CHAPTER THREE

Compilation

In this chapter we study the internal structure of compilers. A compiler's basic function is to translate a high-level source program to a low-level object program, but before doing so it must check that the source program is well-formed. So compilation is decomposed into three *phases*: syntactic analysis, contextual analysis, and code generation. In this chapter we study these phases and their relationships. We also examine some possible compiler designs, each design being characterized by the number of passes over the source program or its internal representation, and discuss the issues underlying the choice of compiler design.

In this chapter we restrict ourselves to a shallow exploration of compilation. We shall take a more detailed look at syntactic analysis, contextual analysis, and code generation in Chapters 4, 5, and 7, respectively.

3.1 Phases

Inside any compiler, the source program is subjected to several transformations before an object program is finally generated. These transformations are called ***phases***. The three principal phases of compilation are as follows:

- ***Syntactic analysis:*** The source program is parsed to check whether it conforms to the source language's syntax, and to determine its phrase structure.
- ***Contextual analysis:*** The parsed program is analyzed to check whether it conforms to the source language's contextual constraints.
- ***Code generation:*** The checked program is translated to an object program, in accordance with the semantics of the source and target languages.

The three phases of compilation correspond directly to the three parts of the source language's specification: its syntax, its contextual constraints, and its semantics.[1]

[1] Some compilers include a fourth phase, code optimization. Lexical analysis is sometimes treated as a distinct phase, but in this book we shall treat it as a sub-phase of syntactic analysis.

Between the phases we need to represent the source program in such a way as to reflect the analysis already done on it. A suitable choice of representation is an abstract syntax tree (AST). The AST explicitly represents the source program's phrase structure. Its subtrees will correspond to the phrases (commands, expressions, declarations, etc.) of the source program. Its leaf nodes will correspond to the identifiers, literals, and operators of the source program. All other terminal symbols in the source program can be discarded after syntactic analysis.

We can conveniently summarize the phases of a compiler by means of a data flow diagram.[2] Figure 3.1 shows the data flow diagram of a typical compiler. It shows the successive transformations effected by the three phases. It also shows that syntactic and contextual analysis may generate error reports, which will be transmitted to the programmer.

Let us now examine the three principal phases in more detail. We shall follow a tiny Triangle program through all the phases of compilation. The source program is shown in Figure 3.2, and the results of successive transformations in Figures 3.3, 3.4, and 3.7.

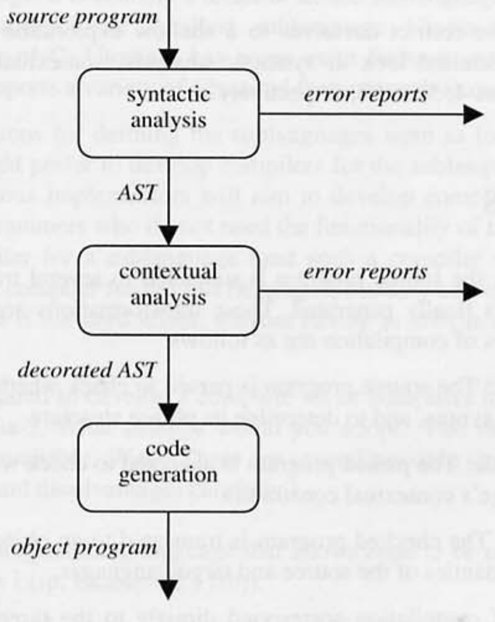

Figure 3.1 Data flow diagram for a typical compiler.

[2] A data flow diagram summarizes data flows and transformations in a system. An arrow represents a data flow, and is labeled by a description of the data. A rounded box represents a transformation, and is labeled accordingly.

In order to be concrete, we shall explain these transformations as implemented in the Triangle compiler that is our case study. It should be understood, however, that another Triangle compiler could implement the transformations in a different way. The main purpose of this section is to explain *what* transformations are performed, not *how* they are implemented. In Section 3.2.2 we shall emphasize this point by sketching an alternative Triangle compiler with a very different design, which nevertheless performs essentially the same processing on the source program.

3.1.1 Syntactic analysis

The purpose of syntactic analysis is to determine the source program's phrase structure. This process is called ***parsing***. It is an essential part of compilation because the subsequent phases (contextual analysis and code generation) depend on knowing how the program is composed from commands, expressions, declarations, and so on.

The source program is parsed to check whether it conforms to the source language's syntax, and to construct a suitable representation of its phrase structure. Here we assume that the chosen representation is an AST.

Example 3.1 Triangle AST

Syntactic analysis of the Triangle source program of Figure 3.2 yields the AST of Figure 3.3. As we shall be studying the compilation of this program in some detail, let us examine those parts of the AST that are numbered in Figure 3.3.

(1) The program is a let-command. It consists of a declaration ('`var n: Integer; var c: Char`' in the source program) and a subcommand ('`c := '&'; n := n+1`'). This is represented by an AST whose root node is labeled 'LetCommand', and whose subtrees represent the declaration and subcommand, respectively.

(2) This is a variable declaration. It consists of an identifier (`n`) and a type-denoter (`Integer`).

(3) This also is a variable declaration. It consists of an identifier (`c`) and a type-denoter (`Char`).

(4) This is a sequential command. It consists of two subcommands ('`c := '&'`' and '`n := n+1`').

(5) This is an assignment command. It consists of a value-or-variable-name on the left-hand side (`n`) and an expression on the right-hand side (`n+1`).

(6) This value-or-variable-name is just an identifier (`n`).

(7) This is an expression that applies an operator ('`+`') to two subexpressions.

(8) This expression is a value-or-variable-name (`n`).

(9) This expression is an integer-literal (`1`).

□

```
! This program is useless
! except for illustration.
let
    var n: Integer;
    var c: Char
in
    begin
    c := '&';
    n := n+1
    end
```

Figure 3.2 A Triangle source program.

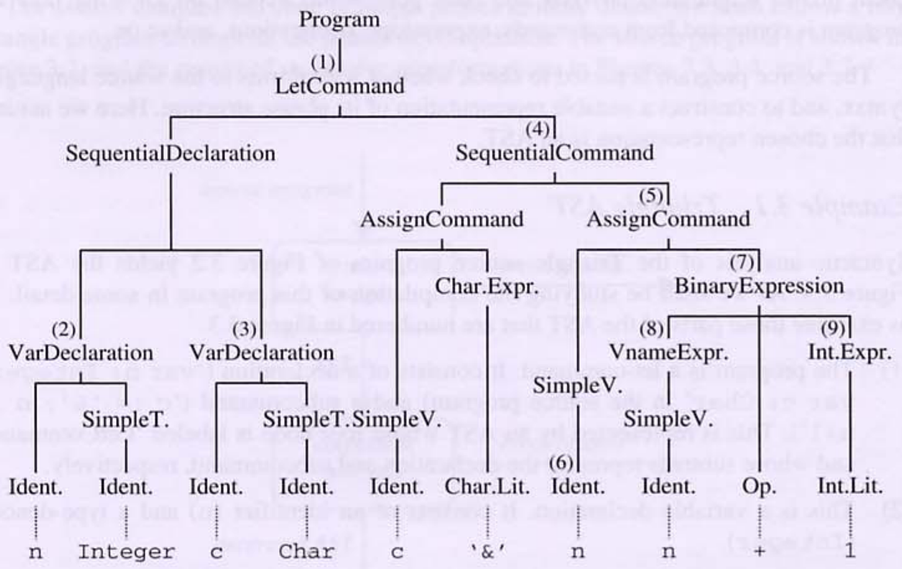

Figure 3.3 AST after syntactic analysis of the source program of Figure 3.2.

In general, the AST has terminal nodes that correspond to identifiers, literals, and operators in the source program, and subtrees that represent the phrases of the source program. Blank space and comments are not represented in the AST, because they contribute nothing to the source program's phrase structure. Punctuation and brackets also have no counterparts in the AST, because they serve only to separate and enclose phrases of the source program; once the source program has been parsed, they are no longer needed. For example, the '`begin`' and '`end`' brackets in Figure 3.2 serve only to enclose the sequential command '`c := '&'; n := n+1`', thus ensuring that the sequential command as a whole is taken as the body of the let-command. The AST's very structure represents this bracketing perfectly well.

If the source program contains syntactic errors, it has no proper phrase structure. In that case, syntactic analysis generates error reports instead of constructing an AST.

3.1.2 Contextual analysis

In contextual analysis the parsed program is further analyzed, to determine whether it conforms to the source language's contextual constraints:

- The source language's scope rules allow us, at compile-time, to associate each applied occurrence of an identifier (e.g., in an expression or command) with the corresponding declaration of that identifier, and to detect any undeclared identifiers. (Here we are assuming that the source language exhibits static binding.)
- The source language's type rules allow us, at compile-time, to infer the type of each expression and to detect any type errors. (Here we are assuming that the source language is statically typed.)

If the parsed program is represented by its AST, then contextual analysis will yield a ***decorated AST***. This is an AST enriched with information gathered during contextual analysis:

- As a result of applying the scope rules, each applied occurrence of an identifier is linked to the corresponding declaration. We show this diagrammatically by a dashed arrow.
- As a result of applying the type rules, each expression is decorated by its type *T*. We show this diagrammatically by marking the expression's root node ': *T*'.

Example 3.2 Triangle contextual analysis

Triangle exhibits static binding and is statically typed. Contextual analysis of the AST of Figure 3.3 yields the decorated AST of Figure 3.4.

The contextual analyzer checks the declarations as follows:

(2) It notes that identifier `n` is declared as a variable of type *int*.

(3) It notes that identifier `c` is declared as a variable of type *char*.

The contextual analyzer checks the second assignment command as follows:

(6) At this applied occurrence of identifier `n`, it finds the corresponding declaration at (2). It links this node to (2). From the declaration it infers that `n` is a variable of type *int*.

(8) Here, similarly, it infers that the expression `n` is of type *int*.

(9) This expression, being an integer-literal, is manifestly of type *int*.

(7) Since the operator '+' is of type $int \times int \rightarrow int$, it checks that the left and right subexpressions are of type *int*, and infers that the whole expression is of type *int*.

(5) It checks that the left-hand side of the assignment command is a variable, and that the right-hand side is an expression of equivalent type. Here both (6) and (7) are of type *int*, so the assignment command is indeed well-typed.

In this way the contextual analyzer verifies that the source program satisfies all the contextual constraints of Triangle.

□

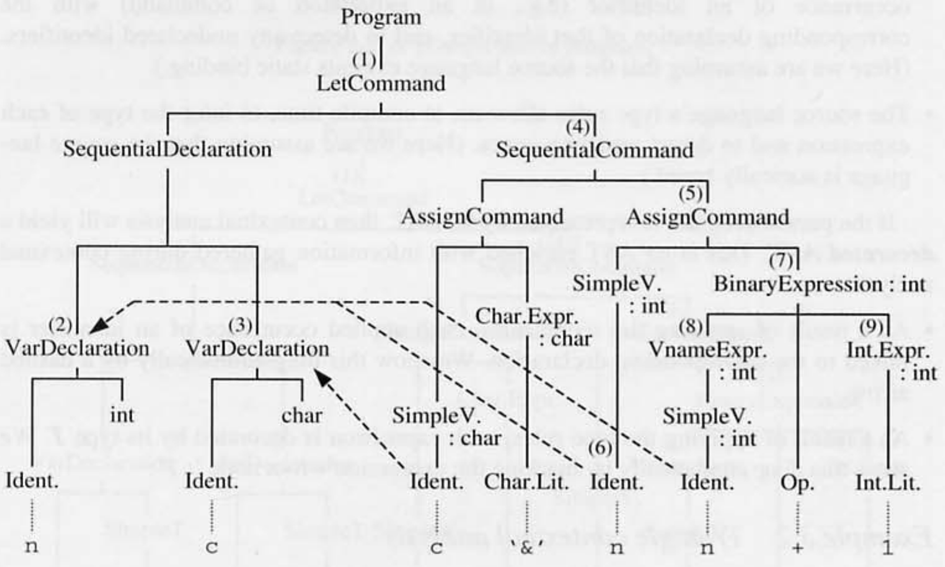

Figure 3.4 Decorated AST after contextual analysis of the AST of Figure 3.3.

If the source program does not satisfy the source language's contextual constraints, contextual analysis generates error reports.

Example 3.3 Detection of Triangle contextual errors

Figures 3.5 and 3.6 illustrate how contextual analysis will detect violations of scope rules and type rules. This particular Triangle program contains three contextual errors:

(1) The expression of this while-command is not of type *bool*.

(2) Identifier m is used but not declared.

(3) In this application of operator '>', which is of type *int* × *int* → *bool*, one subexpression has the wrong type.

□

```
let
   var n: Integer
in  ! ill-formed program
   while n/2 do
      m := 'n' > 1
```

Figure 3.5 An ill-formed Triangle source program.

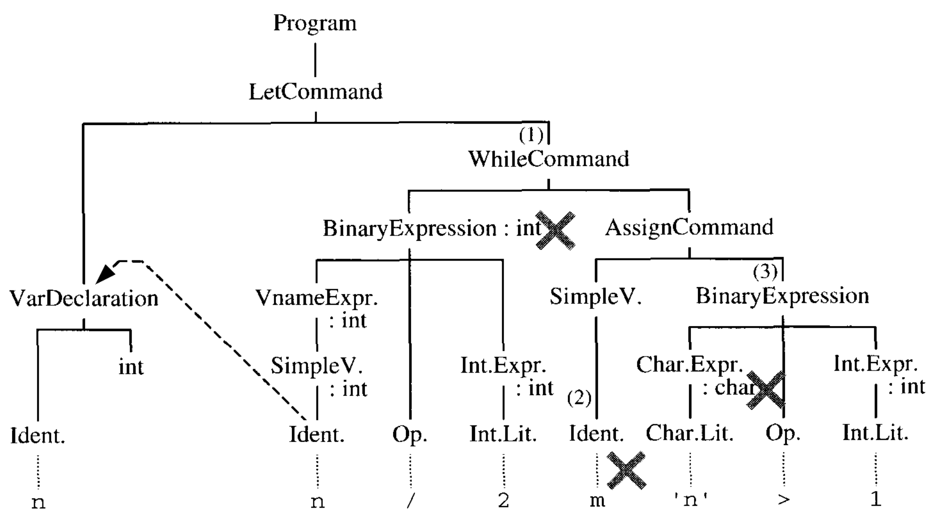

Figure 3.6 Discovering errors during contextual analysis of the Triangle program of Figure 3.5.

3.1.3 Code generation

After syntactic and contextual analysis, the source program has been thoroughly checked and is known to be well-formed. Code generation is the final translation of the checked program to an object program, in accordance with the source and target languages' semantics.

A pervasive issue in code generation is the treatment of identifiers that are declared and/or used in the source program. In semantic terms, a declaration *binds* an identifier to some sort of entity. For example:

- A constant declaration such as '`const m ~ 7`' binds the identifier `m` to the value 7. The code generator must then replace each applied occurrence of `m` by the value 7.
- A variable declaration such as '`var b: Boolean`' binds the identifier `b` to some address (storage cell), which is decided by the code generator itself. The code generator must then replace each applied occurrence of `b` by the address to which it is bound.

A rather different issue for the compiler designer is the exact nature of the target language: should the compiler generate machine code or the assembly language of the target machine? Actually, the choice has only minor influence on the structure of the compiler, and we shall not pursue the issue in this book. When presenting examples of object code, however, we always write instructions mnemonically (as in Figure 3.7), since this is considerably more readable than the equivalent binary machine code.

```
PUSH  2
LOADL 38
STORE 1[SB]
LOAD  0[SB]
LOADL 1
CALL  add
STORE 0[SB]
POP   2
HALT
```

Figure 3.7 Object program after code generation from Figure 3.4.

Example 3.4 TAM code generation

Code generation from the decorated AST of Figure 3.4 yields the TAM object program of Figure 3.7.

The code generator processes the declarations as follows:

(2) It allocates an address for the variable `n`, say `0[SB]`. It stores that address at node (2), for later retrieval.[3]

(3) It similarly allocates an address for the variable `c`, say `1[SB]`. It stores that address at node (3), for later retrieval.

The code generator processes the second assignment command as follows:

(8) By following the link to the declaration of `n`, it retrieves this variable's address, namely `0[SB]`. Then it generates the instruction '`LOAD 0[SB]`'. (When executed, this instruction will fetch the current value of that variable.)

(9) It generates the instruction '`LOADL 1`'. (When executed, this instruction will fetch the literal value 1.)

[3] Here '`0[SB]`' means address 0 relative to the base register SB – but you will be able to follow this example without knowing TAM's addressing mechanism.

(7) It generates the instruction '`CALL` *add*'. (When executed, this instruction will add the two previously-fetched values.)

(5) By following the link to the declaration of `n`, it retrieves this variable's address, namely `0[SB]`. Then it generates the instruction '`STORE 0[SB]`'. (When executed, this instruction will store the previously-computed value in that variable.)

In this way the code generator translates the whole program into object code.

□

3.2 Passes

In the previous section we examined the principal phases of compilation, and the flow of data between them. In this section we go on to examine and compare alternative compiler designs.

In designing a compiler, we wish to decompose it into modules, in such a way that each module is responsible for a particular phase. In practice there are several ways of doing so. The design of the compiler affects its modularity, its time and space requirements, and the number of passes over the program being compiled.

A ***pass*** is a complete traversal of the source program, or a complete traversal of an internal representation of the source program (such as an AST). A *one-pass* compiler makes a single traversal of the source program; a *multi-pass* compiler makes several traversals.

In practice, the design of a compiler is inextricably linked to the number of passes it makes. In this section we contrast multi-pass and one-pass compilation, and summarize the advantages and disadvantages of each.

3.2.1 Multi-pass compilation

One possible compiler design is shown by the structure diagram[4] of Figure 3.8.

The compiler consists of a top-level driver module together with three lower-level modules, the syntactic analyzer, the contextual analyzer, and the code generator. First, the compiler driver calls the syntactic analyzer, which reads the source program, parses it, and constructs a complete AST. Next, the compiler driver calls the contextual

[4] A structure diagram summarizes the modules and module dependencies in a system. The higher-level modules are those near the top of the structure diagram. A connecting line represents a dependency of a higher-level module on a lower-level module. This dependency consists of the higher-level module using the services (e.g., types or methods) provided by the lower-level module.

analyzer, which traverses the AST, checks it, and decorates it. Finally, the compiler driver calls the code generator, which traverses the decorated AST and generates an object program.

In general, a compiler with this design makes at least three passes over the program being compiled. The syntactic analyzer takes one pass, and the contextual analyzer and code generator take at least one pass each.

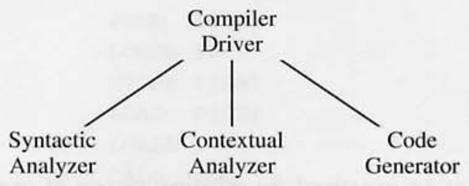

Figure 3.8 Structure diagram for a typical multi-pass compiler.

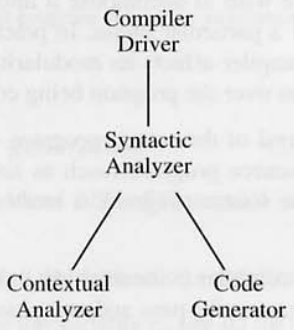

Figure 3.9 Structure diagram for a typical one-pass compiler.

3.2.2 One-pass compilation

An alternative compiler design is for the syntactic analyzer to control the other phases of compilation, as shown in Figure 3.9. A compiler with this design makes a single pass over the source program.

Contextual analysis and code generation are performed 'on the fly' during syntactic analysis. As soon as a phrase (e.g., expression, command, or declaration) has been parsed, the syntactic analyzer calls the contextual analyzer to perform any necessary checks. It also calls the code generator to generate any object code. Then the syntactic analyzer continues parsing the source program.

Example 3.5 One-pass compilation

A one-pass Triangle compiler would work as follows. Consider the following Triangle source program:

```
! This program is useless
! except for illustration.
let
   var n: Integer(1);
   var c: Char(2)
in
   begin
   c(3) := '&'(4)(5);
   n(6) := n+1(7)(8)
   end
```

This is identical to the source program of Figure 3.2, but some of the key points in the program have been numbered for easy reference. At these points the following actions are taken:

(1) After parsing the variable declaration 'var n: Integer', the syntactic analyzer calls the contextual analyzer to record the fact (in a table) that identifier n is declared to be a variable of type *int*. It then calls the code generator to allocate and record an address for this variable, say 0[SB].

(2) After parsing the variable declaration 'var c: Char', the syntactic analyzer similarly calls the contextual analyzer to record the fact that identifier c is declared to be a variable of type *char*. It then calls the code generator to allocate and record an address for this variable, say 1[SB].

(3) After parsing the value-or-variable-name c, the syntactic analyzer infers (by calling the contextual analyzer) that it is a variable of type *char*. It then calls the code generator to retrieve its address, 1[SB].

(4) After parsing the expression '&', the syntactic analyzer infers that it is of type *char*. It then calls the code generator to generate instruction 'LOADL 38'.

(5) After parsing the assignment command 'c := '&'', the syntactic analyzer calls the contextual analyzer to check type compatibility. It then calls the code generator to generate instruction 'STORE 1[SB]', using the address retrieved at point (3).

(6) After parsing the value-or-variable-name n, the syntactic analyzer infers (by calling the contextual analyzer) that it is a variable of type *int*. It then calls the code generator to retrieve the variable's address, 0[SB].

(7) While parsing the expression n+1, the syntactic analyzer infers (by calling the contextual analyzer) that the subexpression n is of type *int*, that the operator '+' is of type *int* × *int* → *int*, that the subexpression 1 is of type *int*, and hence that the whole expression is of type *int*. It calls the code generator to generate instructions 'LOAD 0[SB]', 'LOADL 1', and 'CALL *add*'.

(8) After parsing the assignment command '`n := n+1`', the syntactic analyzer calls the contextual analyzer to check type compatibility. It then calls the code generator to generate instruction '`STORE 0[SB]`'.

□

3.2.3 Compiler design issues

The choice between one-pass and multi-pass compilation is one of the first and most important design decisions for the compiler writer. It is not an easy decision, for both designs have important advantages and disadvantages. We summarize the main issues here.

- *Speed* is an issue where a one-pass compiler wins. Construction and subsequent traversals of the AST (or other internal program representation) is a modest time overhead in any multi-pass compiler. If the AST is stored on disk, however, the input–output overhead is likely to be large, even dominating compilation time.

- *Space* might also seem to favor a one-pass compiler. A multi-pass compiler must find memory to store the AST. But the situation is not really so clear-cut. In a multi-pass compiler, only one of the principal modules (syntactic analyzer, contextual analyzer, and code generator) is active at a time, so their code can share memory. In a one-pass compiler, all these modules are active throughout compile-time, so they must be co-resident in memory. As a result, the code of a one-pass compiler occupies more memory than the code of a multi-pass compiler.

  Of course, a very large source program will give rise to a very large AST, perhaps occupying more memory than the compiler itself. Fortunately, modern programming languages allow larger programs to be decomposed into compilation units, which are compiled separately; and individual compilation units tend to be moderately-sized. (See also Exercises 3.5 and 3.6.)

- *Modularity* favors the multi-pass compiler. In a one-pass compiler, the syntactic analyzer not only parses the source program but also coordinates the contextual analyzer and code generator. That is to say, it calls these modules, and maintains the data passed to and from them. In practice, the coordinating code may swamp the syntactic analysis code. In a multi-pass compiler, each module (including the syntactic analyzer) is responsible for a single function.

- *Flexibility* is an issue that favors the multi-pass compiler. Once the syntactic analyzer has constructed the AST, the contextual analyzer and code generator can traverse the AST in any convenient order. In particular, the code generator can translate phrases out of order, and sometimes this allows it to generate more efficient object code. A one-pass compiler is restricted to check and translate the phrases in exactly the order in which they appear in the source program.

- Semantics-preserving *transformations* of the source program or object program are performed by some compilers in order to make the object code as efficient as possible. (These are the so-called 'optimizing' compilers.) Such transformations generally require analysis of the whole program prior to code generation, so they force a multi-pass design on the compiler.

- *Source language properties* might restrict the choice of compiler design. A source program can be compiled in one pass only if every phrase (e.g., command or expression) can be compiled using only information obtained from the preceding part of the source program. This requirement usually boils down to whether identifiers must be declared before use. If they must be declared before use (as in Pascal, Ada, and Triangle), then one-pass compilation is possible in principle. If identifiers need not be declared before use (as in Java and ML), then multi-pass compilation is required.

Example 3.6 Pascal compiler design

In Pascal, the usual rule is that identifiers must be declared before use. Thus an applied occurrence of an identifier can be compiled in the sure knowledge that the identifier's declaration has already been processed (or is missing altogether).

Consider the following Pascal block:

```
var n: Integer;

procedure inc;
   begin
   n := n+1
   end;

begin
n := 0; inc
end
```

When a Pascal one-pass compiler encounters the command '`n := n+1`', it has already processed the declaration of `n`. It can therefore retrieve the type and address of the variable, and subject the command to contextual analysis and code generation.

Suppose, instead, that the declaration of `n` *follows* the procedure. When the Pascal one-pass compiler encounters the command '`n := n+1`', it has not yet encountered the declaration of `n`. So it cannot subject the command to contextual analysis and code generation. Fortunately, the compiler is not obliged to do so: it can safely generate an error report that the declaration of `n` is either misplaced or missing altogether.

□

Example 3.7 Java compiler design

The situation is different in Java, in which variable or method declarations need not be in any particular order. The following Java class is perfectly well-formed:

```
class Example {
    void inc() { n = n + 1; }
    int n;
    void use() { n = 0; inc(); }
}
```

The command `n = n + 1;` cannot be subjected to contextual analysis and code generation until the variable declaration `int n;` has been processed. A Java compiler must therefore process variable declarations in one pass, and the commands and expressions inside a method body in a later pass.

□

3.3 Case study: the Triangle compiler

In Section 2.7 we introduced our case study, the Triangle language processor. This consists of a compiler, an interpreter, and a disassembler. In this section we look more closely at the Triangle compiler, explaining its design.

The Triangle compiler has the usual three phases of syntactic analysis, contextual analysis, and code generation, as shown in the data flow diagram of Figure 3.1. It has three passes, having the outline structure shown in Figure 3.8. The syntactic analyzer, contextual analyzer, and code generator modules take one pass each, communicating via an AST that represents the source program. This was illustrated in Examples 3.1, 3.2, and 3.4.

Omitting minor details, the compiler driver looks like this:

```
public class Compiler {
    public static void compileProgram (...) {
        Parser parser = new Parser(...);
        Checker checker = new Checker(...);
        Encoder generator = new Encoder(...);
        // Call the syntactic analyzer to parse the source program and
        // construct theAST...
        Program theAST = parser.parse();
        // Call the contextual analyzer to check and decorate theAST...
        checker.check(theAST);
        // Call the code generator to translate theAST to an object program...
        generator.encode(theAST);
    }
    public static void main (String[] args) {
        ...
        compileProgram(...);
    }
}
```

A one-pass Triangle compiler would have been perfectly feasible, so the choice of a three-pass design needs to be justified. The Triangle compiler is intended primarily for educational purposes, so simplicity and clarity are paramount. Efficiency is a secondary consideration; in any case, efficiency arguments for a one-pass compiler are inconclusive, as we saw in Section 3.2.3. So the Triangle compiler was designed to be as modular as possible, allowing the different phases to be studied independently of one another.

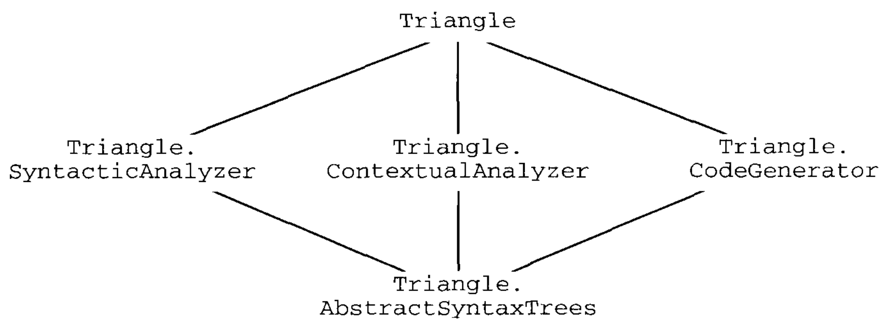

Figure 3.10 Structure diagram for the Triangle compiler.

A detailed structure diagram of the Triangle compiler is given in Figure 3.10, showing the main classes and packages. Here are brief explanations of the packages and the main classes they contain:

- The `Triangle.AbstractSyntaxTrees` package contains classes defining the AST data structure. There is a class for each Triangle construct, e.g., `AssignCommand`, `IfCommand`, `BinaryExpression`, `ConstDeclaration`, `VarDeclaration`, etc. Each class contains a constructor for building the AST for that construct, and a visitor method used by the contextual analyzer and the code generator to traverse the AST. The other parts of the compiler are allowed to manipulate the fields of the AST objects directly.

- The `Triangle.SyntacticAnalyzer` package contains the `Parser` class (and some classes of no concern here). The parser parses the source program, and constructs the AST. It generates an error report if it detects a syntactic error.

- The `Triangle.ContextualAnalyzer` package contains the `Checker` class. The checker traverses the AST, links applied occurrences of identifiers to the corresponding declarations, infers the types of all expressions, and performs all necessary type checks. It decorates the AST with these types. It generates an error report if it detects a contextual error.

- The `Triangle.CodeGenerator` package contains the `Encoder` class. The encoder traverses the decorated AST, allocates addresses to variables, and generates TAM object code.
- The `Triangle` package contains the `Compiler` class. The compiler simply drives the three phases of the compilation, as described above.

Diagrams describing the complete design of the Triangle compiler are given in Appendix D. In later chapters we shall continue this case study by looking inside the individual packages and their classes. Detailed documentation about the contents of each class can also be found at our Web site (see Preface, page).

3.4 Further reading

The textbook by Aho *et al.* (1985) offers a comprehensive treatment of all aspects of compilation. Chapter 1 discusses compiler designs in general; Chapter 2 presents a complete example of one-pass compilation; Chapter 11 discusses compiler design issues; and Chapter 12 looks at several case studies of real compilers.

This book concentrates on multi-pass compilation, in the interests of clarity and modularity. Other authors, such as Hoare (1973), have stressed the advantages of one-pass compilation. Welsh and McKeag (1980) devote a large part of their textbook to one-pass compilation. As a case study they develop a complete compiler for a subset of Pascal. Welsh and Hay (1986) is a complete one-pass Pascal compiler, together with an interpreter. That book is a fine example of literate programming.

The idea of using abstract syntax as a basis for compilation seems to be due to McCarthy (1963). Despite the attractions of this idea, it has received scant attention in most compiler textbooks.

Many internal representations other than ASTs are possible, of course. Lower-level internal representations tend to be more convenient for code generation to real machine code. A prominent example of this is the Gnu compiler kit, which uses a machine-independent but low-level intermediate language RTL. We can then construct 'front-ends' translating a variety of high-level languages to RTL, and 'back-ends' translating RTL to a variety of target machine codes. (See Exercise 2.5.) If we have *m* front-ends and *n* back-ends, we can combine these *m*+*n* components to make *mn* distinct compilers. This is a major saving of effort.

Exercises

3.1 In Examples 3.2 and 3.4, the first assignment command '`c := '&'`' was ignored. Describe how this command would have been subjected to contextual analysis and code generation.

3.2 The Mini-Triangle source program below left would be compiled to the object program below right:

```
let
  const m ~ 7;
  var x: Integer           PUSH   1
in
  x := m * x               LOADL  7
                           LOAD   0[SB]
                           CALL   mult
                           STORE  0[SB]
                           POP    1
                           HALT
```

Describe the compilation in the same manner as Examples 3.1, 3.2, and 3.4. (You may ignore the generation of the `PUSH`, and `POP` instructions.)

3.3 The Mini-Triangle source program below contains several contextual errors:

```
let
  var a: Logical;
  var b: Boolean;
  var i: Integer
in
  if i then b := i = 0 else b := yes
```

In the same manner as Example 3.3, show how contextual analysis will detect these errors.

3.4* Choose a compiler with which you are familiar. Find out and describe its phases and its pass structure. Draw a data flow diagram (like Figure 3.1) and a structure diagram (like Figure 3.8 or Figure 3.9).

3.5 Consider a source language, like Fortran or C, in which the source program consists of one or more distinct subprograms – a main program plus some procedures or functions. Design a compiler that uses ASTs, but (assuming that individual subprograms are moderately-sized) requires only a moderate amount of memory for ASTs.

3.6* The Triangle compiler would be unable to translate a very large source program, because of the memory required to store its AST. Consider the following proposal to redesign the compiler to improve its handling of very large source programs.

One procedure/function body is to be (completely) compiled at a time. Whenever the compiler has parsed a procedure/function declaration and constructed its AST, it breaks off to perform contextual analysis and code generation on the procedure/function body's AST, and then prunes the AST leaving a stub in place of the procedure/function body. Then the compiler resumes parsing the source program.

Would such a restructuring of the compiler be feasible? If *no*, explain why not. If *yes*, work through the following small source program, showing the steps that would be taken by the compiler, along the same lines as Example 3.5:

```
let
   var n: Integer;
   proc inc () ~
      n := n + 1
in
   begin n := 0; inc() end
```