g., by while- and for-commands).
- *Declarations:* Declarations allow the high-level language programmer to introduce identifiers to denote entities such as constant values, variables, procedures, functions, and types.
- *Abstraction:* An essential mental tool of the programmer is abstraction, or separation of concerns: separating the notion of *what* computation is to be performed from the details of *how* it is to be performed. The programmer can emphasize this separation by use of named procedures and functions. Moreover, these can be parameterized with respect to the entities on which they operate.
- *Encapsulation* (or *data abstraction*)*:* Packages and classes allow the programmer to group together related declarations, and selectively to hide some of them. A particularly important usage of this concept is to group hidden variables together with operations on these variables, which is the essence of object-oriented programming.

Section 1.5 suggests further reading on the concepts of high-level programming languages.

1.2 Programming language processors

A ***programming language processor*** is any system that manipulates programs expressed in some particular programming language. With the help of language processors we can run programs, or prepare them to be run.

This definition of language processors is very general. It encompasses a variety of systems, including the following:

- ***Editors***. An editor allows a program text to be entered, modified, and saved in a file. An ordinary text editor lets us edit any textual document (not necessarily a program text). A more sophisticated kind of editor is one tailored to edit programs expressed in a particular language.
- ***Translators*** and ***compilers***. A translator translates a text from one language to another. In particular, a compiler translates a program from a high-level language to a low-level language, thus preparing it to be run on a machine. Prior to performing this translation, a compiler checks the program for syntactic and contextual errors.
- ***Interpreters***. An interpreter takes a program expressed in a particular language, and runs it immediately. This mode of execution, omitting a compilation stage in favor of immediate response, is preferred in an interactive environment. Command languages and database query languages are usually interpreted.

In practice, we use all the above kinds of language processor in program development. In a conventional programming system, these language processors are usually separate tools; this is the 'software tools' philosophy. However, most systems now offer integrated language processors, in which editing, compilation, and interpretation are just options within a single system. The following examples contrast these two approaches.

Example 1.1 Language processors as software tools

The 'software tools' philosophy is well exemplified by the UNIX operating system. Indeed, this philosophy was fundamental to the system's design.

Consider a UNIX user developing a chess-playing application in Java, using the Sun Java Development Kit (JDK). The user invokes an editor, such as the screen editor `vi`, to enter and store the program text in a file named (say) `Chess.java`:

```
vi Chess.java
```

Then the user invokes the Java compiler, `javac`:

```
javac Chess.java
```

This translates the stored program into object code, which it stores in a file named `Chess.class`. The user can now test the object-code program by running it using the interpreter, `java`:

```
java Chess
```

If the program fails to compile, or misbehaves when run, the user reinvokes the editor to modify the program; then reinvokes the compiler; and so on. Thus program development is an edit–compile–run cycle.

There is no direct communication between these language processors. If the program fails to compile, the compiler will generate one or more error reports, each indicating the position of the error. The user must note these error reports, and on reinvoking the editor must find the errors and correct them. This is very inconvenient, especially in the early stages of program development when errors might be numerous.

□

The essence of the 'software tools' philosophy is to provide a small number of common and simple tools, which can be used in various combinations to perform a large variety of tasks. Thus only a single editor need be provided, one that can be used to edit programs in a variety of languages, and indeed other textual documents too.

What we have described is the 'software tools' philosophy in its purest form. In practice, the philosophy is compromised in order to make program development easier. The editor might have a facility that allows the user to compile the program (or indeed issue any system command) without leaving the editor. Some compilers go further: if the program fails to compile, the editor is automatically reinvoked and positioned at the first error.

These are *ad hoc* solutions. A fresh approach seems preferable: a fully integrated language processor, designed specifically to support the edit–compile–run cycle.

Example 1.2 Integrated language processor

Borland JBuilder is a fully integrated language processor for Java, consisting of an editor, a compiler, and other facilities. The user issues commands to open, edit, compile, and run the program. These commands may be selected from pull-down menus, or from the keyboard.

The editor is tailored to Java. It assists with the program layout using indentation, and it distinguishes between Java keywords, literals and comments using color. The editor is also fully integrated with the visual interface construction facilities of JBuilder.

The compiler is integrated with the editor. When the user issues the 'compile' command, and the program is found to contain a compile-time error, the erroneous phrase is highlighted, ready for immediate editing. If the program contains several errors, then the compiler will list all of them, and the user can select a particular error message and have the relevant phrase highlighted.

The object program is also integrated with the editor. If the program fails at run-time, the failing phrase is highlighted. (Of course, this phrase is not necessarily the one that contains the logical error. But it would be unreasonable to expect the language processor to debug the program automatically!)

□

1.3 Specification of programming languages

Several groups of people have a direct interest in a programming language: the *designer* who invented the language in the first place; the *implementors*, whose task it is to write language processors; and the much larger community of ordinary *programmers*. All of these people must rely on a common understanding of the language, for which they must refer to an agreed ***specification*** of the language.

Several aspects of a programming language need to be specified:

- ***Syntax*** is concerned with the form of programs. A language's syntax defines what tokens (symbols) are used in programs, and how phrases are composed from tokens and subphrases. Examples of phrases are commands, expressions, declarations, and complete programs.
- ***Contextual constraints*** (sometimes called *static semantics*) are rules such as the following. *Scope rules* determine the scope of each declaration, and allow us to locate the declaration of each identifier. *Type rules* allow us to infer the type of each expression, and to ensure that each operation is supplied with operands of the correct types. Contextual constraints are so called because whether a phrase such as an expression is well-formed depends on its context.
- ***Semantics*** is concerned with the meanings of programs. There are various points of view on how we should specify semantics. From one point of view, we can take the meaning of a program to be a mathematical function, mapping the program's inputs to its outputs. (This is the basis of *denotational semantics*.) From another point of view, we can take the meaning of a program to be its behavior when it is run on a machine. (This is the basis of *operational semantics*.) Since this book is about language processors, i.e., systems that run programs or prepare them to be run, we shall prefer the operational point of view.

When a programming language is specified, there is a choice between formal and informal specification:

- An ***informal specification*** is one written in English or some other natural language. Such a specification can be readily understood by any user of the programming language, if it is well-written. Experience shows, however, that it is very hard to make an informal specification sufficiently precise for all the needs of implementors and programmers; misinterpretations are common. Even for the language designer, an informal specification is unsatisfactory because it can too easily be inconsistent or incomplete.
- A ***formal specification*** is one written in a precise notation. Such a specification is more likely to be unambiguous, consistent, and complete, and less likely to be misinterpreted. However, a formal specification will be intelligible only to people who understand the notation in which the specification is written.

In practice, most programming language specifications are hybrids. Syntax is usually specified formally, using BNF or one of its variants, because this notation is easy and

widely understood. But contextual constraints and semantics are usually specified informally, because their formal specification is more difficult, and the available notations are not yet widely understood. A typical language specification, with formal syntax but otherwise informal, may be found in Appendix B.

1.3.1 Syntax

Syntax is concerned with the form of programs. We can specify the syntax of a programming language formally by means of a ***context-free grammar***. This consists of the following elements:

- A finite set of ***terminal symbols*** (or just *terminals*). These are atomic symbols, the ones we actually enter at the keyboard when composing a program in the language. Typical examples of terminals in a programming language's grammar are '>=', '`while`', and '`;`'.
- A finite set of ***nonterminal symbols*** (or just *nonterminals*). A nonterminal symbol represents a particular class of phrases in the language. Typical examples of nonterminals in a programming language's grammar are Program, Command, Expression, and Declaration.
- A ***start symbol***, which is one of the nonterminals. The start symbol represents the principal class of phrases in the language. Typically the start symbol in a programming language's grammar is Program.
- A finite set of ***production rules***. These define how phrases are composed from terminals and subphrases.

Grammars are usually written in the notation *BNF* (Backus–Naur Form). In BNF, a production rule is written in the form $N ::= \alpha$, where N is a nonterminal symbol, and where α is a (possibly empty) string of terminal and/or nonterminal symbols. Several production rules with a common nonterminal on their left-hand sides:

$N ::= \alpha$

$N ::= \beta$

...

may be grouped as:

$N ::= \alpha \mid \beta \mid \ldots$.

The BNF symbol '::=' is pronounced 'may consist of', and '|' is pronounced 'or alternatively'.

Example 1.3 Mini-Triangle syntax

Mini-Triangle is a toy programming language that will serve as a running example here and elsewhere. (It is a subset of Triangle, the language to be introduced in Section 1.4.)

Here is a trivial Mini-Triangle program:

```
! This is a comment. It continues to the end-of-line.
let
   const m ~ 7;
   var n: Integer
in
   begin
   n := 2 * m * m;
   putint(n)
   end
```

Here we present the context-free grammar of Mini-Triangle.

The terminal symbols of Mini-Triangle include:

```
begin     const   do      else    end     if
in        let     then    var     while
;         :       :=      ~       (       )
+         -       *       /       <       >       =
          \
```

(These are emboldened in the production rules below, for emphasis.)

The nonterminal symbols of Mini-Triangle include:

Program (start symbol)	
Command	single-Command
Expression	primary-Expression
V-name	
Declaration	single-Declaration
Type-denoter	
Operator	Identifier
Integer-Literal	

The production rules are:

Program	::=	single-Command	(1.1)
Command	::=	single-Command	(1.2a)
	|	Command **;** single-Command	(1.2b)
single-Command	::=	V-name **:=** Expression	(1.3a)
	|	Identifier **(** Expression **)**	(1.3b)
	|	**if** Expression **then** single-Command **else** single-Command	(1.3c)
	|	**while** Expression **do** single-Command	(1.3d)
	|	**let** Declaration **in** single-Command	(1.3e)
	|	**begin** Command **end**	(1.3f)

Expression	::=	primary-Expression	(1.4a)
	|	Expression Operator primary-Expression	(1.4b)
primary-Expression	::=	Integer-Literal	(1.5a)
	|	V-name	(1.5b)
	|	Operator primary-Expression	(1.5c)
	|	**(** Expression **)**	(1.5d)
V-name	::=	Identifier	(1.6)
Declaration	::=	single-Declaration	(1.7a)
	|	Declaration **;** single-Declaration	(1.7b)
single-Declaration	::=	**const** Identifier **~** Expression	(1.8a)
	|	**var** Identifier **:** Type-denoter	(1.8b)
Type-denoter	::=	Identifier	(1.9)
Operator	::=	**+** | **-** | ***** | **/** | **<** | **>** | **=** | ****	(1.10a–h)
Identifier	::=	Letter | Identifier Letter | Identifier Digit	(1.11a–c)
Integer-Literal	::=	Digit | Integer-Literal Digit	(1.12a–b)
Comment	::=	**!** Graphic* eol	(1.13)

Production rule (1.3f) tells us that a single-command may consist of the terminal symbol '`begin`', followed by a command, followed by the terminal symbol '`end`'.

Production rule (1.3a) tells us that a single-command may consist of a value-or-variable-name, followed by the terminal symbol '`:=`', followed by an expression.

A value-or-variable-name, represented by the nonterminal symbol V-name, is the name of a declared constant or variable. Production rule (1.6) tells us that a value-or-variable-name is just an identifier. (More complex value-or-variable-names can be written in full Triangle.)

Production rules (1.2a–b) tell us that a command may consist of a single-command alone, or alternatively it may consist of a command followed by the terminal symbol '`;`' followed by a single-command. In other words, a command consists of a sequence of one or more single-commands separated by semicolons.

In production rules (1.11a–c), (1.12a–b), and (1.13):

- eol stands for an end-of-line 'character';
- Letter stands for one of the lowercase letters '`a`', '`b`', ..., or '`z`';
- Digit stands for one of the digits '`0`', '`1`', ..., or '`9`';
- Graphic stands for a space or visible character.

The nonterminals Letter, Digit, and Graphic each represents a set of single characters. Specifying them formally is simple but tedious, for example:

```
Digit      ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

□

Each context-free grammar generates a language, which is a set of strings of terminal symbols. We define this language in terms of syntax trees and phrases. Consider a particular context-free grammar *G*.

A ***syntax tree*** of *G* is an ordered labeled tree such that: (a) the terminal nodes are labeled by terminal symbols; (b) the nonterminal nodes are labeled by nonterminal symbols; and (c) each nonterminal node labeled by *N* has children labeled by $X_1, \ldots, X_n$ (in order from left to right) such that $N ::= X_1 \ldots X_n$ is a production rule. More specifically, an *N-tree* of *G* is a syntax tree whose root node is labeled by *N*.

A ***phrase*** of *G* is a string of terminal symbols labeling the terminal nodes (taken from left to right) of a syntax tree. More specifically, an *N-phrase* of *G* is a string of terminal symbols labeling the terminal nodes of an *N*-tree.

A ***sentence*** of *G* is an *S*-phrase, where *S* is the start symbol. The ***language*** generated by *G* is the set of all sentences of *G*.

Example 1.4 Mini-Triangle syntax trees

Figures 1.1 through 1.3 show some Mini-Triangle syntax trees. Some of the nonterminal symbols have been abbreviated. The syntax trees of identifiers, operators and literals have been elided, being of little interest.

From the syntax tree of Figure 1.1 we can see that the following is an expression (formally, an Expression-phrase):

```
d + 10 * n
```

Note that this expression will be evaluated like '`(d+10)*n`', since Mini-Triangle's binary operators all have the same precedence. This is implicit in production rule (1.4b), and in the shape of the syntax tree.

From the syntax tree of Figure 1.2 we can see that the following is a single-command (formally, a single-Command-phrase):

```
while b do begin n := 0; b := false end
```

From the syntax tree of Figure 1.3 we can see that the following is a program (formally, a sentence or Program-phrase):

```
let var y: Integer in y := y + 1
```

□

A grammar like that of Example 1.3 has two roles:

- The grammar tells us, for each form of phrase, what its subphrases are. For example, a Mini-Triangle assignment command (1.3a) has two subphrases: a value-or-variable-

name and an expression. A Mini-Triangle if-command (1.3c) has three subphrases: an expression and two (sub)commands. The way in which a program is composed from phrases and subphrases is called its ***phrase structure***.

- The grammar also tells us the order in which the subphrases must be written, and the terminal symbols with which they must be delimited. For example, a Mini-Triangle assignment command (1.3a) consisting of a value-or-variable-name *V* and an expression *E* must be written in the form '*V* `:=` *E*'. A Mini-Triangle if-command (1.3c) consisting of an expression *E* and subcommands C_1 and C_2 must be written in the form '`if` *E* `then` C_1 `else` C_2'. Moreover, the grammar tells us that C_1 and C_2 must be *single*-commands (in order to avoid ambiguity).

Because of its concentration on concrete syntactic details, a grammar such as this specifies what we call the ***concrete syntax*** of the language. The concrete syntax is important to the programmer who needs to know exactly how to write syntactically well-formed programs.

But concrete syntax has no influence on the *semantics* of the programs. For example, whether the assignment command is written in the form '*V* `:=` *E*' or '*V* ← *E*' or '*E* → *V*' or '`set` *V* `=` *E*' or '`assign` *E* `to` *V*' does not affect how the command will be executed. These are all different in terms of concrete syntax, but all the same in terms of phrase structure.

When specifying semantics, it is convenient to concentrate on phrase structure alone. This is the point of ***abstract syntax***. A grammar specifying abstract syntax generates only a set of ***abstract syntax trees*** (ASTs). Each nonterminal node of an AST is labeled by a production rule, and it has exactly one subtree for each subphrase. The grammar does not generate sentences, for terminal symbols have no real role in abstract syntax.

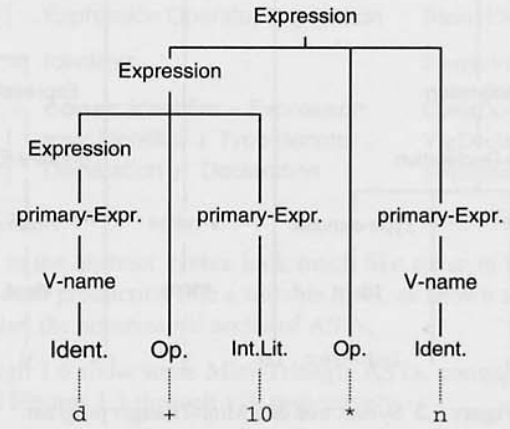

Figure 1.1 Syntax tree of a Mini-Triangle expression.

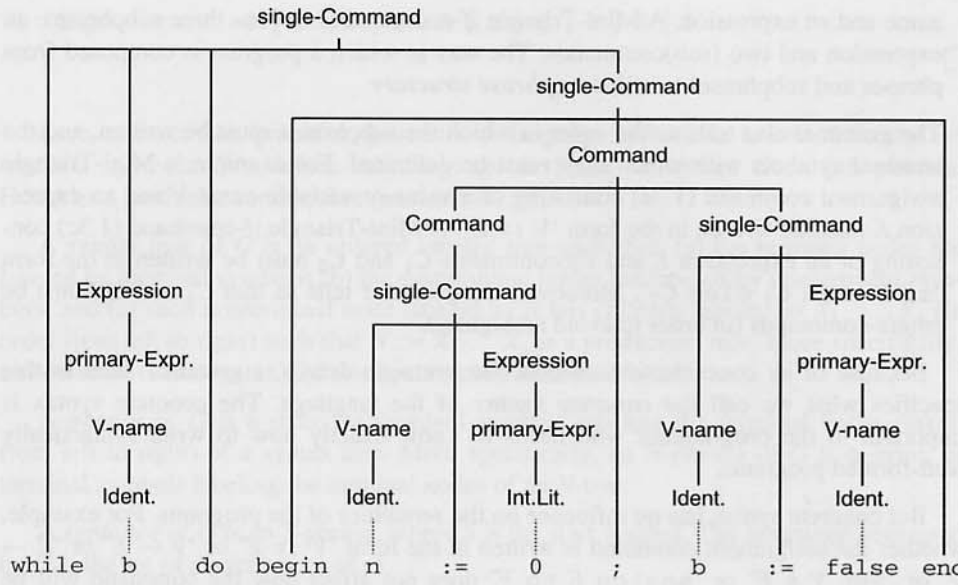

Figure 1.2 Syntax tree of a Mini-Triangle single-command.

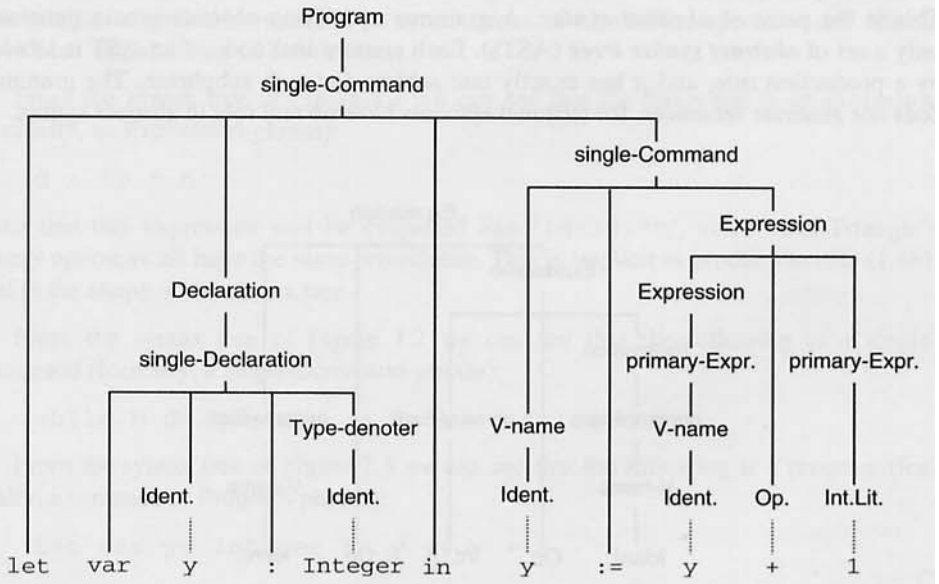

Figure 1.3 Syntax tree of a Mini-Triangle program.

Example 1.5 Mini-Triangle abstract syntax

Here we present a grammar specifying the abstract syntax of Mini-Triangle. This specify only the phrase structure of Mini-Triangle. Distinctions between commands single-commands, between declarations and single-declarations, and between expressions and primary-expressions, will be swept away.

The nonterminal symbols are:

Program (start symbol)
Command
Expression
V-name
Declaration
Type-denoter

The production rules are:

Program	::=	Command	Program	(1.1
Command	::=	V-name **:=** Expression	AssignCommand	(1.15
	|	Identifier **(** Expression **)**	CallCommand	(1.15
	|	Command **;** Command	SequentialCommand	(1.15
	|	**if** Expression **then** Command **else** Command	IfCommand	(1.15
	|	**while** Expression **do** Command	WhileCommand	(1.15
	|	**let** Declaration **in** Command	LetCommand	(1.15
Expression	::=	Integer-Literal	IntegerExpression	(1.16a
	|	V-name	VnameExpression	(1.16b
	|	Operator Expression	UnaryExpression	(1.16c
	|	Expression Operator Expression	BinaryExpression	(1.16d
V-name	::=	Identifier	SimpleVname	(1.17
Declaration	::=	**const** Identifier **~** Expression	ConstDeclaration	(1.18a
	|	**var** Identifier **:** Type-denoter	VarDeclaration	(1.18b
	|	Declaration **;** Declaration	SequentialDeclaration	(1.18c
Type-denoter	::=	Identifier	SimpleTypeDenoter	(1.19)

Production rules in the abstract syntax look much like those in the concrete syntax. In addition, we give each production rule a suitable label, as shown above right. We will use these labels to label the nonterminal nodes of ASTs.

Figures 1.4 through 1.6 show some Mini-Triangle ASTs, corresponding to the (concrete) syntax trees of Figures 1.1 through 1.3, respectively.

The AST of Figure 1.5 represents the following command:

```
while b do begin n := 0; b := false end
```

This AST's root node is labeled WhileCommand, signifying the fact that this is a while-command. The root node's second child is labeled SequentialCommand, signifying the fact that the body of the while-command is a sequential-command. Both children of the SequentialCommand node are labeled AssignCommand.

When we write down the above command, we need the symbols 'begin' and 'end' to bracket the subcommands 'n := 0' and 'b := false'. These brackets distinguish the above command from:

```
while b do n := 0; b := false
```

whose meaning is quite different. (See Exercise 1.5.) There is no trace of these brackets in the abstract syntax, nor in the AST of Figure 1.5. They are not needed because the AST structure itself represents the bracketing of the subcommands.

□

A program's AST represents its phrase structure explicitly. The AST is a convenient structure for specifying the program's contextual constraints and semantics. It is also a convenient representation for language processors such as compilers. For example, consider again the assignment command 'while *E* do *C*'. The meaning of this command can be specified in terms of the meanings of its subphrases *E* and *C*. The translation of this command into object code can be specified in terms of the translations of *E* and *C* into object code. The command is represented by an AST with root node labeled 'WhileCommand' and two subtrees representing *E* and *C*, so the compiler can easily access these subphrases.

In Chapter 3 we shall use ASTs extensively to discuss the internal phases of a compiler. In Chapter 4 we shall see how a compiler constructs an AST to represent the source program. In Chapter 5 we shall see how the AST is used to check that the program satisfies the contextual constraints. In Chapter 7 we shall see how to translate the program into object code.

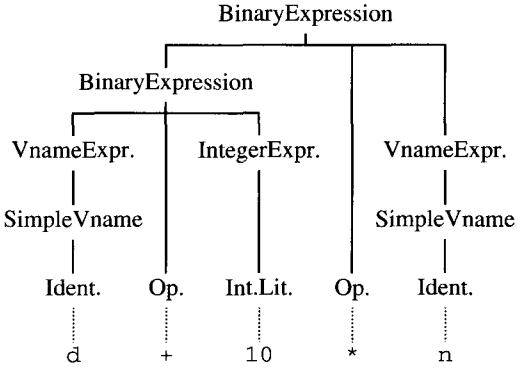

Figure 1.4 Abstract syntax tree of a Mini-Triangle expression.

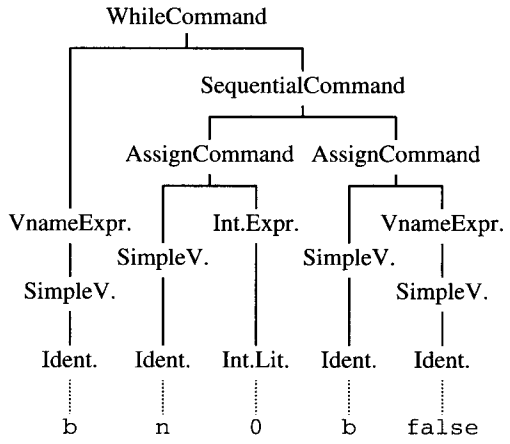

Figure 1.5 Abstract syntax tree of a Mini-Triangle command.

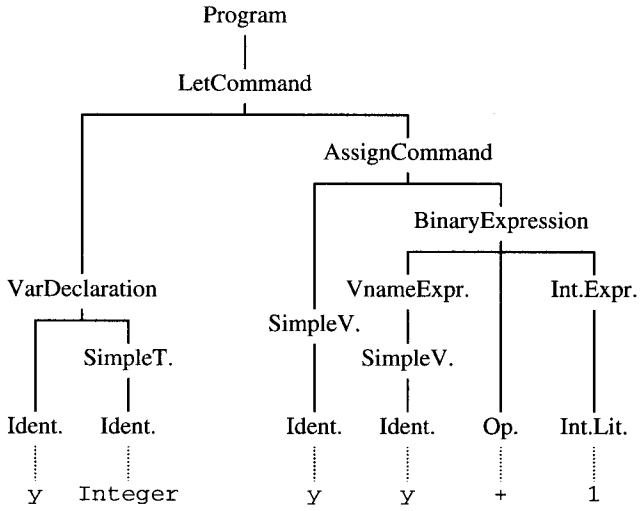

Figure 1.6 Abstract syntax tree of a Mini-Triangle program.

1.3.2 Contextual constraints

Contextual constraints are things like scope rules and type rules. They arise from the possibility that whether a phrase is well-formed or not may depend on its context.

Every programming language allows identifiers to be declared, and thereafter used in ways consistent with their declaration. For instance, an identifier declared as a

constant can be used as an operand in an expression; an identifier declared as a variable can be used either as an operand in an expression or on the left-hand side of an assignment; an identifier declared as a procedure can be used in a procedure call; and so on.

The occurrence of an identifier *I* at which it is declared is called a ***binding occurrence***. Any other occurrence of *I* (at which it is used) is called an ***applied occurrence***. At its binding occurrence, the identifier *I* is bound to some entity (such as a value, variable, or procedure). Each applied occurrence of *I* then denotes that entity. A programming language's rules about binding and applied occurrences of identifiers are called its ***scope rules***.

If the programming language permits the same identifier *I* to be declared in several places, we need to be careful about which binding occurrence of *I* corresponds to a given applied occurrence of *I*. The language exhibits ***static binding*** if this can be determined by a language processor without actually running the program; the language exhibits ***dynamic binding*** if this can be determined only at run-time. In fact, nearly all major programming languages do exhibit static binding; only a few languages (such as Lisp and Smalltalk) exhibit dynamic binding.

Example 1.6 Triangle scope rules

Mini-Triangle is too simplistic a language for static binding to be an issue, so we shall use Triangle itself for illustration. In the following Triangle program outline, binding occurrences of identifiers are underlined, and applied occurrences are italicized:

```
let
   const m ~ 2;
   var n: Integer;
   func f (i: Integer) : Integer ~
      i * m
in
   begin
   ...;
   n := f(n);          (1)
   ...
   end
```

Each applied occurrence of `m` denotes the constant value 2. Each applied occurrence of `n` denotes a particular variable. Each applied occurrence of `f` denotes a function that doubles its argument. Each applied occurrence of `i` denotes that function's argument. Each applied occurrence of `Integer` denotes the standard type *int*, whose values are integer numbers.

Triangle exhibits static binding. The function call at point (1) above doubles its argument. Imagine a call to `f` in a block where `m` is redeclared:

```
let
   const m ~ 3
```

```
in
    ... f(n) ...          (2)
```

The function call at point (2) also doubles its argument, because the applied occurrence of `m` inside the function `f` always denotes 2, regardless of what `m` denotes at the point of call.

In a language with dynamic binding, on the other hand, the applied occurrence of `m` would denote the value to which `m` was *most recently* bound. In such a language, the function call at (1) would double its argument, whereas the function call at (2) would *triple* its argument.

□

Every programming language has a universe of discourse, the elements of which we call ***values***. Usually these values are classified into ***types***. Each operation in the language has an associated ***type rule***, which tells us the expected operand type(s), and the type of the operation's result (if any). Any attempt to apply an operation to a wrongly-typed value is called a *type error*.

A programming language is ***statically typed*** if a language processor can detect all type errors without actually running the program; the language is ***dynamically typed*** if type errors cannot be detected until run-time.

Example 1.7 Mini-Triangle type rules

Mini-Triangle is statically typed. Consider the following program outline:

```
let
   var n: Integer
in
   begin
   ...
   while n > 0 do          (1)
      n := n - 1;          (2)
   ...
   end
```

The type rule of '`>`' is:

If both operands are of type *int*, then the result is of type *bool*.

Thus the expression '`n > 0`' at point (1) is indeed of type *bool*. Although we cannot tell in advance what particular values `n` will take, we know that such values will always be integers. Likewise, although we cannot tell in advance what particular values the expression '`n > 0`' will take, we know that such values will always be truth values.

The type rule of '`while` *E* `do` *C*' is:

E must be of type *bool*.

Thus the while-command starting at point (2) is indeed well-typed.

The type rule of '`-`' is:

If both operands are of type *int*, then the result is of type *int*.

Thus the expression '`n - 1`' at point (2) is indeed of type *int*.

The type rule of '*V* := *E*' is:

V and *E* must be of equivalent type.

Thus the assignment command at point (2) is indeed well-typed.

In a dynamically-typed language, each variable, parameter, etc., may take values of any type. For example, a given variable `x` might contain an integer or a truth value or a value of some other type. The same variable might even contain values of different types at different times. Thus we could not tell in advance what *type* of value `x` will contain, never mind what individual value. It follows that we could not tell in advance whether evaluating an expression such as '`x + 1`' will satisfy the type rule of '`+`'.

□

The fact that a programming language is statically typed implies the following:

- Every well-formed expression *E* has a unique type *T*, which can be inferred without actually evaluating *E*.
- Whenever *E* is evaluated, it will yield a value of type *T*. (Evaluation of *E* might fail due to overflow or some other run-time error, or it might diverge, but its evaluation will never fail due to a type error.)

In this book we shall generally assume that the source language exhibits static binding and is statically typed.

1.3.3 Semantics

Semantics is concerned with the meanings of programs, i.e., their behavior when run. Many notations have been devised for specifying semantics formally, but so far none has achieved widespread acceptance. Here we show how to specify the semantics of a programming language informally.

Our first task is to specify, *in general terms*, what will be the semantics of each class of phrase in the language. We may specify the semantics of commands, expressions, and declarations as follows:

- A command is executed to update variables. [It may also have the side effect of performing input–output.]
- An expression is evaluated to yield a value. [It may also have the side effect of updating variables.]

- A declaration is elaborated to produce bindings. [It may also have the side effect of allocating [and initializing] variables.]

In each case, the text in brackets is applicable only in certain languages.

Our remaining task is to systematically specify the semantics of each specific form of command, expression, declaration, and so on. Here we should follow the language's abstract syntax. In the abstract syntax there is one production rule for each form of phrase; in the semantics there should be one (or occasionally more than one) clause for each form of phrase.

Example 1.8 Mini-Triangle semantics

We specified the abstract syntax of Mini-Triangle in Example 1.5. Here we specify the semantics of Mini-Triangle, following the structure of the abstract syntax.

A *command C* is executed in order to update variables. (This includes input–output.)

The assignment-command 'V `:=` E' is executed as follows. The expression E is evaluated to yield a value v; then v is assigned to the value-or-variable-name V. (1.20a)

The call-command 'I `(` E `)`' is executed as follows. The expression E is evaluated to yield a value v; then the procedure bound to I is called with v as its argument. (1.20b)

The sequential command 'C_1 `;` C_2' is executed as follows. First C_1 is executed; then C_2 is executed. (1.20c)

The if-command '`if` E `then` C_1 `else` C_2' is executed as follows. The expression E is evaluated to yield a truth-value t; if t is true, C_1 is executed; if t is false, C_2 is executed. (1.20d)

The while-command '`while` E `do` C' is executed as follows. The expression E is evaluated to yield a truth-value t; if t is true, C is executed, and then the while-command is executed again; if t is false, execution of the while-command is completed. (1.20e)

The let-command '`let` D `in` C' is executed as follows. The declaration D is elaborated to produce bindings b; C is executed, in the environment of the let-command overlaid by the bindings b. The bindings b have no effect outside the let-command. (1.20f)

Note that clauses (1.20a–f) correspond respectively to production rules (1.15a–f) of the abstract syntax.

Note also that clauses (1.20d) and (1.20e) assume that evaluation of E will yield a truth-value. Likewise, clause (1.20a) assumes that evaluation of E will yield a value of the same type as V. These assumptions are justified if the command is well-typed.

An *expression E* is evaluated to yield a value.

The expression '*IL*' yields the value of the integer-literal *IL*. (1.21a)

The expression '*V*' yields the value of the value-or-variable-name *V*. (1.21b)

The unary expression '*O E*' yields the value obtained by applying unary operator *O* to the value yielded by the expression *E*. (1.21c)

The binary expression 'E_1 *O* E_2' yields the value obtained by applying binary operator *O* to the values yielded by the expressions E_1 and E_2. (1.21d)

Note that clauses (1.21a–d) correspond respectively to production rules (1.16a–d) of the abstract syntax.

Note also that expressions have no side effects in Mini-Triangle.

A *value-or-variable-name V* may be identified *either* to yield a value *or* to assign a value to a variable (as required by the context).

A simple value-or-variable-name *I* yields a value as follows. If *I* is bound to a value, it yields that value. If *I* is bound to a variable, it yields the value contained in that variable. (1.22)

A simple value-or-variable-name *I* is assigned a value *v* as follows. If *I* is bound to a variable, it updates that variable to contain *v*. (1.23)

A *declaration D* is elaborated to produce bindings; it may also have the side effect of allocating variables.

The constant declaration '`const` *I* ~ *E*' is elaborated by binding *I* to the value yielded by the expression *E*. (1.24a)

The variable declaration '`var` *I* : *T*' is elaborated by binding *I* to a newly allocated variable, whose initial value is undefined. The variable will be deallocated on exit from the block containing the variable declaration. (1.24b)

The sequential declaration 'D_1 ; D_2' is elaborated by elaborating D_1 followed by D_2, and combining the bindings they produce. D_2 is elaborated in the environment of the sequential declaration, overlaid by the bindings produced by D_1. (1.24c)

Note that clauses (1.24a–c) correspond respectively to production rules (1.18a–c) of the abstract syntax.

□

In Chapter 7 we shall use the semantics of Mini-Triangle to build a code generator for Mini-Triangle. In Chapter 8 we shall use the semantics to build a Mini-Triangle interpreter.

1.4 Case study: the programming language Triangle

In this book we shall use small examples – such as the toy language Mini-Triangle – to illustrate various implementation methods without getting lost in details. Nevertheless, it is also important to illustrate how these methods can be applied to realistic programming languages.

A major language like Pascal or Java is just *too* complicated for the purposes of an introductory textbook. Instead we shall use *Triangle*, a small but realistic programming language, as a case study. Triangle is a Pascal-like language, but generally simpler and more regular. Here we give a brief overview of Triangle. (A complete description may be found in Appendix B.)

Triangle commands are similar to Pascal's, but for simplicity there is only one conditional command and one iterative command. Unlike Pascal, Triangle has a let-command with local declarations.

Example 1.9 Triangle commands

The following illustrates the Triangle if-command and let-command:

```
if x > y then
   let const xcopy ~ x
   in
      begin x := y; y := xcopy end
else
```

Note the empty else-part. (It is actually a skip command.)

□

Triangle expressions are richer than Pascal's, but free of side effects. Conditional expressions, let-expressions with local declarations, and aggregates (record and array expressions) are all provided. A function body is just an expression. For simplicity, only three primitive types (denoted by the identifiers `Boolean`, `Char`, and `Integer`), and two forms of composite type (records and arrays), are provided. Unlike Pascal, Triangle is type-complete, i.e., no operations are arbitrarily restricted in the types of their operands. Thus values of any type may be passed as parameters, returned as function results, assigned, and compared using the binary operators '=' and '\='.

Example 1.10 Triangle expressions

The following illustrates a Triangle let-expression and if-expression:

```
let
   const taxable ~ if income > allowance
                   then income - allowance
                   else 0
in
   taxable / 4
```

The following illustrates Triangle record and array types and aggregates:

```
let
   type  Date ~ record
                   m: Integer, d: Integer
                end;
   const days ~ [31, 28, 31, 30, 31, 30,
                 31, 31, 30, 31, 30, 31];
   var   today: Date
in
   ...
   if today.d < days[today.m - 1]
   then {m ~ today.m, d ~ today.d + 1}
   else if today.m \= 12
   then {m ~ today.m + 1, d ~ 1}
   else {m ~ 1, d ~ 1}
   ...
   if today = {m ~ 2, d ~ 29} then ... else ...
```

Here `days` is declared to be a constant of type '`array 12 of Integer`', i.e., an ar
with elements 31, 28, 31, etc. The first if-expression yields a value of the record ty
`Date`, representing the day after `today`. The second if-expression illustrates rec
comparison.

Triangle declarations of different kinds may be mixed freely. Constant, variable, a
type declarations have been illustrated in Examples 1.9 and 1.10. A Triangle const
declaration may have any expression, of any type, on its right-hand side. T
expression must be evaluated at run-time, but thereafter the constant identifier's valu
fixed. (The Triangle constant declaration is more general than Pascal's, where the rig
hand side is restricted to be a constant.)

Triangle has procedure and function declarations. A procedure body is just a co
mand, which may be (but not necessarily) a let-command. Likewise, a function body
just an expression, which may be (but not necessarily) a let-expression. Functions
free of side effects.

Procedures and functions may have constant, variable, procedural, or functio
parameters. These have uniform semantics: in each case the formal-parameter identi

is simply bound to the corresponding argument, which is a value, variable, procedure, or function, respectively.

Example 1.11 Triangle procedures and functions

The following function and procedure implement operations on a type `Point`:

```
type Point ~ record
                x: Integer,
                y: Integer
             end;

func projection (pt: Point) : Point ~
      { x ~ pt.x, y ~ 0 - pt.y };

proc moveup (yshift: Integer, var pt: Point) ~
      pt.y := pt.y + yshift;

...
var p: Point; var q: Point;
...
moveup(3, var p);
q := projection(p)
```

□

Triangle has the usual variety of operators, standard functions, and standard procedures. These behave exactly like ordinary declared functions and procedures; unlike Pascal, they have no special type rules or parameter mechanisms. In particular, Triangle operators behave exactly like functions of one or two parameters.

Example 1.12 Triangle operators

The Triangle operator '`/\`' (logical conjunction) is, in effect, declared as follows:

```
func /\ (b1: Boolean, b2: Boolean) : Boolean ~
      if b1 then b2 else false
```

The expression '`a /\b`' is, in effect, a function call:

```
/\(a, b)
```

and the more complicated expression '`(n > 0) /\ (sum/n > 40)`' likewise:

```
/\(>(n, 0), >(/(sum, n), 40))
```

Note that the above declaration of `/\` implies that both operands of `/\` are evaluated before the function is called. (Some other programming languages allow *short-circuit evaluation*: the second operand of `/\` is skipped if the first operand evaluates to *false*.)

□

A complete informal specification of Triangle may be found in Appendix B. E section is devoted to a major construct, e.g., commands, expressions, or declarati Within the section there are subsections describing the intended *usage* of the constr its *syntax* (expressed in BNF), its *semantics* (and contextual constraints), and fin *examples*. Browse through Appendix B, attempting to fill the gaps in y understanding of Triangle left by the brief overview here. Appendix B is intende serve as a model of a carefully written informal specification of a programm language. Nevertheless, if you read carefully, you might well find loopholes!

1.5 Further reading

This book assumes that you are familiar with the basic concepts of high-level progr ming languages, including those summarized in Section 1.1. A detailed study of th concepts, using terminology consistent with this book, may be found in the compan textbook by Watt (1990). Some other good textbooks cover similar material, includ those by Ghezzi and Jazayeri (1987), Sethi (1988), and Tennent (1981).

A very brief review of syntax and semantics was given in Section 1.3. A much fu treatment may be found in the companion textbook by Watt (1991). The advantages disadvantages of formal and informal specification are discussed in detail, as are vari methods for formally specifying syntax, contextual constraints, and semantics. Ther an introduction to formal semantics. Formal specifications of the syntax and seman of Triangle are given as case studies.

A typical and recent example of a programming language specification is tha Java (Gosling *et al.* 1996). Java's syntax is specified formally in BNF, but its contex constraints and semantics are specified informally. This specification is by no me easy reading.

Exercises

1.1 In this chapter editors, compilers, and interpreters have been cited as kind language processor. Can you think of any other kinds of language processor

1.2* Recall Examples 1.1 and 1.2. Write a similar critical account of any other gramming system with which you are familiar.

1.3** Design an editor tailored to your favorite programming language.

(*Hints:* Think of the editing operations you perform most frequently on y programs. You probably delete or replace complete symbols more often t individual characters, and you probably delete or replace complete phrase

expressions, commands, declarations – rather than individual lines. You probably spend a lot of time on chores such as good layout. Also think of the common syntactic errors that might reasonably be detected immediately.)

1.4 According to the context-free grammar of Mini-Triangle in Example 1.3, which of the following are Mini-Triangle expressions?

(a) `true`

(b) `sin(x)`

(c) `-n`

(d) `m >= n`

(e) `m - n * 2`

Draw the syntax tree and AST of each one that *is* an expression.

Similarly, which of the following are Mini-Triangle commands?

(f) `n := n + 1`

(g) `halt`

(h) `put(m, n)`

(i) `if n > m then m := n`

(j) `while n > 0 do n := n-1`

Similarly, which of the following are Mini-Triangle declarations?

(k) `const pi ~ 3.1416`

(l) `const y ~ x+1`

(m) `var b: Boolean`

(n) `var m, n: Integer`

(o) `var y: Integer; const dpy ~ 365`

1.5 Draw the syntax tree and AST of the Mini-Triangle command:

```
while b do n := 0; b := false
```

cited at the end of Example 1.5. Compare with Figures 1.2 and 1.5.

1.6 According to the syntax and semantics of Mini-Triangle in Examples 1.3 and 1.8, what value is written by the following Mini-Triangle program? (The standard procedure `putint` writes its argument, an integer value.)

```
let
  const m ~ 2;
  const n ~ m + 1
in
  putint(m + n * 2)
```

(*Note:* Do not be misled by your knowledge of any other languages.)