CHAPTER NINE

Conclusion

The subject of this book is programming language implementation. As we study this subject, we should remember that implementation is only part of the *programming language life cycle*, where it takes its place along with programming language design and specification. In Section 9.1 we discuss the programming language life cycle, emphasizing the interactions among design, specification, and implementation. We also distinguish between cheap, low-quality implementations (*prototypes*) and high-quality implementations.

This naturally leads to a discussion of quality issues in implementation. In previous chapters we have concentrated on introducing the basic methods of compilation and interpretation, and relating these to the source language's specification. Correctness of the implementation, with respect to the language specification, has been our primary consideration. Quality of the implementation is a secondary consideration, although still very important. The key quality issues are *error reporting* and *efficiency*. Sections 9.2 and 9.3 discuss these issues, as they arise both at compile-time and at run-time.

9.1 The programming language life cycle

Every programming language has a life cycle, which has some similarities to the well-known software life cycle. The language is *designed* to meet some requirement. A formal or informal *specification* of the language is written in order to communicate the design to other people. The language is then implemented by means of language processors. Initially, a *prototype* implementation might be developed so that programmers can try out the language quickly. Later, high-quality (industrial-strength) *compilers* will be developed so that realistic application programming can be undertaken.

As the term suggests, the programming language life cycle is an iterative process. Language design is a highly creative and challenging endeavor, and no designer makes a perfect job at the first attempt. The experience of specifying or implementing a new language tends to expose irregularities in the design. Implementors and programmers might discover flaws in the specification, such as ambiguity, incompleteness, or inconsistency. They might also discover unpleasant features of the language itself, features that make the language unduly hard to implement efficiently, or unsatisfactory for programming.

In any case, the language might have to be redesigned, respecified, and reimplemented, perhaps several times. This is bound to be costly, i.e., time-consuming and expensive. It is necessary, therefore, to plan the life cycle in order to minimize costs.

Figure 9.1 illustrates a life cycle model that has much to recommend it. Design is immediately followed by specification. (This is needed to communicate the design to implementors and programmers.) Development of a prototype follows, and development of compilers follows that. Specification, prototyping, and compiler development are successively more costly, so it makes sense to order them in this way. The designer gets the fastest possible feedback, and costly compiler development is deferred until the language design has more or less stabilized.

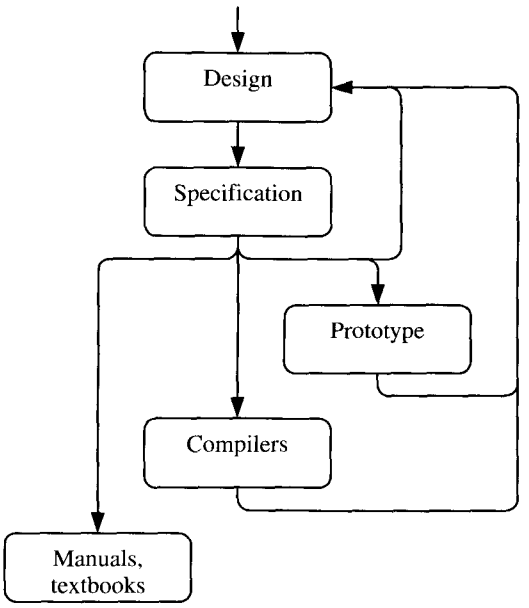

Figure 9.1 A programming language life cycle model.

9.1.1 Design

The essence of programming language design is that the designer selects concepts and decides how to combine them. This selection is, of course, determined largely by the intended use of the language. A variety of concepts have found their way into programming languages: basic concepts such as values and types, storage, bindings, and abstraction; and more advanced concepts such as encapsulation, polymorphism, exceptions, and concurrency. A single language that supports all these concepts is likely to be very large and complex indeed (and its implementations will be large, complex, and costly). Therefore a judicious selection of concepts is necessary.

The designer should strive for simplicity and regularity. Simplicity implies that the language should support only the concepts essential to the applications for which the language is intended. Regularity implies that the language should combine these concepts in a systematic way, avoiding restrictions that might surprise programmers or make their task more difficult. (Language irregularities also tend to make implementation more difficult.)

A number of principles have been discovered that provide useful guidance to the designer:

- The *type completeness principle* suggests that no operation should be arbitrarily restricted in the types of its operands. For instance, operations like assignment and parameter passing should, ideally, be applicable to all types in the language.
- The *abstraction principle* suggests that, for each program phrase that specifies some kind of computation, there should be a way of abstracting that phrase and parameterizing it with respect to the entities on which it operates. For instance, it should be possible to abstract any expression to make a function, or (in an imperative language) to abstract any command to make a procedure.
- The *correspondence principle* suggests that, for each form of declaration, there should be a corresponding parameter mechanism. For instance, it should be possible to take a block with a constant definition and transform it into a procedure (or function) with a constant parameter.

These are principles, not dogma. Designers often have to make compromises (for example to avoid constructions that would be unduly difficult to implement). But at least the principles help the designer to make the hard design decisions rationally and fully conscious of their consequences.

The main purpose of this brief discussion has been to give an insight into why language design is so difficult. Pointers to more extensive discussions of language design may be found in Section 9.4.

9.1.2 Specification

A new language design exists only in the mind of the designer until it is communicated to other people. For this purpose a precise specification of the language's syntax and semantics must be written. The specification may be informal, formal, or (most commonly) a hybrid.

Nearly all language designers specify their syntax formally, using BNF, EBNF or syntax diagrams. These formalisms are widely understood and easy to use. Some older languages, such as Fortran and Cobol, did not have their syntax formalized, and it is noteworthy that their syntax is clumsy and irregular. Formal specification of syntax tends to encourage syntactic simplicity and regularity, as illustrated by Algol (the language for which BNF was invented) and its many successors. For example, the earlier versions of Fortran had several different classes of expression, permissible in

different contexts (assignment, array indexing, loop parameters); whereas Algol from the start had just one class of expression, permissible in all contexts.

Similarly, formal specification of semantics tends to encourage semantic simplicity and regularity. Unfortunately, few language designers yet attempt this. Semantic formalisms are much more difficult to master than BNF. Even then, writing a semantic specification of a real programming language (as opposed to a toy language) is a substantial task. Worst of all, the designer has to specify, not a stable well-understood language, but one that is gradually being designed and redesigned. Most semantic formalisms are ill-suited to meet the language designer's requirements, so it is not surprising that almost all designers content themselves with writing informal semantic specifications.

The advantages of formality and the disadvantages of informality should not be underestimated, however. Informal specifications have a strong tendency to be inconsistent or incomplete or both. Such specification errors lead to confusion when the language designer seeks feedback from colleagues, when the new language is implemented, and when programmers try to learn the new language. Of course, with sufficient investment of effort, most specification errors can be detected and corrected, but an informal specification will probably never be completely error-free. The same amount of effort could well produce a formal specification that is at least guaranteed to be precise.

The very act of writing a specification tends to focus the designer's mind on aspects of the design that are incomplete or inconsistent. Thus the specification exercise provides valuable and timely feedback to the designer. Once the design is completed, the specification (whether formal or informal) will be used to guide subsequent implementations of the new language.

9.1.3 Prototypes

A prototype is a cheap low-quality implementation of a new programming language. Development of a prototype helps to highlight any features of the language that are hard to implement. The prototype also gives programmers an early opportunity to try out the language. Thus the language designer gains further valuable feedback. Moreover, since a prototype can be developed relatively quickly, the feedback is timely enough to make a language revision feasible. A prototype might lack speed and good error reporting; but these qualities are deliberately sacrificed for the sake of rapid implementation.

For a suitable programming language, an interpreter might well be a useful prototype. An interpreter is very much easier and quicker to implement than a compiler for the same language. The drawback of an interpreter is that an interpreted program will run perhaps 100 times more slowly than an equivalent machine-code program. Programmers will quickly tire of this enormous inefficiency, once they pass the stage of trying out the language and start to use it to build real applications.

A more durable form of prototype is an interpretive compiler. This consists of a translator from the programming language to some suitable abstract machine code,

together with an interpreter for the abstract machine. The interpreted object program will run 'only' about 10 times more slowly than a machine-code object program. Developing the compiler and interpreter together is still much less costly than developing a compiler that translates the programming language to real machine code. Indeed, a suitable abstract machine might be available 'off the shelf', saving the cost of writing the interpreter.

Another method of developing the prototype implementation is to implement a translator from the new language into an existing high-level language. Such a translation is usually straightforward (as long as the target language is chosen with care). Clearly the existing target language must already be supported by a suitable implementation. This was precisely the method chosen for the first implementation of C++, which used the `cfront` translator to convert the source program into C.

Development of the prototype must be guided by the language specification, whether the specification is formal or informal. The specification tells the implementor which programs are well-formed (i.e., conform to the language's syntax and contextual constraints) and what these programs should do when run.

9.1.4 Compilers

A prototype is not suitable for use over an extended period by a large number of programmers building real applications. When it has served its purpose of allowing programmers to try out the new language and provide feedback to the language designer, the prototype should be superseded by a higher-quality implementation. This is invariably a compiler – or, more likely, a family of compilers, generating object code for a number of target machines. Such a high-quality implementation is referred as an *industrial-strength* compiler.

The work that went into developing a prototype need not go to waste. If the prototype was an interpretive compiler, for example, we can bootstrap it to make a compiler that generates real machine code (see Section 2.6).

Development of compilers must be guided by the language specification. A syntactic analyzer can be developed systematically from the source language's syntactic specification (see Chapter 4). A specification of the source language's scope rules and type rules should guide the development of a contextual analyzer (see Chapter 5). Finally, a specification of the source language's semantics should guide the development of a code specification, which should in turn be used to develop a code generator systematically (see Chapter 7).

In practice, contextual constraints and semantics are rarely specified formally. If we compare separately-developed compilers for the same language, we often find that they are consistent with respect to syntax, but inconsistent with respect to contextual constraints and semantics. This is no accident, because syntax is usually specified formally, and therefore precisely, and everything else informally, leading inevitably to misunderstanding.

9.2 Error reporting

All programmers make errors – frequently. A high-quality language processor assists the programmer to locate and correct these errors. Here we examine detection and reporting of both compile-time and run-time errors.

9.2.1 Compile-time error reporting

The language specification defines a set of well-formed programs. A minimalist view of a compiler's function is that it simply rejects any ill-formed program. But a good-quality compiler should be more helpful.

As well as rejecting an ill-formed program, the compiler should report the location of each error, together with some explanation. It should at least distinguish between the major categories of compile-time error:

- ***Syntactic error:*** missing or unexpected characters or tokens. The error report might indicate what characters or tokens *were* expected.
- ***Scope error:*** a violation of the language's scope rules. The error report should indicate which identifier was declared twice, or used without declaration.
- ***Type error:*** a violation of the language's type rules. The error report should indicate which type rule was violated, and/or what type was expected.

Ideally the error report should be self-explanatory. If this is not feasible, it should at least refer to the appropriate section of the language specification.

If the compiler forms part of an integrated language processor, and thus the programmer can switch very easily between editing and compiling, it is acceptable for the compiler to halt on detecting the first error. The compiler should highlight the erroneous phrase and pass control immediately to the editor. The programmer can then correct the error and reinvoke the compiler.

On the other hand, a 'batch' or 'software tool' compiler – one intended to compile the entire source program without interaction with the programmer – should detect and report as many errors as it can find. This allows the programmer to correct several errors after each compilation. This requirement has a significant impact on the compiler's internal organization. After detecting and reporting an error, the compiler should attempt ***error recovery***. This means that the compiler should try to get itself into a state where analysis of the source program can continue as normally as possible. Unfortunately, effective error recovery is difficult.

Example 9.1 Reporting syntactic errors

The following Triangle program fragment contains some common syntactic errors:

```
        let
            var score: Integer;
            var grade: Char
(1)         var pass:  Boolean
        in
            begin
            ...;
(2)         if 50 <= score /\ score < 60 then
(3)             grade := 'C';
(4)             pass = true
(5)         else
                ...
            end
```

These errors should be detected during parsing.

At (1), the token 'var' is encountered unexpectedly. Clearly, the error is a missing semicolon between the declarations, and the best error recovery is to continue parsing as if the semicolon had been there.

After the assignment command at (3), a semicolon is encountered where 'else' was expected. Here error recovery is more difficult. (Recall that the parser works in a single pass through the source program, and has not yet seen the tokens after the semicolon.) There are two reasonable ways in which the parser might attempt to recover at this point:

(a) The parser might assume that the else-part of the if-command is missing, and continue as if the if-command had been completely parsed. Given the above source program, this error recovery would turn out badly: the parser would later and unexpectedly encounter the token 'else' at (5), and would spuriously report a syntactic error at that point.

(b) Alternatively, the parser might skip tokens in the hope of finding the expected 'else'. Given the above source program, this error recovery would turn out reasonably well: the parser would find the token 'else' at (5), and would then resume parsing the if-command. The only drawback is that the parser would skip the tokens 'pass = true', and thus would overlook the syntactic error there.

In contrast, given a different program where the 'else' really was missing, error recovery (a) would turn out well, but (b) would turn out badly.

The expression at (2) illustrates another problem with error reporting. This expression is syntactically well-formed, but the Triangle parser will treat this expression as equivalent to '((50 <= score) /\ score) < 60' – not at all what the programmer intended! Consequently, contextual analysis will report type errors in connection with the operators '/\' and '<'. The programmer's actual mistake, however, was the syntactic mistake of failing to parenthesize the expression properly.

□

Example 9.2 Reporting contextual errors

The following Triangle program fragment contains scope and type errors:

```
      let
         var phonenum: Integer;
         var local: Boolean
      in
         begin
         ...;
(1)      if phonenum[0] = '0' then
(2)         locale := false
         else
            ...
         end
```

These errors should be detected during contextual analysis.

Consider the expression at (1). The phrase '`phonenum[0]`' clearly violates the indexing operation's type rule, since `phonenum` is not of array type. But what error recovery is appropriate? It is not at all obvious what type should be ascribed to '`phonenum[0]`', to allow type checking to continue. If the type checker ascribes the type *int*, for example, then at the next step it will find that the operands of '=' appear to violate that operator's type rule (one operand being *int* and the other *char*), and it will generate a second error report, which is actually spurious. Fortunately, the result type of '=' does not depend on the types of its operands, so the type checker should obviously ascribe the type *bool* to the expression '`phonenum[0] = '0'`'. At the next step the type checker will find that this expression satisfies the if-command's type rule.

At (2), there is an applied occurrence of an identifier, `locale`, that has not been declared, in violation of a scope rule. Again, what error recovery is appropriate? Suppose that the type checker arbitrarily chooses *int* as the type of `locale`. Subsequently the type checker will find that the assignment command's type rule appears to be violated (one side being *int* and the other *bool*), and again it will generate a spurious error report.

□

To facilitate error recovery during type checking, it is useful for the type checker to ascribe a special improper type, *error-type*, to any ill-typed expression. The type checker can then ignore *error-type* whenever it is subsequently encountered. This technique would avoid both the spurious error reports mentioned in Example 9.2.

As these examples illustrate, it is easy for a compiler to discover that the source program is ill-formed, and to generate error reports; but it is difficult to ensure that the compiler never generates misleading error reports. There is a genuine tension between the task of compiling well-formed source programs and the need to make some sense of ill-formed programs. A compiler is structured primarily to deal with well-formed source programs, so it must be enhanced with special error recovery algorithms to make it deal reasonably with ill-formed programs.

Syntactic error recovery is particularly difficult. At one extreme, an over-ambitious error recovery algorithm might induce an avalanche of spurious error reports. At the opposite extreme, an over-cautious error recovery algorithm might skip a large part of the source program and fail to detect genuine syntactic errors.

9.2.2 Run-time error reporting

Run-time error reporting is a completely different but equally important problem. Among the more common run-time errors are:

- arithmetic overflow
- division by zero
- out-of-range array indexing

These errors can be detected only at run-time, because they depend on values computed at run-time.[1]

Some run-time errors are detected by the target machine. For example, overflow may result in a machine interrupt. But in some machines the only effect of overflow is to set a bit in the condition code register, and the object program must explicitly test this bit whenever there is a risk of overflow.

Other run-time errors are not detected by the machine at all, but instead must be detected by tests in the object program. For example, out-of-range array indexing might result in computing the address of a word that is not actually part of the array. This is usually not detected by the machine unless the computed address is outside the program's address space.

These examples illustrate only typical machine behavior. Real machines range from one extreme, where *no* run-time errors are detected automatically, to the opposite extreme, where all the more common run-time errors are detected automatically. The typical situation is that some run-time errors are detected by hardware, leaving others to be detected by software.

Where a particular run-time error is *not* detected by hardware, the compiler should generate code to test for the error explicitly. In array indexing, for example, the compiler should generate code not only to evaluate the index but also to check whether it lies within the array's index range.

---

[1] If the language is dynamically typed, i.e., a variable can take values of different types a different times, then type errors also are run-time errors. However, we do not conside dynamically-typed languages here.

Example 9.3 Detecting array indexing errors

The following Triangle program fragment illustrates array indexing:

```
        let
           var name: array 4 of Char;
           var i: Integer
        in
           begin
           ...;
(1)        name[i] := ' ';
           ...
           end
```

Assume that characters and integers occupy one word each, and that the addresses of global variables `name` and `i` are 200 and 204, respectively. Thus `name` occupies words 200 through 203; and the address of `name[`i`]` is 200 + i, provided that $0 \leq i \leq 3$.

The Triangle compiler does not currently generate index checks. The assignment command at (1) will be translated to object code like this (omitting some minor details):

```
LOADL 48          – fetch the blank character
LOAD  204         – fetch the value of i
LOADL 200         – fetch the address of name[0]
CALL  add         – compute the address of name[i]
STOREI            – store the blank character at that address
```

This code is dangerous. If the value of `i` is out of range, the blank character will be stored, not in an element of `name`, but in some other variable – possibly of a different type. (If the value of `i` happens to be 4, then `i` itself will be corrupted in this way.)

We could correct this deficiency by making the compiler generate object code with index checks, like this:

```
LOADL 48          – fetch the blank character
LOAD  204         – fetch the value of i
LOADL 0           – fetch the lower bound of name
LOADL 3           – fetch the upper bound of name
CALL  rangecheck  – check that the index is within range
LOADL 200         – fetch the address of name[0]
CALL  add         – compute the address of name[i]
STOREI            – store the blank character at that address
```

The index check is italicized for emphasis. The auxiliary routine *`rangecheck`*, when called with arguments i, m, and n, is supposed to return i if $m \leq i \leq n$, or to fail otherwise. The space cost of the index check is three instructions, and the time cost is three instructions plus the time taken by *`rangecheck`* itself.

□

Software run-time checks are expensive in terms of object-program size and speed. Without them, however, the object program might overlook a run-time error, eventually failing somewhere else, or terminating with meaningless results. And, let it be emphasized, if a compiler generates object programs whose behavior differs from the language specification, it is simply incorrect. The compiler should, at the very least, allow the programmer the option of including or suppressing run-time checks. Then a program's unpredictable behavior would be the responsibility of the programmer who opts to suppress run-time checks.

Whether the run-time check is performed by hardware or software, there remains the problem of generating a suitable error report. This should not only describe the nature of the error (e.g., 'arithmetic overflow' or 'index out of range'), but should also locate it in the source program. An error report stating that overflow occurred at instruction address 1234 (say) would be unhelpful to a programmer who is trying to debug a high-level language program. A better error report would locate the error at a particular line in the source program.

The general principle here is that error reports should relate to the source program rather than the object program. Another example of this principle is a facility to display the current values of variables during or after the running of the program. A simple storage dump is of little value: the programmer cannot understand it without a detailed knowledge of the run-time organization assumed by the compiler (data representation, storage allocation, layout of stack frames, layout of the heap, etc.). Better is a symbolic dump that displays each variable's source-program identifier, together with its current value in source-language syntax.

Example 9.4 Reporting run-time errors

Consider the Triangle program fragment of Example 9.3. Suppose that an out-of-range index is detected at (1). The following error report and storage dump are expressed largely in object-program terms:

```
Array indexing error at instruction address 1234.
Data store at this point:

        address     content
            ...         ...
            200          74
            201          97
            202         118
            203          97
            204          10
            ...         ...
```

This information is hard to understand, to put it mildly. It is not clear which array indexing operation failed. There is no indication that some of the words in the data store constitute an array. There is no distinction between different types of data such as integers and characters.

The following error report and storage dump are expressed more helpfully in source-program terms:

```
Array indexing error at line 45.
Data store at this point:

        name  = ['J', 'a', 'v', 'a']
        i     = 10
```

Here the programmer can tell at a glance what went wrong.

□

But how can the source-program line number be determined at run-time? One possible technique is this. We dedicate a register (or storage cell) that will contain the current line number. The compiler generates code to update this register whenever control passes from one source-program line to another. Clearly, however, this technique is costly in terms of extra instructions in the object program.

An alternative technique is as follows. The compiler generates a table relating line numbers to instruction addresses. If the object program stops, the code pointer is used to search the table and determine the corresponding line number. This technique has the great advantage of imposing no time or space overheads on the object program. (The line-number table can be stored separately from the object program, and loaded only if required.)

The generation of reliable line-number information, however, is extremely difficult in the presence of heavily-optimized code. In this case, the code generator may have eliminated some of the original instructions, and substantially re-ordered others, making it very difficult to identify the line number of a given instruction. In the worst case, a single instruction may actually be part of the code for several different lines of source code.

To generate a symbolic storage dump requires more sophisticated techniques. The compiler must generate a 'symbol table' containing the identifier, type, and address of each variable in the source program, and the identifier and entry address of each procedure (and function). If the object program stops, using the symbol table each (live) variable can be located in the data store. The variable's identifier can be printed along with its current value, formatted according to its type. If one or more procedures are active at the time when the program stops, the store will contain one or more stack frames. To allow the symbolic dump to cover local variables, the symbol table must record which variables are local to which procedures, and the procedure to which each frame belongs must be identified in some way. (See Exercise 9.16.)

This problem is compounded on a register machine, where a variable might be located in a register and not in the store. It is also compounded for heavily-optimized code, where several variables with disjoint lifetimes may share the same memory location.

9.3 Efficiency

When we consider efficiency in the context of a compiler, we must carefully distinguish between compile-time efficiency and run-time efficiency. They are not the same thing at all; indeed, there is often a tradeoff between the two. The more a compiler strives to generate efficient (compact and fast) object code, the less efficient (bulkier and slower) the compiler itself tends to become.

The most efficient compilers are those that generate abstract machine code, where the abstract machine has been designed specifically to support the operations of the source language. Compilation is simple and fast because there is a straightforward translation from the source language to the target language, with few special cases to worry about. Such is the Triangle compiler used as a case study in this book. Of course, the object code has to be interpreted, imposing a significant speed penalty at run-time.

Compilers that generate code for real machines are generally less efficient. They must solve a variety of awkward problems. There is often a mismatch between the operations of the source language and the operations provided by the target machine. The target-machine operations are often irregular, complicating the translation. There might be many ways of translating the same source program into object code, forcing the compiler writer to implement lots of special cases in an attempt to generate the best possible object code.

9.3.1 Compile-time efficiency

Let us examine a compiler from the point of view of algorithmic complexity. Ideally, we would like the compiler to run in $O(n)$ time,[2] where n is some measure of the source program's size (for example, the number of tokens). In other words, a 10-fold increase in the size of the source program should result in a 10-fold increase in compilation time. A compiler that runs in $O(n^2)$ time is normally unacceptable: a 10-fold increase in the size of the source program would result in a 100-fold increase in compilation time! In practice, $O(n \log n)$ might be an acceptable compromise.

If all phases of a compiler run in $O(n)$ time, then the compiler as a whole will run in $O(n)$ time.[3] But if just one of the phases runs in $O(n^2)$ time, then the compiler as a whole

---

[2] The O-notation is a way of estimating the efficiency of a program. Let n be the size of the program's input. If we state that the program's running time is $O(n)$, we mean that its running time is proportional to n. (The actual running time could be $100n$ or $0.01n$.) Similarly, $O(n \log n)$ time means time proportional to $n \log n$, $O(n^2)$ time means time proportional to n^2, and so on. In estimates of algorithmic complexity, the constants of proportionality are generally less important than the difference between, for example, $O(n)$ and $O(n^2)$.

[3] Suppose that phase A runs in time an, and phase B in time bn (where a and b are constants). Then the combination of these phases will run in time $an + bn = (a + b)n$, which is still $O(n)$.

will run in $O(n^2)$ time.[4] In general, compilation time is dominated by the phase with the worst time complexity.

The parsing, type checking, and code generation algorithms described in Chapters 4, 5, and 7 do in fact run in $O(n)$ time. However, identification is often a weak link.

Assume that the number of applied occurrences of identifiers in the source program is $O(n)$, and that the average number of entries in the identification table is $O(n)$. If linear search is used, each identification will take $O(n)$ time, so total identification time will be $O(n^2)$. If instead some kind of binary search is used, each identification will take $O(\log n)$ time, so total identification time will be $O(n \log n)$. With clever use of hashing it is possible to bring each identification down to almost constant time, so total identification time will be $O(n)$, the ideal.

There are other weak links in some compilers. Some code transformation algorithms run in $O(n^2)$ time, as we shall see in the next subsection. An extreme case is the polymorphic type inference algorithm used in an ML compiler, which runs in $O(2^{2^n})$ time in pathological cases. (Fortunately these cases never arise in practice!)

9.3.2 Run-time efficiency

Let us now consider the efficiency of object programs, and in particular programs that run on real machines. Perhaps the most problematic single feature of real machines is the fact that they provide general-purpose registers. Computer architects provide registers because they speed up object programs. But compilers have to work harder to generate object programs that make effective use of the registers.

Example 9.5 Code generation for a register machine

The following Triangle command:

```
a := (b*c) - (d + (e*f))
```

would be translated to the following TAM code:

```
LOAD  b
LOAD  c
CALL  mult
LOAD  d
LOAD  e
LOAD  f
CALL  mult
```

---

[4] Suppose that phase A runs in time an, and phase B in time bn^2 (where a and b are constants). Then the combination of these phases will run in time $an + bn^2$. Even if a is much greater than b, the second term will eventually dominate as n increases.

```
CALL   add
CALL   sub
STORE  a
```

As we saw in Chapter 7, a simple efficient code generator can easily perform this translation. The code generator has no registers to worry about.

Now suppose that the target machine has a pool of registers and a typical one-address instruction set. Now the command might be translated to object code like this:

```
LOAD   R1 b
MULT   R1 c
LOAD   R2 d
LOAD   R3 e
MULT   R3 f
ADD    R2 R3
SUB    R1 R2
STORE  R1 a
```

Although this is comparatively straightforward, some complications are already evident. The code generator must allocate a register for the result of each operation. It must ensure that the register is not reused until that result has been used. (Thus `R1` cannot be used during the evaluation of '`d + (e*f)`', because at that time it contains the unused result of evaluating '`b*c`'.) Furthermore, when the right operand of an operator is a simple variable, the code generator should avoid a redundant load by generating, for example, '`MULT R1 c`' rather than '`LOAD R2 c`' followed by '`MULT R1 R2`'.

The above is not the only possible object code, nor even the best. One improvement is to evaluate '`d + (e*f)`' *before* '`b*c`'. A further improvement is to evaluate '`(e*f) + d`' instead of '`d + (e*f)`', exploiting the commutativity of '`+`'. The combined effect of these improvements is to save an instruction and a register:

```
LOAD   R1 e
MULT   R1 f
ADD    R1 d
LOAD   R2 b
MULT   R2 c
SUB    R2 R1
STORE  R2 a
```

The trick illustrated here is to evaluate the more complicated subexpression of a binary operator first.

But that is not all. The compiler might decide to allocate registers to selected variables throughout their lifetimes. Supposing that registers `R6` and `R7` are thus allocated to variables `a` and `d`, the object code could be further improved as follows:

```
LOAD  R1 e
MULT  R1 f
ADD   R1 R7
LOAD  R6 b
MULT  R6 c
SUB   R6 R1
```

□

Several factors make code generation for a register machine rather complicated. Register allocation is one factor. Another is that compilers *must* in practice achieve code improvements of the kind illustrated above – programmers demand nothing less!

But even a compiler that achieves such improvements will still generate rather mediocre object code (typically four times slower than hand-written assembly code). A variety of algorithms have been developed that allow a compiler to generate much more efficient object code (typically twice as slow as hand-written assembly code). These are called ***code transformation*** (or *code optimization*[5]) algorithms. Some of the more common code transformations are:

- ***Constant folding:*** If an expression depends only on known values, it can be evaluated at compile-time rather than run-time.
- ***Common subexpression elimination:*** If the same expression occurs in two different places, and is guaranteed to yield the same result in both places, it might be possible to save the result of the first evaluation and reuse it later.
- ***Code movement:*** If a piece of code executed inside a loop always has the same effect, it might be possible to move that code out of the loop, where it will be executed fewer times.

Example 9.6 Constant folding

Consider the following Java program fragment:

```
static double pi = 3.1416;
...
double volume = 4/3 * pi * r * r * r;
```

The compiler could replace the subexpression '`4/3 * pi`' by 4.1888. This constant folding saves a run-time division and multiplication. The programmer could have written '`4.1888 * r * r * r`' in the first place, of course, but only at the expense of making the program less readable and less maintainable.

The following illustrates a situation where only the compiler can do the folding.

---

[5] The more widely used term, *code optimization*, is actually inappropriate: it is infeasible for a compiler to generate truly optimal object code.

Consider the following Triangle program fragment:

```
type Date = record
               y: Integer, m: Integer, d: Integer
            end;
var hol: array 6 of Date
...
hol[2].m := 12
```

The relevant addressing formula is:

$$\begin{aligned} \textit{address}[\![\texttt{hol[2].m}]\!] &= \textit{address}[\![\texttt{hol[2]}]\!] + 1 \\ &= \textit{address}[\![\texttt{hol}]\!] + 2 \times 3 + 1 \\ &= \textit{address}[\![\texttt{hol}]\!] + 7 \end{aligned}$$

(assuming that each integer occupies one word). Furthermore, if the compiler decide that $\textit{address}[\![\texttt{hol}]\!] = 20$ (relative to SB), then $\textit{address}[\![\texttt{hol[2].m}]\!]$ can be folded t the constant address 27. This is shown in the following object code:

```
LOADL 12
STORE 27[SB]
```

Address folding makes field selection into a compile-time operation. It even mak indexing of a static array by a literal into a compile-time operation.

Example 9.7 Common subexpression elimination

Consider the following Triangle program fragment:

```
var x: Integer;  var y: Integer;  var z: Integer
...
... (x-y) * (x-y+z) ...
```

Here the subexpression 'x-y' is a common subexpression. If the compiler takes special action, the two occurrences of this subexpression will be translated into tv separate instruction sequences, as in object code (a) below. But their results are guara teed to be equal, so it would be more efficient to compute the result once and then co it when required, as in object code (b) below.

(a)
```
LOAD x
LOAD y
CALL sub
LOAD x
LOAD y
CALL sub
LOAD z
CALL add
CALL mult
```

(b)
```
LOAD x
LOAD y
CALL sub      – computes the value of x-y
LOAD -1[ST]   – copies the value of x-y
LOAD z
CALL add
CALL mult
```

Now consider the following Triangle program fragment:

```
type T = ...;
var a: array 10 of T;  var b: array 20 of T
...
a[i] := b[i]
```

Here there is another, less obvious, example of a common subexpression. It is revealed in the addressing formulas for `a[i]` and `b[i]`:

$$address[\![\texttt{a[i]}]\!] = address[\![\texttt{a}]\!] + (i \times 4)$$
$$address[\![\texttt{b[i]}]\!] = address[\![\texttt{b}]\!] + (i \times 4)$$

where i is the value of variable `i`, and where we have assumed that each value of type `T` occupies four words.

The common subexpression '`x-y`' could have been eliminated by modifying the source program. But the common subexpression '$i \times 4$' can be eliminated only by the compiler, because it exists only at the target machine level.

□

Example 9.8 Code movement

Consider the following Triangle program fragment:

```
var name: array 3 of array 10 of Char
...
i := 0;
while i < 3 do
   begin
   j := 0;
   while j < 10 do
      begin name[i][j] := ' '; j := j + 1 end;
   i := i + 1
   end
```

The addressing formula for `name[i][j]` is:

$$address[\![\texttt{name[i][j]}]\!] = address[\![\texttt{name}]\!] + (i \times 10) + j$$

(assuming that each character occupies one word). A straightforward translation of this program fragment will generate code to evaluate $address[\![\texttt{name}]\!] + (i \times 10)$ inside the inner loop. But this code will yield the same address in every iteration of the inner loop, since the variable `i` is not updated by the inner loop.

The object program would be more efficient if this code were moved out of the inner loop. (It cannot be moved out of the outer loop, of course, because the variable `i` is updated by the outer loop.)

□

Constant folding is a relatively straightforward transformation, requiring only local analysis, and is performed even by simple compilers. For example, the Triangle compiler performs constant folding on address formulas.

Other code transformations such as common subexpression elimination and code movement, on the other hand, require nontrivial analysis of large parts of the source program, to discover which transformations are feasible and safe. To ensure that common subexpression elimination is safe, the relevant part of the program must be analyzed to ensure that no variable in the subexpression has been updated between the first and second evaluations of the subexpression. To ensure that code can be safely moved out of a loop, the whole loop must be analyzed to ensure that the movement does not change the program's behavior.

Code transformation algorithms always slow down the compiler, in an absolute sense, even when they run in $O(n)$ time. But some of these algorithms, especially ones that require analysis of the entire source program, may consume as much as $O(n^2)$ time.

Code transformations are only occasionally justified. During program development, when the program is compiled and recompiled almost as often as it is run, fast compilation is more important than generating very efficient object code. It is only when the program is ready for production use, when it will be run many times without recompilation, that it pays to use the more time-consuming code transformation algorithms.

For an industrial-strength compiler, a sensible compromise is to provide *optional* code transformation algorithms. The programmer (who is the best person to judge) can then compile the program without code transformations during the development phase, and can decide when the program has stabilized sufficiently to justify compiling it with code transformations.

9.4 Further reading

More detailed discussions of the major issues in programming language design and specification, and their interaction, may be found in the concluding chapters of the companion textbooks by Watt (1990, 1991). Interesting accounts of the design of a number of major programming languages – including Ada, C, C++, Lisp, Pascal, Prolog, and Smalltalk – may be found in Bergin and Gibson (1996).

A formal specification of a programming language makes a more reliable guide to the implementor than an informal specification. More radically, it might well be feasible to use a suitable formal specification of a programming language to *generate* an implementation automatically. A system that does this is called a ***compiler generator***. Development of compiler generators has long been a major goal of programming languages research.

Good-quality compiler generators are not yet available, but useful progress has been made. From a syntactic specification we can generate a scanner and parser, as described

in Chapters 3 and 4 of Aho *et al.* (1985). The idea of generating an interpreter from a formal semantic specification is explored in Sections 4.4 and 8.4 of Watt (1991). Generating compilers from semantic specifications is the hardest problem. Lee (1989) surveys past efforts in this direction, which have succeeded in generating only poor-quality compilers.

A large variety of syntactic error recovery methods have been proposed and used in practice. One method, which is particularly suitable for use in conjunction with a recursive-descent parser, is described in Welsh and McKeag (1980).

Code transformation is a major topic in compilers. A detailed account is beyond the scope of this textbook. Instead, see the very extensive account in Chapter 10 of Aho *et al.* (1985). Although code transformation is now regarded as an advanced topic, surprisingly it was one of the first topics to engage the attention of compiler writers. In the 1950s, the writers of the first Fortran compiler went to extraordinary lengths to generate efficient object code, perceiving that this was the only way to attract hard-bitten assembly-language programmers to Fortran. The resulting compiler was noteworthy both for its Byzantine compiling algorithms and for its remarkably good object code!

Fortunately, our understanding of compiling algorithms – and of programming language design and specification – has developed a long way since those early days. In this textbook, and in its companions, we have tried to convey this understanding to a wide readership. We hope we have succeeded!

Exercises

9.1 Obtain a sample of ill-formed programs. (A first programming course should be a good source of such programs!) Compile them, and study the error reports. Does the compiler detect every error, and report it accurately? Does the compiler generate any spurious error reports?

9.2 Write a critical account of your favorite language processor's reporting of run-time errors and its diagnostic facilities. Does it detect every run-time error? Does it report errors in source-program terms? Does it provide a symbolic diagnostic facility?

9.3 Obtain a sample of well-formed programs, varying in size. Using your favorite compiler, measure and plot compilation time against source program size (n). Do you think that the compiler takes $O(n)$ time, $O(n \log n)$ time, or worse?

9.4 Obtain a sample of working programs. Using a compiler with a code transformation option, measure these programs' running time with and without code transformation. (If the compiler has several 'levels' of code transformation, try them all.)

9.5* Consider the following Triangle program fragment:

```
var a: array ... of Integer
...
i := m - 1; j := n; pivot := a[n];
while i < j do
  begin
  i := i + 1; while a[i] < pivot do i := i + 1;
  j := j - 1; while a[j] > pivot do j := j - 1;
  if i < j then
     begin
     t := a[i]; a[i] := a[j]; a[j] := t
     end
  end;
t := a[i]; a[i] := a[n]; a[n] := t
```

(a) Find out the object code that would be generated by the Triangle compiler.

(b) Write down the object code that would be generated by a Triangle compiler that performs code transformations such as constant folding, common subexpression elimination, and code movement.

Projects with the Triangle language processor

All of the following projects involve modifications to the Triangle language processor, so you will need to obtain a copy. (It is available from our Web site. See page xv of the Preface for downloading instructions.)

Nearly every project involves a modification to the language. Rather than plunging straight into implementation, you should first *specify* the language extension. Do this by modifying the informal specification of Triangle in Appendix B, following the same style.

9.6** Extend Triangle with additional loops as follows.

(a) A repeat-command:

`repeat` *C* `until` *E*

is executed as follows. The subcommand *C* is executed, then the expression *E* is evaluated. If the value of the expression is *true*, the loop terminates, otherwise the loop is repeated. The subcommand *C* is therefore executed at least once. The type of the expression *E* must be `Boolean`.

(b) A for-command:

`for` *I* `from` E_1 `to` E_2 `do` *C*

is executed as follows. First, the expressions E_1 and E_2 are evaluated, yielding the integers m and n (say), respectively. Then the subcommand C is executed repeatedly, with identifier I bound in successive iterations to each integer in the range m through n. If $m > n$, C is not executed at all. (The scope of I is C, which may use the value of I but may not update it. The types of E_1 and E_2 must be `Integer`.) Here is an example:

```
for n from 2 to m do
  if prime(n) then
    putint(n)
```

9.7** Extend Triangle with a case-command of the form:

```
case E of
  IL1: C1;
  ...;
  ILn: Cn;
  else: C0
```

This command is executed as follows. First E is evaluated; then if the value of E matches the integer-literal IL_i, the corresponding subcommand C_i is executed. If the value of E matches none of the integer-literals, the subcommand C_0 is executed. (The expression E must be of type `Integer`, and the integer-literals must all be distinct.) Here is an example:

```
case today.m of
   2:   days := if leap then 29 else 28;
   4:   days := 30;
   6:   days := 30;
   9:   days := 30;
  11:   days := 30;
  else: days := 31
```

9.8** Extend Triangle with an initializing variable declaration of the form:

```
var I := E
```

This declaration is elaborated by binding I to a newly created variable. The variable's initial value is obtained by evaluating E. The lifetime of the variable is the activation of the enclosing block. (The type of I will be the type of E.)

9.9** Extend Triangle with unary and binary operator declarations of the form:

```
func O (I1: T1) : T ~ E
```

```
func O (I1: T1, I2: T2) : T ~ E
```

Operators are to be treated like functions. A unary operator application '$O\ E$' is to be treated like a function call '$O(E)$', and a binary operator application '$E_1\ O\ E_2$' is to be treated like a function call '$O(E_1, E_2)$'.

Here are some examples:

```
func -- (i: Integer) : Integer ~ 0 - n;

func ** (b: Integer, n: Integer) : Integer ~
  if n = 0
  then 1
  else n * (b ** (n-1))   ! assuming that n > 0
```

(*Notes:* The Triangle lexicon, Section B.8, already provides a whole class of operators from which the programmer may choose. The Triangle standard environment, Section B.9, already treats the standard operators '+', '-', '*', etc., like predeclared functions.)

9.10** Replace Triangle's constant and variable parameters by value and result parameters. Design your own syntax.

9.11** Extend Triangle with enumeration types. Provide a special enumeration type declaration of the form:

`enum type` I ~ $(I_1, \ldots, I_n)$

which creates a new and distinct primitive type with n values, and respectively binds the identifiers I_1, ..., and I_n to these values. Make the generic operations of assignment, '=', and '\=' applicable to enumeration types. (They are applicable to all Triangle types.) Provide new operations of the form '`succ` E' (successor) and '`pred` E' (predecessor), where `succ` and `pred` are keywords.

9.12** Extend Triangle with a new family of types, `string` n, whose values are strings of exactly n characters ($n \geq 1$). Provide string-literals of the form "$c_1 \ldots c_n$". Make the generic operations of assignment, '=', and '\=' applicable to strings. Provide a new binary operator '<<' (lexicographic comparison). Finally, provide an array-like string indexing operation of the form 'V[E]', where V names a string value or variable. (*Hint:* Represent a string in the same way as a static array.)

Or:
Extend Triangle with a new type, `String`, whose values are character strings of any length (including the empty string). Provide string-literals of the form "$c_1 \ldots c_n$" ($n \geq 0$). Make the generic operations of assignment, '=', and '\=' applicable to strings. Provide new binary operators '<<' (lexicographic comparison) and '++' (concatenation). Finally, provide an array-like string indexing operation of the form 'V[E]', and a substring operation of the form '$V[E_1 : E_2]$', where V names a string value or variable. But do not permit string variables to be selectively updated. (*Hint:* Use an indirect representation for strings. The handle should consist of a length field and a pointer to an array of characters stored in the heap. In the absence of selective updating, string assignment can be implemented simply by copying the handle.)

9.13** Extend Triangle with recursive types. Provide a special recursive type declaration of the form:

```
rec type I ~ T
```

where the type-denoter T may contain applied occurrences of I. (Typically, T will be a record type containing one or more fields of type I.) Every recursive type is to include a special empty value, denoted by the keyword `nil`. Do not permit variables of recursive types to be selectively updated. Example:

```
rec type IntList ~
     record head: Integer, tail: IntList end;

func cons (n: Integer, ns: IntList): IntList ~
  {head ~ n, tail ~ ns};

proc putints (ns: IntList) ~
  if ns \= nil then
     begin
     putint(ns.head); put(' ');
     putints(ns.tail)
     end;

var primes: IntList
...
primes := cons(2, cons(3, cons(5,
             cons(7, cons(11, nil)))));
putInts(primes)
```

(*Hint:* See Section 6.1.6 for a suggested indirect representation of recursive types. In the absence of selective updating, assignment can be implemented simply by copying the handle.)

9.14** Extend Triangle with packages.

(a) First make package declarations of the form:

```
package I ~ D end
```

This declaration is elaborated as follows. I is bound to a package of entities declared in D. The packaged entities may be constants, variables, types, procedures, functions, or any mixture of these. A packaged entity declared with identifier I_2 is named $I\$I_2$ outside the package declaration. Example:

```
package Graphics ~

  type Point ~
       record h: Integer, v: Integer end;

  func cart (x: Integer, y: Integer): Point ~
     {h ~ x, v ~ y};
```

```
    proc movehoriz (dist: Integer, var p: Point) ~
        p.h := p.h + dist

end;

var z: Graphics$Point
...
z := Graphics$cart(3, 4);
Graphics$movehoriz(7, z)
```

(b) Further extend Triangle with package declarations of the form:

```
package I ~ D1 where D2 end
```

A declaration of this form supports information hiding. Only the entities declared in D_1 are visible outside the package. The entities declared in D_2 are visible only in D_1.

9.15** Modify the Triangle language processor to perform run-time index checks, wherever necessary. (*Hint:* Add a new primitive routine `rangecheck` to TAM, as suggested in Example 9.3.)

9.16** Modify the Triangle language processor to produce run-time error reports and symbolic dumps along the lines illustrated in Example 9.4. You will have to modify both the compiler and the interpreter. (*Hint:* First, restrict your attention to global variables of primitive type. Then deal with procedures and local variables. Finally, deal with variables of composite type.)

9.17** Modify the Triangle compiler to perform constant folding wherever possible.

9.18** Modify the Triangle compiler to perform code movement in the following circumstances. Suppose that in the following loop:

```
while ... do ... E ...
```

the (sub)expression E is *invariant*, i.e., it does not use the value of any variable that is updated anywhere in the loop. Then transform the loop to:

```
let const I ~ E
in
   while ... do ... I ...
```

where I is some identifier not used elsewhere in the loop. (*Hint:* Implement this by a transformation of the decorated AST representing the source program.)