CHAPTER SEVEN

Code Generation

Syntactic and contextual analysis are concerned with analysis of the source program; thus they are dependent only on the source language. Code generation is concerned with translation of the source program to object code, and so is dependent on both the source language and the target machine (whether real or abstract). It is possible to expound general principles of syntactic or contextual analysis, as in Chapters 4 and 5. But the influence of the target machine makes it much harder to expound general principles of code generation.

The main problem is that target machines are extremely varied. Some machines provide registers for storage of intermediate results; others provide a stack; still others provide both. Some machines provide instructions with zero, one, two, or three operands, or a mixture of these. Some machines provide a single addressing mode; others provide many. The structure of a code generator is heavily influenced by such aspects of the target machine architecture. A code generation algorithm suitable for one target machine might be difficult or impossible to adapt to a dissimilar target machine.

The major subproblems of code generation are the following:

- ***Code selection***. This is the problem of deciding which sequence of target machine instructions will be the object code of each phrase in the source program. For this purpose we write *code templates*. A code template is a general rule specifying the object code of all phrases of a particular form (e.g., all assignment commands, or all function calls). In practice, code selection is often complicated by special cases.
- ***Storage allocation***. This is the problem of deciding the storage address of each variable in the source program. The code generator can decide the address of each global variable exactly (*static storage allocation*), but it can decide the address of each local variable only relatively (*stack storage allocation*).
- ***Register allocation***. If the target machine has registers, they should be used to hold intermediate results during expression evaluation. The code generator, knowing that a particular register contains the current value of variable *v*, should take advantage of that to save a memory cycle when the value of *v* is needed. Many complications arise in practice: there might not be enough registers to evaluate a complex expression; or some registers might be reserved for particular purposes (such as indexing).

Code generation for a stack machine is significantly easier than code generation for a register machine. As we saw in Chapter 6, we need a stack anyway to implement

procedures and local variables. A stack is also convenient for expression evaluation. The problem of register allocation simply disappears. In this book, therefore, we consider only code generation for a stack machine, and concentrate on the subproblems of code selection and storage allocation. We use the abstract machine TAM as an illustrative target machine. (TAM was introduced in Chapter 6, and is fully described in Appendix C.)

7.1 Code selection

The function of the code generator is to translate source programs to semantically equivalent object programs. When we design a code generator, therefore, we must be guided by the semantics of the source language. Now a semantic specification is generally structured in terms of the semantics of phrases such as expressions, commands, and declarations. In code generation we should follow the same structure: we should specify the translation of source programs to object programs inductively, by specifying the translation of individual phrases to object code.

Usually there are many correct translations of a given program or phrase. There may be several sequences of instructions that correctly perform a given source-language operation. So a basic task of the code generator is to decide which sequence of instructions to generate in each case. This is called ***code selection***.

7.1.1 Code templates

In general, we specify code selection inductively over the phrases of the source language, using *code functions* and *code templates*. The following example introduces the basic idea.

Example 7.1 Code functions and code templates

Consider translation of some hypothetical source language to object code. We can specify the translation of commands to object code by introducing the following code function:

execute : Command → Instruction*

This function will translate each command C to a sequence of target-machine instructions that achieves the effect of executing C. We must define this function over all the commands in the source language. This we do by means of code templates.

Consider a sequential command, typically of the form 'C_1 ; C_2'. Its semantics was specified by (1.20c):

> The sequential command 'C_1 ; C_2' is executed as follows. First C_1 is executed; then C_2 is executed.

We can easily specify the translation of 'C_1 ; C_2' to object code by means of the following code template:

execute ⟦C_1 ; C_2⟧ =
    execute C_1
    execute C_2

This code template may be read as follows: the code to execute 'C_1 ; C_2' consists of the code to execute C_1, followed by the code to execute C_2.

Most code templates contain specific instructions. For example, the code template for a simple assignment command 'I := E', where I is a variable identifier, might look like this:

execute ⟦I := E⟧ =
    evaluate E
    `STORE` a where a = address of variable named I

This code template may be read as follows: the code to execute 'I := E' consists of the code to evaluate E, followed by a `STORE` instruction whose operand field is the address of the variable I.

For instance, here is a possible translation of a sequential command containing two assignment commands:[1]

execute ⟦`f := f*n; n := n-1`⟧ =

execute ⟦`f := f*n`⟧	`LOAD f`
	`LOAD n`
	`CALL mult`
	`STORE f`
execute ⟦`n := n-1`⟧	`LOAD n`
	`CALL pred`
	`STORE n`

□

Each code template specifies the object code to which a phrase is translated, in terms of the object code to which its subphrases are translated. If a phrase is primitive (i.e., contains no subphrases), then its code template should specify its object code directly. A complete set of code functions and code templates specifies the translation of the entire source language to object code. More precisely:

- The ***object code*** of each source-language phrase is the sequence of instructions to which it will be translated. The object code is in Instruction*.

---

[1] The actual machine instructions will contain numerical addresses. Here we write `f` and `n` to stand for the addresses of variables `f` and `n`, and `mult` and `pred` for the addresses of the respective primitive routines. We will use this convention freely, to make the examples readable.

- For each phrase class ***P*** in the source language's abstract syntax, we introduce a ***code function***, f_P, that translates each phrase in class ***P*** to object code:

  f_P : ***P*** → Instruction*

- We define the code function f_P by a number of ***code templates***, with (at least) one code template for each distinct form of phrase in class ***P***. If some form of phrase in ***P*** has subphrases Q and R, then the corresponding code template will look something like this:

  f_P [[... Q ... R ...]] =
  ...
  f_Q Q
  ...
  f_R R
  ...

  where f_Q and f_R are code functions appropriate for subphrases Q and R. (The order shown above is not fixed: Q's object code may precede or follow R's object code.)

A ***code specification*** is a collection of code functions and code templates. It must cover the entire source language, i.e., it must specify the translation of every well-formed source program to object code. Let us now examine a complete code specification.

Example 7.2 Code specification for Mini-Triangle to TAM code

Consider the language Mini-Triangle, whose syntax and semantics were given in Examples 1.3 and 1.8. We will present a code specification for the translation from Mini-Triangle to TAM code.

The relevant phrase classes in this language are Program, Command, Expression, Operator, V-name, and Declaration. We first introduce code functions for these phrase classes:

run	: Program	→ Instruction*	(7.1)
execute	: Command	→ Instruction*	(7.2)
evaluate	: Expression	→ Instruction*	(7.3)
fetch	: V-name	→ Instruction*	(7.4)
assign	: V-name	→ Instruction*	(7.5)
elaborate	: Declaration	→ Instruction*	(7.6)

The object code of each phrase is a sequence of instructions that will behave as shown in Table 7.1.

(Mini-Triangle has other phrase classes, but these will not have corresponding code functions. There are two code functions for V-name, which will be used in different contexts.)

Table 7.1 Summary of code functions for Mini-Triangle.

Phrase class	Code function	Effect of generated object code
Program	*run P*	Run the program *P* and then halt, starting and finishing with an empty stack.
Command	*execute C*	Execute the command *C*, possibly updating variables, but neither expanding nor contracting the stack.
Expression	*evaluate E*	Evaluate the expression *E*, pushing its result on to the stack top, but having no other effect.
V-name	*fetch V*	Push the value of the constant or variable named *V* on to the stack top.
V-name	*assign V*	Pop a value from the stack top, and store it in the variable named *V*.
Declaration	*elaborate D*	Elaborate the declaration *D*, expanding the stack to make space for any constants and variables declared therein.

A Mini-Triangle program is simply a command *C*. The program is run simply by executing *C* and then halting. The code template for a program is therefore as follows:

$$\begin{aligned} &\textit{run}\ [\![C]\!] = \\ &\qquad \textit{execute}\ C \\ &\qquad \texttt{HALT} \end{aligned} \tag{7.7}$$

The code templates for commands are as follows:

$$\begin{aligned} &\textit{execute}\ [\![V := E]\!] = \\ &\qquad \textit{evaluate}\ E \\ &\qquad \textit{assign}\ V \end{aligned} \tag{7.8a}$$

This is easy to understand: '*evaluate E*' will have the net effect of pushing the value yielded by *E* on to the stack, and '*assign V*' will pop that value and store it in the variable *V*.

$$\begin{aligned} &\textit{execute}\ [\![I\ (\ E\ )\ ]\!] = \\ &\qquad \textit{evaluate}\ E \\ &\qquad \texttt{CALL}\ p \qquad\qquad \text{where } p = \text{address of primitive routine } I \end{aligned} \tag{7.8b}$$

This is the code template for a procedure call. Since there are no procedure declarations in Mini-Triangle, *I* must be the identifier of a predefined procedure such as `putint`. The above `CALL` instruction calls the corresponding primitive routine in TAM.

$$\begin{aligned} &\textit{execute}\ [\![C_1\ ;\ C_2]\!] = \\ &\qquad \textit{execute}\ C_1 \\ &\qquad \textit{execute}\ C_2 \end{aligned} \tag{7.8c}$$

This was explained in Example 7.1.

execute ⟦`if` *E* `then` C_1 `else` C_2⟧ = (7.8d)
    evaluate E
    `JUMPIF(0)` *g*
    execute C_1
    `JUMP` *h*
g: *execute* C_2
h:

Here the code '*evaluate E*' will have the net effect of pushing a truth-value on to the stack. The `JUMPIF` instruction will pop and test this value. If it is 0 (representing *false*), control will be transferred to *g* where the code '*execute* C_2' will be selected; otherwise the code '*execute* C_1' will be selected. (The labels *g* and *h* stand for the addresses of the following instructions.)

execute ⟦`while` *E* `do` *C*⟧ = (7.8e)
    `JUMP` *h*
g: *execute C*
h: *evaluate E*
    `JUMPIF(1)` *g*

Here again, the code '*evaluate E*' will have the net effect of pushing a truth-value on to the stack. The `JUMPIF` instruction will pop and test this value. If it is 1 (representing *true*), the code '*execute C*' will be iterated; otherwise iteration will cease. The initial `JUMP` instruction ensures that the code '*evaluate E*' will be executed first; this is in accordance with the semantics of the while-command. (See Exercise 7.1 for discussion of an alternative code template.)

execute ⟦`let` *D* `in` *C*⟧ = (7.8f)
    elaborate D
    execute C
    `POP(0)` *s* if $s > 0$
    where *s* = amount of storage allocated by *D*

This code template shows how storage allocation and deallocation comes in. The code '*elaborate D*' will expand the stack, as a consequence of allocating storage for constants and variables declared in *D*. The code '*execute C*' will be able to access these variables. The `POP` instruction contracts the stack to its original size – in effect, deallocating these constants and variables.

The code templates for expressions are as follows. In each case the object code must have the net effect of pushing a value on to the stack.

evaluate ⟦*IL*⟧ = (7.9a)
    `LOADL` *v* where *v* = *value of IL*

The `LOADL` instruction will simply push the integer-literal's value *v* on to the stack top.

evaluate ⟦*V*⟧ = (7.9b)
    fetch V

This is self-explanatory.

evaluate ⟦*O E*⟧ = (7.9c)
    evaluate E
    `CALL` *p* where *p* = address of primitive routine corresponding to *O*

evaluate ⟦E_1 *O* E_2⟧ = (7.9d)
    evaluate E_1
    evaluate E_2
    `CALL` *p* where *p* = address of primitive routine corresponding to *O*

The above two code templates show how applications of unary and binary operators will be translated. Note how expression evaluation exploits the stack: the object code will first evaluate the operand(s), and then apply the operation corresponding to *O*. The latter is achieved by calling an appropriate primitive routine, e.g., `not` for '\' , `add` for '+' , or `lt` for '<'.

In Mini-Triangle, a value-or-variable-name is just an identifier that has been declared as a constant or variable. Being global, that constant or variable will be addressed relative to register SB. Here (anticipating Section 7.3) we assume that its address has already been determined:

fetch ⟦*I*⟧ = (7.10)
    `LOAD` *d*`[SB]` where *d* = address bound to *I* (relative to SB)

assign ⟦*I*⟧ = (7.11)
    `STORE` *d*`[SB]` where *d* = address bound to *I* (relative to SB)

The code templates for declarations are as follows. In each case the object code must expand the stack to make space for the declared constants and variables.

elaborate ⟦`const` *I* ~ *E*⟧ = (7.12a)
    evaluate E

The constant declaration's object code must expand the stack by enough space to contain the constant's value. The code '*evaluate E*' will do that, simply by depositing the value at the stack top. (In addition, the constant's address must be bound to *I* for future reference – this address will be needed whenever (7.10) is applied. We will see in Section 7.3 how this is done.)

elaborate ⟦`var` *I* : *T*⟧ = (7.12b)
    `PUSH 1`

This `PUSH` instruction will expand the stack by one word, enough space to accommodate the newly allocated variable. (The only types in Mini-Triangle are truth values and integers, and these occupy one word each in TAM.) The newly allocated variable is not initialized. (The variable's address must be bound to *I* for future reference – this address will be needed whenever (7.10) or (7.11) is applied.)

elaborate ⟦D_1 ; D_2⟧ = (7.12c)
elaborate D_1
elaborate D_2

Note that in code templates (7.12a–c) it is a simple matter to predict the total amount of storage allocated by the declaration. This information is required in (7.8f).

□

Example 7.3 Translation of a while-command

The following translation illustrates code templates (7.8a) and (7.8e), among others:

execute ⟦`while i > 0 do i := i-2`⟧ {
  execute ⟦`i := i-2`⟧ {
    30: `JUMP 35`
    31: `LOAD i`
    32: `LOADL 2`
    33: `CALL sub`
    34: `STORE i`
  }
  evaluate ⟦`i > 0`⟧ {
    35: `LOAD i`
    36: `LOADL 0`
    37: `CALL gt`
  }
  38: `JUMPIF(1) 31`
}

Here we are assuming that the while-command's object code starts at address 30. The numbers to the left of the instructions are their addresses.[2]

□

Example 7.4 Translation of a let-command

The following translation of a let-command illustrates code templates (7.8f) and (7.12b), among others:

execute ⟦`let var i:Integer in i := i+2`⟧ {
  elaborate ⟦`var i:Integer`⟧ {
    `PUSH 1`
  }
  execute ⟦`i := i+2`⟧ {
    `LOAD i`
    `LOADL 2`
    `CALL add`
    `STORE i`
  }
  `POP(0) 1`
}

The code generated from this let-command expands the stack by one word to allocate storage for the local variable `i`, and later contracts the stack by one word to deallocate it. The address of this word, say *i*, is used to access the variable within the let-command.

□

---

[2] Usually we omit instruction addresses, but we show them in instruction sequences that include jump instructions.

Example 7.5 Code templates for Triangle values and variables

Code templates (7.10), (7.11), and (7.12b) assume that every Mini-Triangle value occupies one word exactly. This is justified because Mini-Triangle supports only truth values and integers, which occupy one word each in TAM.

The full Triangle language, on the other hand, supports a variety of types including arrays and records. A value or variable of type T will occupy a number of words given by *size T*. (See Section 6.1.) For Triangle we must generalize the code templates to take this into account:

fetch ⟦I⟧ = (7.13)

```
LOAD(s)  d[SB]
```

where s = *size*(type of I),
d = address bound to I (relative to SB)

assign ⟦I⟧ = (7.14)

```
STORE(s)  d[SB]
```

where s = *size*(type of I),
d = address bound to I (relative to SB)

elaborate ⟦`var` I : T⟧ = (7.15)

```
PUSH  s
```

where s = *size* T

We shall use these more general code templates from now on. They are still valid for Mini-Triangle, in which *size T* is always 1.

□

7.1.2 Special-case code templates

There are often several ways to translate a given source-language phrase to object code, some more efficient than others. For example, the TAM code to evaluate the expression 'n + 1' could be:

(a)
```
LOAD   n
LOADL  1
CALL   add
```

or (b)
```
LOAD  n
CALL  succ
```

Object code (a) follows code template (7.9d). That code template is always applicable, being valid for any binary operator and for any subexpressions. Object code (b) is correct only in the special case of the binary operator '+' being applied to the literal value 1. When applicable, this special case gives rise to more efficient object code. It could be specified as follows:

evaluate ⟦E_1 + 1⟧ =
    evaluate E_1
    `CALL succ`

A ***special-case code template*** is one that is applicable only to phrases of a special form. Invariably such phrases are also covered by a more general code template. A

special-case code template is worth having if phrases of the special form occur frequently, and if they allow translation into particularly efficient object code. The following example illustrates another common special case.

Example 7.6 Mini-Triangle constant declarations

The right side of a constant declaration is frequently a literal, as in:

```
let
   ...
   const n ~ 7
   ...
in
   ... n ... n ...
```

Code template (7.12a) specifies that the code '*elaborate* ⟦`const n ~ 7`⟧' will deposit the value 7 in a suitable cell (at the current stack top). Whenever `n` is used, code template (7.10) specifies that the value will be loaded from that cell. The following translation illustrates these code templates:

execute ⟦`let const n ~ 7; var i: Integer in i := n*n`⟧

elaborate ⟦`const n ~ 7`⟧	`LOADL 7`
elaborate ⟦`var i: Integer`⟧	`PUSH 1`
	`LOAD` *n*
execute ⟦`i := n*n`⟧	`LOAD` *n*
	`CALL` *mult*
	`STORE` *i*
	`POP(0)2`

The first instruction '`LOADL 7`' makes space for the constant `n` on the stack top. Instructions of the form '`LOAD` *n*' fetch the constant's value, wherever required. The final instruction '`POP(0) 2`' pops the constant and variable off the stack.

A much better translation is possible: simply use the literal value 7 wherever `n` is fetched. This special treatment is possible whenever an identifier is bound to a *known value* in a constant declaration. This is expressed by the following special-case code templates:

fetch ⟦*I*⟧ = (7.16)

`LOADL` *v* where *v* = value bound to *I* (if known)

elaborate ⟦`const` *I* ~ *IL*⟧ = (7.17)

(i.e., no code)

In (7.17) no code is required to elaborate the constant declaration. It is sufficient that the value of the integer-literal *IL* is bound to *I* for future reference. In (7.16) that value is incorporated into a `LOADL` instruction. Thus the object code is more efficient in both places. The following alternative translation illustrates these special-case code templates:

```
                                    ⎧ elaborate [[const n ~ 7]] ⎧
                                    ⎪ elaborate [[var i: Integer]] ⎨ PUSH  1
execute [[let const n ~ 7;          ⎪                           ⎪ LOADL 7
        var i: Integer              ⎪          execute [[i := n*n]] ⎪ LOADL 7
           in i := n*n]]            ⎪                           ⎪ CALL  mult
                                    ⎪                           ⎪ STORE i
                                    ⎩                           ⎩ POP(0) 1
```

In this object code, each applied occurrence of n has been translated to the literal value 7, and the instruction to elaborate the declaration of n has been eliminated.

□

7.2 A code generation algorithm

A code specification does more than specify a translation from the source language to object code. It also suggests an algorithm for performing this translation. This algorithm traverses the decorated AST representing the source program, emitting target-machine instructions one by one. Both the order of traversal and the instructions to be emitted are determined straightforwardly by the code templates.

In this section we see how to develop a code generator from a code specification. We illustrate this with the Mini-Triangle code specification of Example 7.2.

7.2.1 Representation of the object program

Since its basic function is to generate an object program consisting of target-machine instructions, the code generator must obviously define representations of instructions and instruction sequences. This is easy, as the following example illustrates.

Example 7.7 Representing TAM instructions

A code generator that generates TAM object code must represent TAM instructions and their fields (see Section C.2):

```
public class Instruction {
   public byte  op;            // op-code (0 .. 15)
   public byte  r;             // register field (0 .. 15)
   public byte  n;             // length field (0 .. 255)
   public short d;             // operand field (–32767 .. +32767)

   public static final byte    // op-codes (Table C.2)
         LOADop   =  0,  LOADAop  =  1,
         LOADIop  =  2,  LOADLop  =  3,
         STOREop  =  4,  STOREIop =  5,
```

```
        CALLop  =  6,  CALLIop  =  7,
        RETURNop =  8,
        PUSHop  = 10,  POPop    = 11,
        JUMPop  = 12,  JUMPIop  = 13,
        JUMPIFop = 14,  HALTop  = 15;

    public static final byte  // register numbers (Table C.1)
        CBr =  0,  CTr =  1,  PBr =  2,  PTr =  3,
        SBr =  4,  STr =  5,  HBr =  6,  HTr =  7,
        LBr =  8,  L1r =  9,  L2r = 10,  L3r = 11,
        L4r = 12,  L5r = 13,  L6r = 14,  CPr = 15;

    public Instruction (byte op, byte r, byte n,
                        short d)
    { ... }
}
```

Now the object program can be represented as follows:

```
private Instruction[] code = new Instruction[1024];
private short nextInstrAddr = 0;    // address of next instruction
                                    // to be stored in code
```

The code generator will append instructions in the correct order by successive calls to the following method:

```
private void emit (byte op, byte n, byte r, short d) {
    // Append an instruction with fields op, n, r, d to the object program.
    code[nextInstrAddr++] =
        new Instruction(op, n, r, d);
}
```

□

7.2.2 Systematic development of a code generator

A code specification determines the action of a code generator. The code generator will consist of a set of ***encoding methods***, which cooperate to traverse the decorated AST representing the source program. There will be one encoding method for each ordinary code template, and its task will be to emit object code according to that code template.

In Section 5.3.2 we showed how to design a contextual analyzer as a visitor object. This consisted of a set of visitor methods, `visit`*A*, one for each concrete subclass *A* of `AST`. These visitor methods performed tasks appropriate to contextual analysis (identification and type checking).

Here we will show how to design a code generator likewise as a visitor object. In this case the visitor methods `visit`*A* will perform tasks appropriate to code generation.

Many of these visitor methods will simply be encoding methods. For example, the visitor/encoding methods for commands will be `visitAssignCommand`, `visitCallCommand`, etc., and their implementations will be determined by the code templates for '*execute* ⟦*V* := *E*⟧', '*execute* ⟦*I* (*E*)⟧', etc.

Table 7.2 Summary of visitor/encoding methods for the Mini-Triangle code generator.

Phrase class	Visitor/encoding method	Behavior of visitor/encoding method
Program	`visitProgram`	Generate code as specified by '*run P*'.
Command	`visit...Command`	Generate code as specified by '*execute C*'.
Expression	`visit...Expression`	Generate code as specified by '*evaluate E*'.
V-name	`visit...Vname`	Return an entity description for the given value-or-variable-name (explained in Section 7.3.)
Declaration	`visit...Declaration`	Generate code as specified by '*elaborate D*'.
Type-denoter	`visit...TypeDenoter`	Return the size of the given type.

Example 7.8 Mini-Triangle-to-TAM code generator

Let us design a code generator that translates Mini-Triangle to TAM object code. We shall assume the code specification of Example 7.2, and the definition of `AST` and its subclasses in Example 4.19.

The code generator will include visitor/encoding methods for commands, expressions, and declarations:

```
public Object visit...Command
                (...Command com, Object arg);
  // Generate code as specified by 'execute com'.

public Object visit...Expression
                (...Expression expr, Object arg);
  // Generate code as specified by 'evaluate expr'.

public Object visit...Declaration
                (...Declaration decl, Object arg);
  // Generate code as specified by 'elaborate decl'.
```

There will be one visitor/encoding method for each form of command (`visitAssignmentCommand`, `visitIfCommand`, `visitWhileCommand`, etc.). Each such method will have an argument `com` of the appropriate concrete subclass of `Command` (`AssignmentCommand`, `IfCommand`, `WhileCommand`, etc.). Each such method will also have an `Object` argument and an `Object` result, but for the moment these will not actually be needed.

Likewise there will be one visitor/encoding method for each form of expression, and one visitor/encoding method for each form of declaration.

Value-or-variable-names cannot be mapped so simply on to the visitor pattern. There are two code functions for value-or-variable-names, *fetch* and *assign*, each with its own code template. So we need distinct visitor and encoding methods. The encoding methods will be:

```
private void encodeFetch (Vname vname);
    // Generate code as specified by 'fetch vname'.

private void encodeAssign (Vname vname);
    // Generate code as specified by 'assign vname'.
```

Each of these encoding methods will call the visitor method (`visit...Vname`) to find out information about the run-time representation of `vname`. However, they will use this information differently: one to generate a `LOAD` instruction, the other to generate a `STORE` instruction.

There is a single encoding method for a program, `visitProgram`, that will generate code for the entire program:

```
public Object visitProgram (Program prog, Object arg);
    // Generate code as specified by 'run prog'.
```

The visitor/encoding methods of the Mini-Triangle code generator are summarized in Table 7.2.

Now that we have designed the code generator, let us implement some of the encoding methods. The following method generates code for a complete program, using code template (7.7):

```
public Object visitProgram                    run ⟦C⟧ =
             (Program prog,
              Object arg) {
   prog.C.visit(this, arg);                     execute C
   emit(Instruction.HALTop, 0, 0, 0);           HALT
}
```

(For ease of comparison, we show each code template alongside the corresponding code generator steps.)

Now let us implement the visitor/encoding methods for commands. Each such method translates one form of command to object code, according to the corresponding code template (7.8a–f):

```
public Object visitAssignCommand              execute ⟦V := E⟧ =
                (AssignCommand com,
                 Object arg) {
```

```
    com.E.visit(this, arg);                    evaluate E
    encodeAssign(com.V);                       assign V
    return null;
  }
  public Object visitCallCommand               execute ⟦I ( E )⟧ =
                (CallCommand com,
                 Object arg) {
    com.E.visit(this, arg);                    evaluate E
    short p = address of primitive routine
                named com.I;
    emit(Instruction.CALLop,                   CALL p
         Instruction.SBr,
         Instruction.PBr, p);
    return null;
  }
  public Object visitSequentialCommand         execute ⟦C1 ; C2⟧ =
                (SequentialCommand com,
                 Object arg) {
    com.C1.visit(this, arg);                   execute C1
    com.C2.visit(this, arg);                   execute C2
    return null;
  }
  public Object visitLetCommand                execute ⟦let D
                (LetCommand com,                          in C⟧ =
                 Object arg) {
    com.D.visit(this, arg);                    elaborate D
    com.C.visit(this, arg);                    execute C
    short s = amount of storage allocated by D;
    if (s > 0)                                 if s > 0
      emit(Instruction.POPop, 0, 0, s);          POP(0) s
    return null;
  }
```

The `visitIfCommand` and `visitWhileCommand` methods, omitted here, will be implemented in Example 7.9. The `visitLetCommand` method will be completed in Example 7.13.

In `visitCallCommand`, the address of a primitive routine *I* relative to PB is determined from information associated with the declaration of that routine. The various primitive routines and their addresses are given in Table C.3.

Now let us implement the visitor/encoding methods for expressions. Each such method translates one form of expression to object code, according to the corresponding code template (7.9a–d):

```
public Object visitIntegerExpression      evaluate [[IL]] =
               (IntegerExpression expr,
                Object arg) {
   short v = valuation(expr.IL.spelling);
   emit(Instruction.LOADLop, 0, 0, v);      LOADL v
   return null;
}

public Object visitVnameExpression        evaluate [[V]] =
               (VnameExpression expr,
                Object arg) {
   encodeFetch(expr.V);                      fetch V
   return null;
}

public Object visitUnaryExpression        evaluate [[O E]] =
               (UnaryExpression expr,
                Object arg) {
   expr.E.visit(this, arg);                  evaluate E
   short p = address of primitive routine
                named expr.O;
   emit(Instruction.CALLop,                  CALL p
         Instruction.SBr,
         Instruction.PBr, p);
   return null;
}

public Object visitBinaryExpression       evaluate [[E1 O
               (BinaryExpression expr,                E2]] =
                Object arg) {
   expr.E1.visit(this, arg);                 evaluate E1
   expr.E2.visit(this, arg);                 evaluate E2
   short p = address of primitive routine
                named expr.O;
   emit(Instruction.CALLop,                  CALL p
         Instruction.SBr,
         Instruction.PBr, p);
   return null;
}
```

In `visitIntegerExpression`, we used the following auxiliary function:

```
private static short valuation (String intLit)
   // Return the value of the integer-literal spelled intLit.
```

The visitor/encoding methods for declarations, and the encoding methods `encodFetch` and `encodeAssign`, will be implemented in Example 7.13.

Finally, the code generator must define a method that initiates the AST traversal. The completed code generator becomes:

```
public final class Encoder implements Visitor {

    ... // Auxiliary methods, as above.
    ... // Visitor/encoding methods, as above.

    public void encode (Program prog) {
        prog.visit(this, null);
    }
}
```

□

Compare the code generator developed in Example 7.8 with the code specification of Example 7.2. For the most part, it is easy to see how the code generator was developed:

- For each `AST` concrete subclass *A* there is an encoding method, `visit`*A*. This method has an argument that represents a phrase of class *A*. The implementation of `visit`*A* is developed from the corresponding code template.
- Wherever a code template applies a code function to a subphrase, `visit` is applied to that subphrase to generate the corresponding object code. Where the subphrase is a value-or-variable-name, however, the auxiliary method `encodeFetch` or `encodeAssign` is applied to the subphrase.
- Wherever a code template contains a target machine instruction, the auxiliary method `emit` is called to append that instruction to the object program.

The encoding methods developed in this way cooperate to traverse the AST, generating the object program one instruction at a time.

In a code template, the order of the object code most commonly follows the order of the subphrases. But sometimes the order is different, as in code templates (7.8a) and (7.8e). This causes no difficulty to our code generator: it simply traverses the AST in the order specified by the code templates.[3]

A special-case code template does not turn into a distinct encoding method. Instead, it influences the behavior of the encoding method that deals with the more general case. For example, the special-case code templates (7.16) and (7.17) influence the behavior of the encoding methods `encodeFetch` and `visitConstDeclaration`, as we shall see in Example 7.13.

Now we have outlined a code generator, but a number of particular problems require particular solutions. The following subsection deals with the problem of generating code

---

[3] On the other hand, out-of-order code generation cannot easily be achieved by a one-pass compiler, since such a compiler generates object code 'on the fly' as it parses the source program.

for control structures. Thereafter Sections 7.3 and 7.4 deal with the problems of generating code for declared constants and variables, procedures, functions, and parameters.

7.2.3 Control structures

The code generator appends one instruction at a time to the object program. It can easily determine the address of each instruction, simply by counting the instructions as they are generated.

Source-language control structures, such as if-commands and while-commands, are implemented using unconditional and conditional jump instructions. The *destination address* (i.e., the address of the instruction to which the jump is directed) is the operand field of the jump instruction. A *backward* jump causes no problem, because the jump instruction is generated *after* the instruction at the destination address, so the destination address is already known. But a *forward* jump is awkward, because the jump instruction must be generated *before* the instruction at the destination address, and the destination address cannot generally be predicted at the time the jump instruction is generated.

Fortunately, there is a simple solution to the problem of forward jumps, a technique known as ***backpatching***. When the code generator has to generate a forward jump, it generates an incomplete jump instruction, whose destination address is temporarily set to (say) zero. At the same time the code generator records the address of the jump instruction in a local variable. Later, when the destination address becomes known, the code generator goes back and patches it into the jump instruction.[4]

The following example illustrates the method. Recall that the code generator maintains a variable, `nextInstrAddr`, that contains the address of the next instruction to be generated, and is incremented whenever an instruction is appended to the object program. (See Example 7.7.)

Example 7.9 Backpatching

Recall code template (7.8e):

execute ⟦`while` *E* `do` *C*⟧ =
    `JUMP` *h*
g: *execute C*
h: *evaluate E*
    `JUMPIF(1)` *g*

Here *g* stands for the address of the first instruction of the object code '*execute C*', and *h* stands for the address of the first instruction of the object code '*evaluate E*'. Let us see how `visitWhileCommand` should implement this code template.

---

[4] A similar solution to a similar problem is also used in one-pass assemblers.

The backward jump instruction '`JUMPIF(1) g`' is easily generated as follows. Immediately before code is generated for '*execute C*', the next instruction address is saved in a variable, say `g`, local to `visitWhileCommand`. When the backward jump instruction is later generated, the address in `g` is used as its destination address.

When the forward jump instruction '`JUMP h`' is to be generated, on the other hand, its destination address is not yet known. Instead, an incomplete `JUMP` instruction is generated, with a zero address field. The address of this incomplete instruction is saved in another local variable, say `j`. Later, just before code is generated for '*evaluate E*', the next instruction address is noted, and patched into the instruction at address `j`.

For instance, in Example 7.3 we saw the translation of '`while i > 0 do i := i - 2`'. Here we show in detail how `visitWhileCommand` generates this object code:

(1) It saves the next instruction address (say 30) in `j`.

(2) It generates a `JUMP` instruction with a zero address field:

```
30: JUMP  0
```

(3) It saves the next instruction address (namely 31) in `g`.

(4) It translates the subcommand '`i := i - 2`' to object code:

```
31: LOAD  i
32: LOADL 2
33: CALL  sub
34: STORE i
```

(5) It takes the next instruction address (namely 35), and patches it into the address field of the instruction whose address was saved in `j` (namely 30):

```
30: JUMP  35
```

(6) It translates the expression '`i > 0`' to object code:

```
35: LOAD  i
36: LOADL 0
37: CALL  gt
```

(7) It generates a `JUMPIF` instruction whose address field contains the address that was saved in `g` (namely 31):

```
38: JUMPIF(1) 31
```

The following encoding methods illustrate how backpatching is implemented:

```
public Object visitWhileCommand (
                (WhileCommand com,
                 Object arg) {
    short j = nextInstrAddr;
    emit(Instruction.JUMPop, 0,
         Instruction.CBr, 0);
```

execute ⟦`while` *E* `do` *C*⟧ =

j:
`JUMP` *h*

```
    short g = nextInstrAddr;                    g:
    com.C.visit(this, arg);                     execute C
    short h = nextInstrAddr;                    h:
    patch(j, h);
    com.E.visit(this, arg);                     evaluate E
    emit(Instruction.JUMPIFop, 1,               JUMPIF(1) g
         Instruction.CBr, g);
    return null;
}

public Object visitIfCommand                    execute [[if E
                (IfCommand com,                         then C1
                 Object arg) {                          else C2]] =
    com.E.visit(this, arg);                     evaluate E
    short i = nextInstrAddr;                    i:
    emit(Instruction.JUMPIFop, 0,               JUMPIF(0) g
         Instruction.CBr, 0);
    com.C1.visit(this, arg);                    execute C1
    short j = nextInstrAddr;                    j:
    emit(Instruction.JUMPop, 0,                 JUMP h
         Instruction.CBr, 0);
    short g = nextInstrAddr;                    g:
    patch(i, g);
    com.C2.visit(this, arg);                    execute C2
    short h = nextInstrAddr;                    h:
    patch(j, nextInstrAddr);
    return null;
}
```

Here we have used the following auxiliary method for patching instructions:

```
private void patch (short addr, short d) {
    // Store d in the operand field of the instruction at address addr.
    code[addr].d = d;
}
```

□

7.3 Constants and variables

In a source program, the role of each declaration is to bind an identifier I to some ***entity***, such as a value, variable, or procedure. Within the scope of its declaration, there may be many applied occurrences of I in expressions, commands, and so on. Each applied occurrence of I denotes the entity to which I was bound.

In an object program, each entity will have a suitable representation, which is decided by the code generator. Identifiers will not themselves occur in the object program. Instead, the code generator translates each applied occurrence of an identifier to (the representation of) the corresponding entity.

How the code generator handles identifiers and declarations is the topic of this section and Section 7.4. Here we concentrate on declarations and applied occurrences of constants and variables, and the closely related topic of storage allocation. In Section 7.4 we go on to consider procedures, functions, and parameters.

7.3.1 Constant and variable declarations

A constant declaration binds an identifier to an ordinary value (such as a truth-value, integer, or record). We studied the representation of values of various types in Section 6.1.

A variable declaration allocates a variable and binds an identifier to it. A variable will be represented by one or more consecutive storage cells, based at a particular data address.

The code generator, when it visits a constant or variable declaration, must decide how to represent the declared entity (as a value or address). It should create an ***entity description***, containing details of how the declared entity will be represented, and bind the identifier to that entity description for future reference. The following example illustrates the idea, and also suggests a simple method by which the code generator can represent the binding of identifiers to entity descriptions.

Example 7.10 Accessing a known value and known address

Consider the following Mini-Triangle command:

```
let
   const b ~ 10;
   var i: Integer
in
   i := i * b
```

Figure 7.1(a) shows the decorated AST representing this command. The sub-AST (1) represents the declaration of `b`, and the applied occurrence of `b` at (5) has been linked to (1). The sub-AST (2) represents the declaration of `i`, and the applied occurrences of `i` at (3) and (4) have been linked to (2).

The constant declaration binds the identifier `b` to the integer value 10. The variable declaration binds the identifier `i` to a newly allocated integer variable, whose address must be decided by the code generator. To be concrete, let us suppose that this address is 4 (relative to SB).

Each applied occurrence of b should be translated to the value 10 (more precisely, to the target-machine representation of 10), and each applied occurrence of i should be translated to the address 4. So the subcommand 'i := i * b' should be translated to the following object code:

```
LOAD  4[SB]  – fetch from the address bound to i
LOADL 10     – fetch the value bound to b
CALL  mult   – multiply
STORE 4[SB]  – store to the address bound to i
```

Now let us see how this treatment of identifiers can be achieved. The code generator first visits the declarations. It creates an entity description for the known value 10, and attaches that entity description to the declaration of b at (1). It creates an entity description for the known address 4, and attaches that entity description to the declaration of i at (2). Figure 7.1(b) shows the AST at this point.

Thereafter, when the code generator encounters an applied occurrence of b, it follows the link to the declaration (1). From the entity description attached to (1) it determines that b denotes the known value 10. Likewise, when the code generator encounters an applied occurrence of i, it follows the link to the declaration (2). From the entity description attached to (2) it determines that i denotes the known address 4. □

Example 7.11 Accessing an unknown value

Consider the following Mini-Triangle command:

```
let var x: Integer
in
   let const y ~ 365 + x
   in
      putint(y)
```

Figure 7.2 shows the decorated AST representing this command. The applied occurrences of x and y at (3) and (4) have been linked to the corresponding declarations at (1) and (2), respectively.

The variable declaration binds the identifier x to a newly allocated integer variable. To be concrete, let us suppose that its address is 5 (relative to SB).

The constant declaration binds the identifier y to an integer value that is *unknown* at compile-time. So the code generator cannot simply translate an applied occurrence of y to the value that it denotes. (Contrast the constant declaration of Example 7.10.)

Fortunately, there is a simple solution to this problem. The code generator translates the constant declaration to object code that evaluates the unknown value and stores it *at a known address*. Suppose that the value of y is to be stored at address 6 (relative to SB). Then the applied occurrence of y in 'putint(y)' should be translated to an instruction to fetch the value contained at address 6:

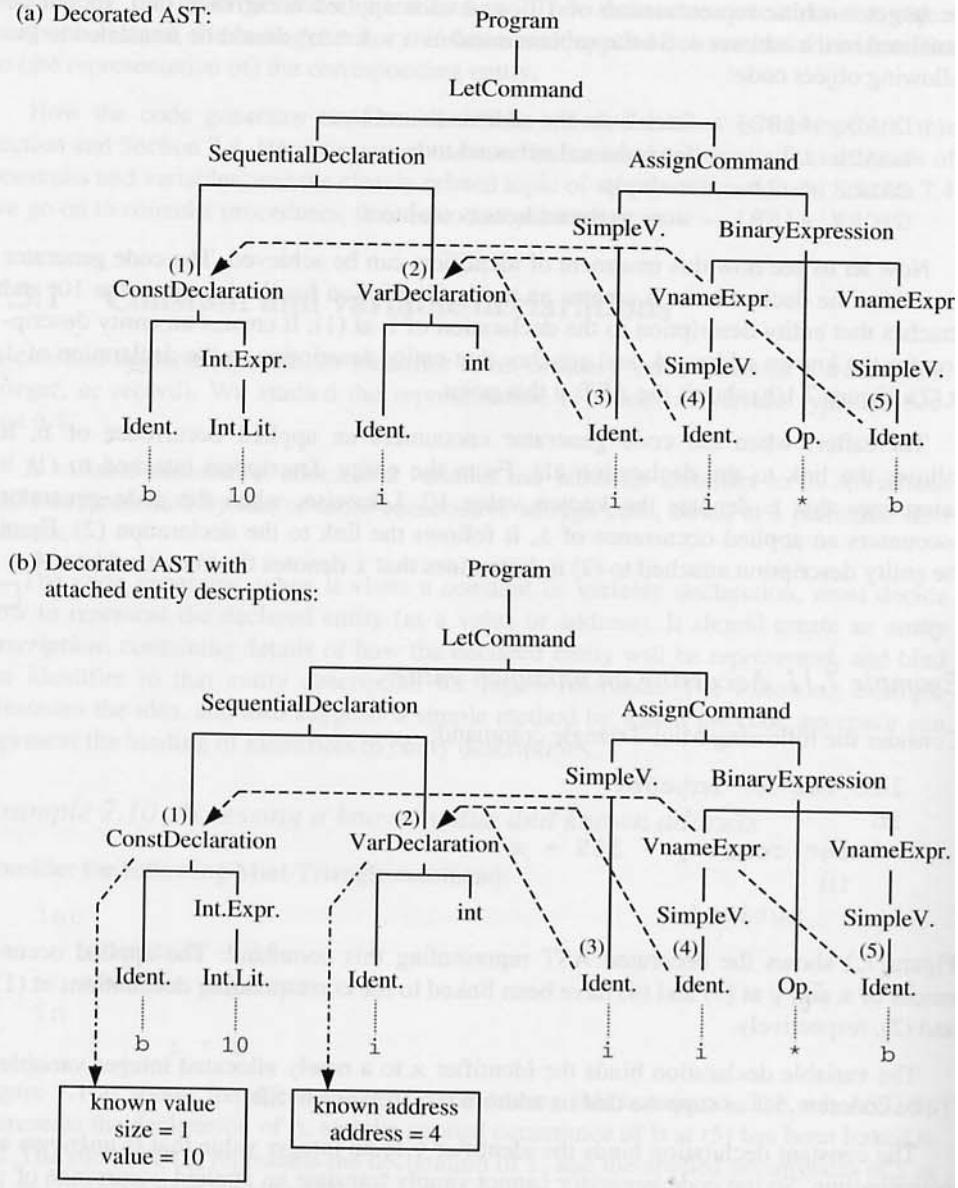

Figure 7.1 Entity descriptions for a known value and a known address.

```
LOAD   6[SB]      – fetch the value bound to y
CALL   putint     – write it
```

The code generator first visits the declarations. It creates an entity description for the known address 5, and attaches that entity description to the declaration of x at (1). It creates an entity description for an unknown value at address 6, and attaches that entity description to the declaration of y at (2). These entity descriptions are shown in Figure 7.2.

Thereafter, whenever the code generator encounters an applied occurrence of y, it follows the link to the declaration (2). From the entity description attached to (2) it determines that y denotes the unknown value contained at address 6.

□

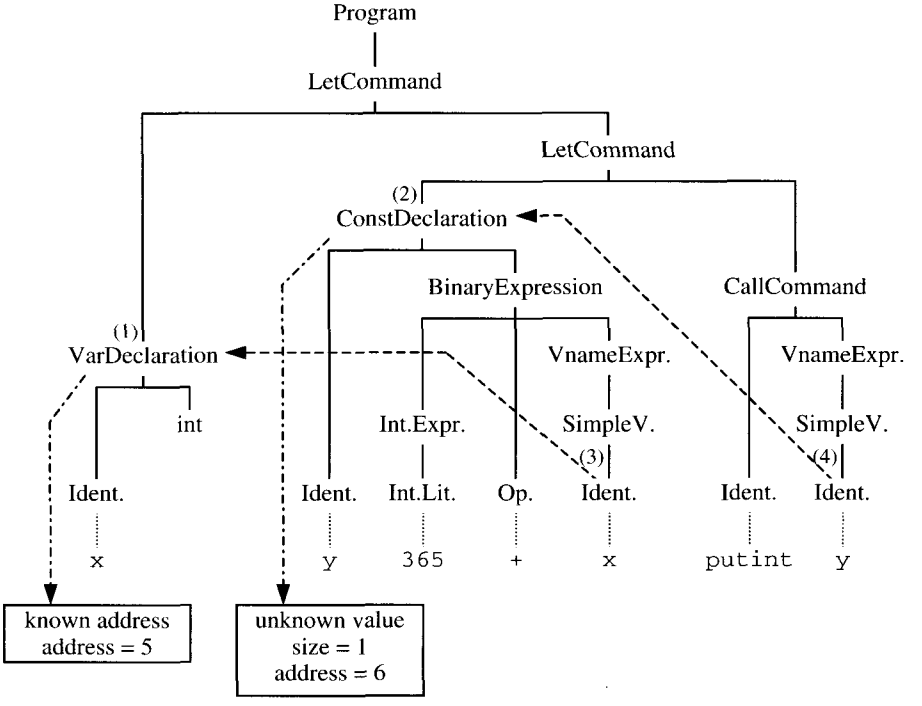

Figure 7.2 Entity descriptions for a known address and an unknown value.

In summary, the code generator handles declarations and applied occurrences of identifiers as follows:

- When it encounters a declaration of identifier *I*, the code generator binds *I* to a newly created entity description. This entity description contains details of the entity bound to *I*.

- When it encounters an applied occurrence of identifier *I*, the code generator consults the entity description bound to *I*, and translates the applied occurrence of *I* to the described entity.

If the source program is represented by a decorated AST, there is a particularly simple way to bind an identifier *I* to an entity description: simply attach the entity description to the sub-AST that represents the declaration of *I*. Every applied occurrence of *I* has already been linked to the corresponding declaration of *I*. So, whenever the code generator encounters an applied occurrence of *I*, it follows the link to the declaration of *I*, and there it finds the attached entity description.

In declarations, identifiers may be bound to entities such as *values* and *addresses*. Each entity may be either *known* or *unknown* (at compile-time). All combinations are possible, and all actually occur in some languages:

- ***Known value:*** This describes a value bound in a constant declaration whose right side is a literal.
- ***Unknown value:*** This describes a value bound in a constant declaration whose right side must be evaluated at run-time, or an argument value bound to a constant parameter.
- ***Known address:*** This describes an address allocated and bound in a variable declaration.
- ***Unknown address:*** This describes an argument address bound to a variable parameter.

(Constant and variable parameters will be discussed in Section 7.4.3.)

We can systematically deal with both known and unknown entities by the techniques illustrated in Examples 7.10 and 7.11. In general:

- If an identifier *I* is bound to a *known* entity, the code generator creates an entity description containing that known entity, and attaches that entity description to the declaration of *I*. It translates each applied occurrence of *I* to that known entity.
- If an identifier *I* is bound to an *unknown* entity, the code generator generates code to evaluate the unknown entity and store it at a known address, creates an entity description containing that known address, and attaches that entity description to the declaration of *I*. At each applied occurrence of *I*, the code generator generates code to fetch the unknown entity from the known address.

An important task for the code generator is to allocate addresses for variables (and unknown values). We study this topic in the following subsections. We shall take advantage of the constant-size requirement explained in Section 6.1: given the type of a variable, the code generator knows exactly how much storage must be allocated for it.

7.3.2 Static storage allocation

Consider a source language with only global variables. As explained in Section 6.3, static storage allocation is appropriate for such a language. The code generator can determine the exact address of every variable in the source program.

Example 7.12 Addressing global variables

Consider the following Mini-Triangle program:

```
let
   var a: Integer;
   var b: Boolean;
   var c: Integer;
   var d: Integer
in
   begin
   ...
   end
```

If the target machine is TAM, each variable of type `Boolean` or `Integer` will occupy one word. If instead the target machine is the x86, each variable of type `Boolean` will occupy one byte, and each variable of type `Integer` will occupy one half-word (= 2 bytes). The following table summarizes the allocated addresses in each case:

Variable	TAM address (words)	x86 address (bytes)
a	0	0
b	1	2
c	2	3
d	3	5

Now consider the following Mini-Triangle program with nested blocks:

```
let var a: Integer
in
   begin
   ...;
   let var b: Boolean;
       var c: Integer
   in
      begin ... end;
   ...;
   let var d: Integer
   in
      begin ... end;
   ...
   end
```

Now the variables `b` and `c` can safely occupy the same storage as the variable `d`, since they can never coexist. The following table summarizes the allocated addresses when the target machine is TAM or the x86:

Variable	TAM address (words)	x86 address (bytes)
a	0	0
b	1	2
c	2	3
d	1	2

□

The code generator must keep track of how much storage has been allocated at each point in the source program. We can arrange this by using the additional `Object` argument of each visitor/encoding method to indicate how much storage is already in use. Since elaborating a declaration may allocate extra storage, we use the `Object` result of a declaration's visitor/encoding method to pass back the amount of extra storage it has allocated. We also use the `Object` result of an expression's visitor/encoding method to pass back the size of the expression's result. In both cases, the result will be of class `Short`.

Example 7.13 Static storage allocation in the Mini-Triangle code generator

We define entity descriptions as follows:

```
public abstract class RuntimeEntity {
    public short size;
    ...
}

public class KnownValue extends RuntimeEntity {
    public short value;  // the known value itself
    ...
}

public class UnknownValue extends RuntimeEntity {
    public short address;  // the address where the
                           // unknown value is stored
    ...
}

public class KnownAddress extends RuntimeEntity {
    public short address;  // the known address itself
    ...
}
```

Each of these classes should be equipped with a suitable constructor.

In addition, to each nonterminal node of the AST we add a field `entity`, which is initially **`null`** but can later be updated to point to an entity description:

```
public abstract class AST {
    ...
    public RuntimeEntity
           entity;   // used in declaration nodes, mainly
}
```

In the Mini-Triangle code generator, we enhance the visitor/encoding methods as follows:

```
public Object visit...Command
                  (...Command com, Object arg) {
    short gs = shortValueOf(arg);
    ...       // Generate code as specified by 'execute com'.
              // gs is the amount of global storage already in use.
    return null;
}

public Object visit...Expression
                  (...Expression expr, Object arg) {
    short gs = shortValueOf(arg);
    ...       // Generate code as specified by 'evaluate expr'.
              // gs is the amount of global storage already in use.
    return new Short(size of expr's result);
}

public Object visit...Declaration
                  (...Declaration decl, Object arg) {
    short gs = shortValueOf(arg);
    ...       // Generate code as specified by 'elaborate decl'.
              // gs is the amount of global storage already in use.
    return new Short(amount of extra storage allocated by decl);
}
```

Here and elsewhere, the following auxiliary method proves useful:

```
private static short shortValueOf (Object obj) {
    return ((Short) obj).shortValue();
}
```

Recall the code templates for declarations (7.12a), (7.17), (7.15), and (7.12c):

elaborate ⟦`const` *I* ~ *E*⟧ =
    evaluate E

elaborate ⟦`const` *I* ~ *IL*⟧ = (i.e., no code) (special case)

elaborate ⟦`var` *I* : *T*⟧ =
    `PUSH` *s* where *s* = *size T*

elaborate ⟦D_1 ; D_2⟧ =
    elaborate D_1
    elaborate D_2

These are implemented by the following visitor/encoding methods:

```
public Object visitConstDeclaration
                (ConstDeclaration decl,
                 Object arg) {
   short gs = shortValueOf(arg);
   if (decl.E instanceof
         IntegerExpression) {
      IntegerLiteral IL =
         ((IntegerExpression) decl.E).IL;
      decl.entity = new KnownValue
            (1, valuation(IL.spelling));
      return new Short(0);
   } else {
      short s = shortValueOf(
            decl.E.visit(this, arg));
      decl.entity = new UnknownValue
            (s, gs);
      return new Short(s);
   }
}
```

elaborate ⟦`cons` *I* ~ *IL*⟧ :
(no code)

elaborate ⟦`cons` *I* ~ *E*⟧
evaluate E

```
public Object visitVarDeclaration
                (VarDeclaration decl,
                 Object arg) {
   short gs = shortValueOf(arg);
   short s = shortValueOf(decl.T.visit
            (this, null));
   emit(Instruction.PUSHop, 0, 0, s);
   decl.entity = new KnownAddress
         (1, gs);
   return new Short(s);
}
```

elaborate ⟦`var` *I* : *T*⟧

s = *size T*
`PUSH` *s*

```
public Object visitSequentialDeclaration
            (SequentialDeclaration decl,
             Object arg) {
   short gs = shortValueOf(arg);
   short s1 = shortValueOf(
         decl.D1.visit(this, arg));
```

elaborate ⟦D_1 ; D_2⟧ =

elaborate D_1

```
    short s2 = shortValueOf(
          decl.D2.visit(this,                          elaborate D2
             new Short (gs + s1)));
    return new Short(s1 + s2);
}
```

The statement 'decl.entity = **new** KnownAddress(...);' creates an entity description for a known address and attaches it to the declaration node in the AST. Entity descriptions for known and unknown values are created and attached to the AST in an analogous way.

Recall the code template for a let-command (7.8f):

execute ⟦let *D* in *C*⟧ =
    elaborate D
    execute C
    POP(0) *s* if $s > 0$
    where s = amount of storage allocated by *D*

The corresponding visitor/encoding method in Example 7.8 omitted one important detail: how does it determine the amount of storage allocated by *D*? We can now see that this information is supplied by the visitor/encoding method for a declaration:

```
public Object visitLetCommand                    execute ⟦let D
                (LetCommand com,                        in C⟧ =
                 Object arg) {
    short gs = shortValueOf(arg);
    short s = shortValueOf(
          com.D.visit(this, arg));                   elaborate D
    com.C.visit(this, new Short (gs + s));           execute C
    if (s > 0)                                       if s > 0
       emit(Instruction.POPop, 0, 0, s);               POP(0) s
    return null;
}
```

Now recall the code templates for value-or-variable-names, namely (7.14), (7.16), and (7.13):

assign ⟦*I*⟧ =
    STORE(*s*) *d*[SB] where s = *size*(type of *I*),
        d = address bound to *I* (relative to SB)

fetch ⟦*I*⟧ = (special case)
    LOADL *v* where v = value bound to *I* (if known)

fetch ⟦*I*⟧ =
    LOAD(*s*) *d*[SB] where s = *size*(type of *I*),
        d = address bound to *I* (relative to SB)

These are implemented by the following encoding methods:

```
private void encodeAssign (Vname vname, short s) {
   RuntimeEntity entity =
         (RuntimeEntity) vname.visit(this, null);
   short d = ((KnownAddress) entity).address;
   emit(Instruction.STOREop, s, Instruction.SBr, d);
}

private void encodeFetch (Vname vname, short s) {
   RuntimeEntity entity =
         (RuntimeEntity) vname.visit(this, null);
   if (entity instanceof KnownValue) {
      short v = ((KnownValue) entity).value
      emit(Instruction.LOADLop, 0, 0, v);
   } else {
      short d = (entity instanceof UnknownValue) ?
            ((UnknownValue) entity).address :
            ((KnownAddress) entity).address;
      emit(Instruction.LOADop, s, Instruction.SBr, d);
   }
}
```

In `encodeAssign` we can safely assume that `entity` is an instance of `KnownAddress`. (The contextual analyzer will already have checked that *I* is a variable identifier.) In `encodeFetch`, however, `entity` could be an instance of `KnownValue`, `UnknownValue`, or `KnownAddress`.

Both `encodeFetch` and `encodeAssign` visit `vname`. The corresponding visitor method simply returns the corresponding entity description:

```
public Object visitSimpleVname
                  (SimpleVname vname, Object arg) {
   return vname.I.decl.entity;
}
```

(Recall that the contextual analyzer has linked each applied occurrence of identifier *I* to the corresponding declaration of *I*. The field `decl` represents this link. Therefore, `I.decl.entity` points to the entity description bound to *I*.)

Finally, method `encode` starts off code generation with no storage allocated:

```
public void encode (Program prog) {
   prog.visit(this, new Short(0));
}
```

7.3.3 Stack storage allocation

Consider now a source language with procedures and local variables. As explained in Section 6.4, stack storage allocation is appropriate for such a language. The code generator cannot predict a local variable's absolute address, but it can predict the variable's address displacement relative to the base of a frame – a frame belonging to the procedure within which the variable was declared. At run-time, a display register will point to the base of that frame, and the variable can be addressed relative to that register. The appropriate register is determined entirely by a pair of routine levels known to the code generator: the routine level of the variable's declaration, and the routine level of the code that is addressing the variable. (See Section 6.4.2 for details.)

To make the code generator implement stack storage allocation, we must modify the form of addresses in entity descriptions. The address of a variable will now be held as a pair (l, d), where l is the routine level of the variable's declaration, and d is the variable's address displacement relative to its frame base. As in Section 6.4.2, we assign a routine level of 0 to the main program, a routine level of 1 to the body of each procedure or function declared at level 0, a routine level of 2 to the body of each procedure or function declared at level 1, and so on.

Example 7.14 Storage allocation for global and local variables

Recall the Triangle program of Figure 6.14. The same program is outlined in Figure 7.3, with each procedure body shaded to indicate its routine level.

Entity descriptions are shown attached to the variable declarations in the source program. (This is for clarity. In reality, of course, the entity descriptions would be attached to the sub-ASTs that represent these declarations, as in Figures 7.1 and 7.2.)

The addresses of the global variables `g1` and `g2` are shown as (0, 0) and (0, 1), meaning displacements of 0 and 1, respectively, relative to the base of the level-0 frame (i.e., the globals).

The addresses of the local variables `p1` and `p2` are shown as (1, 3) and (1, 4), meaning displacements of 3 and 4, respectively, relative to the base of a level-1 frame. The address of the local variable `q` is shown as (2, 3), meaning a displacement of 3 relative to the base of a level-2 frame. And so on.

Notice that the address displacements of local variables start at 3. The reason is that the first three words of a frame contain link data, as shown in Figure 6.16.

□

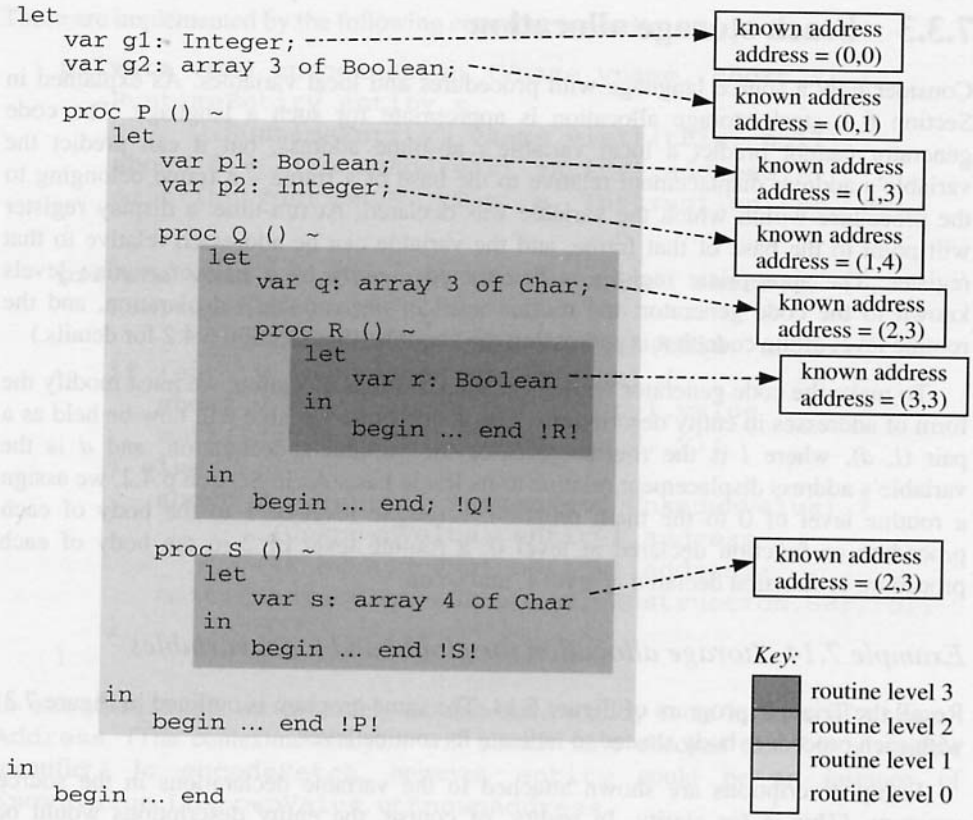

Figure 7.3 Entity descriptions in the presence of stack allocation.

The code templates (7.13) and (7.14) assumed static storage allocation. They must be modified to take account of stack storage allocation.

Example 7.15 Code templates for global and local variables

Although Mini-Triangle has no procedures, let us anticipate their introduction – just in order to study the code generator's treatment of local variables.

The code templates for *fetch* (7.13) and *assign* (7.14) would be generalized as follows:

$$
\begin{aligned}
&\textit{fetch}\ [\![I]\!] = \\
&\qquad \texttt{LOAD}(s)\ \ d[r] \qquad \text{where } s = \textit{size}(\text{type of } I),\\
&\qquad\qquad\qquad\qquad\quad (l, d) = \text{address bound to } I,\\
&\qquad\qquad\qquad\qquad\quad cl = \text{current routine level},\\
&\qquad\qquad\qquad\qquad\quad r = \textit{display-register}(cl, l)
\end{aligned}
\tag{7.18}
$$

$$\textit{assign}\ [\![I]\!] = \quad \texttt{STORE}(s)\ \ d[r] \qquad \text{where } s = \textit{size}(\text{type of } I), \quad (l, d) = \text{address bound to } I, \quad cl = \text{current routine level}, \quad r = \textit{display-register}(cl, l) \tag{7.19}$$

The current routine level, cl, is the routine level of the code that is addressing the variable.

The auxiliary function *display-register*(cl, l) selects the display register that will enable code at routine level cl to address a variable declared at routine level l:

$$\textit{display-register}(cl, l) = \begin{cases} \text{SB} & \text{if } l = 0 & (7.20a) \\ \text{LB} & \text{if } l > 0 \text{ and } cl = l & (7.20b) \\ \text{L1} & \text{if } l > 0 \text{ and } cl = l + 1 & (7.20c) \\ \text{L2} & \text{if } l > 0 \text{ and } cl = l + 2 & (7.20d) \\ \ldots & & \end{cases}$$

Note that the special-case code template (7.16) is unaffected.

□

In order to implement (7.18) and (7.19), the code generator must know the routine level of each command, expression, and so on. Previously, all variables were allocated at level 0, and the argument passed to each encoding method was simply the size of the global frame. Now the argument must include both the level and the size of the current frame. For this purpose let us introduce a `Frame` class with appropriate instance variables:

```
public class Frame {
    public byte  level;
    public short size;
    ...
}
```

Example 7.16 Stack storage allocation in the Mini-Triangle code generator

We generalize entity descriptions as follows:

```
public abstract class RuntimeEntity {
    public short size;
    ...
}

public class KnownValue extends RuntimeEntity {
    public short value;                    // the known value itself
    ...
}
```

```
public class UnknownValue extends RuntimeEntity {
    public EntityAddress address;     // the address where the
    ...                               // unknown  value is stored
}

public class KnownAddress extends RuntimeEntity {
    public EntityAddress address;     // the known address itself
    ...
}

public class EntityAddress {
    public byte  level;
    public short displacement;
    ...
}
```

In the Mini-Triangle code generator, we enhance the visitor/encoding methods as follows:

```
public Object visit...Command
                (...Command com, Object arg) {
    Frame frame = (Frame) arg;
    ...     // Generate code as specified by 'execute com'.
            // frame.level is the routine level of com.
            // frame.size is the amount of frame storage already in use.
    return null;
}

public Object visit...Expression
                (...Expression expr, Object arg) {
    Frame frame = (Frame) arg;
    ...     // Generate code as specified by 'evaluate expr'.
            // frame.level is the routine level of expr.
            // frame.size is the amount of frame storage already in use.
    return new Short(size of expr's result);
}

public Object visit...Declaration
                (...Declaration decl, Object arg) {
    Frame frame = (Frame) arg;
    ...     // Generate code as specified by 'elaborate decl'.
            // frame.level is the routine level of decl.
            // frame.size is the amount of frame storage already in use.
    return new Short(amount of extra storage allocated by decl);
}
```

We can provide `encodeAssign` and `encodeFetch` with explicit `Frame` arguments:

```
private void encodeAssign
            (Vname vname, Frame frame, short s) {
    ...     // Generate code as specified by 'assign vname'.
            // frame.level is the routine level of vname.
            // s is the size of the value to be assigned.
}

private void encodeFetch
            (Vname vname, Frame frame, short s) {
    ...     // Generate code as specified by 'fetch vname'.
            // frame.level is the routine level of vname.
            // s is the size of the value to be assigned.
}
```

The following method implements code template (7.19):

```
private void encodeAssign
            (Vname vname, Frame frame, short s) {
    RuntimeEntity entity =
          (RuntimeEntity) vname.visit(this, null);
    EntityAddress address =
        ((KnownAddress) entity).address;
    emit(Instruction.STOREop, s,
          displayRegister(frame.level, address.level),
             address.displacement);
}
```

The following method implements code templates (7.16) and (7.18):

```
private void encodeFetch (Vname vname,
                Frame frame, short s) {
    RuntimeEntity entity =
          (RuntimeEntity) vname.visit(this, null);
    if (entity instanceof KnownValue) {
       short v = ((KnownValue) entity).value;
       emit(Instruction.LOADLop, 0, 0, v);
    } else {
       EntityAddress address =
             (entity instanceof UnknownValue) ?
             ((UnknownValue) entity).address :
             ((KnownAddress) entity).address;
       emit(Instruction.LOADop, s,
             displayRegister(frame.level,
                address.level), address.displacement);
    }
}
```

The following auxiliary method `displayRegister` implements equations (7.20):

```
private byte displayRegister
                (byte currentLevel, byte entityLevel)
{ ... }
```

The following methods show how the entity descriptions are now set up:

```
public Object visitConstDeclaration
                (ConstDeclaration decl, Object arg) {
   Frame frame = (Frame) arg;
   if (decl.E instanceof IntegerExpression) {
      IntegerLiteral IL =
         ((IntegerExpression) decl.E).IL;
      decl.entity = new KnownValue
         (1, valuation(IL.spelling));
      return new Short(0);
   } else {
      short s =
         shortValueOf(decl.E.visit(this, frame));
      decl.entity = new UnknownValue
            (s, frame.level, frame.size);
      return new Short(s);
   }
}

public Object visitVarDeclaration
                (VarDeclaration decl, Object arg) {
   Frame frame = (Frame) arg;
   short s = shortValueOf(decl.T.visit(this, null));
   emit(Instruction.PUSHop, 0, 0, s);
   decl.entity = new KnownAddress
         (1, frame.level, frame.size);
   return new Short(s);
}
```

When the appropriate visitor/encoding method is called to translate a procedure body, the frame level must be incremented by one and the frame size set to 3, leaving just enough space for the link data:

```
Frame outerFrame = ...;
Frame localFrame = new Frame(outerFrame.level + 1, 3);
```

Finally, method encode starts off with a frame at level 0 and with no storage allocated:

```
public void encode (Program prog) {
   Frame globalFrame = new Frame(0, 0);
   prog.visit(this, globalFrame);
}
```

□

7.4 Procedures and functions

In this section we study how the code generator handles procedure and function declarations, procedure and function calls, and the association between actual and formal parameters. We start by looking at global procedures and functions. Then we consider nested procedures and functions. Finally we examine the implementation of parameter mechanisms.

A procedure declaration binds an identifier to a procedure, and a function declaration binds an identifier to a function. The run-time representation of a procedure or function is a *routine*. At its simplest, a routine is just a sequence of instructions with a designated entry address.

7.4.1 Global procedures and functions

Consider a programming language in which all procedures are declared globally. In the implementation of such a language, a routine is completely characterized by its *entry address* (i.e., the address of its first instruction). The routine is called by a call instruction that designates the entry address. This instruction will pass control to the routine, where control will remain until a return instruction is executed.

From the above, we see that the code generator should treat a procedure declaration as follows. It should create an entity description for a known routine, containing the routine's entry address, and bind that entity description to the procedure identifier. At an applied occurrence of this identifier, in a procedure call, the code generator should retrieve the corresponding routine's entry address, and generate a call instruction designating that entry address.

Example 7.17 Code templates for Mini-Triangle plus global procedures

Consider again the language Mini-Triangle, for which code templates were given in Example 7.2. Let us now extend Mini-Triangle with parameterless procedures. The syntactic changes, for procedure declarations and procedure calls, are as follows:

```
Declaration   ::=  ...
                |  proc Identifier ( ) ~ Command                (7.21)

Command       ::=  ...
                |  Identifier ( )                               (7.22)
```

We shall assume that all procedure declarations are global, not nested.

The code template specifying translation of a procedure declaration to TAM code would be:

elaborate ⟦`proc` *I* `( ) ~` *C*⟧ = (7.23)

```
        JUMP g
   e:   execute C
        RETURN(0) 0
   g:
```

The generated routine body consists simply of the object code '*execute C*' followed by a `RETURN` instruction. The two zeros in the `RETURN` instruction indicate that the routine has no result and no arguments. Since we do not want the routine body to be executed at the point where the procedure is *declared*, only where the procedure is *called*, we must generate a jump round the routine body. The routine's entry address, *e*, must be bound to *I* for future reference.

The code template specifying translation of a procedure call would be:

execute ⟦*I* `( )` ⟧ = (7.24)

```
        CALL(SB)  e        where e = entry address of routine bound to I
```

This is straightforward. The net effect of executing this `CALL` instruction will be simply to execute the body of the routine bound to *I*.

□

Example 7.18 Object code for Mini-Triangle plus global procedures

The following extended Mini-Triangle program illustrates a procedure declaration and call:

```
let
   var n: Integer;

   proc p () ~
      n := n * 2

in
   begin
   n := 9;
   p()
   end
```

The corresponding object program illustrates code templates (7.23) and (7.24):

elaborate ⟦`var n: Integer`⟧

elaborate ⟦`proc p () ~ n := n*2`⟧ — *execute* ⟦`n := n*2`⟧

```
0:  PUSH        1
1:  JUMP        7
2:  LOAD        0[SB]
3:  LOADL       2
4:  CALL        mult
5:  STORE       0[SB]
6:  RETURN(0)   0
```

execute ⟦`begin n := 9; p() end`⟧ {
  execute ⟦`n := 9`⟧ {
    7: LOADL 9
    8: STORE 0[SB]
  execute ⟦`p()`⟧ {
    9: CALL(SB) 2
    10: POP(0) 1
    11: HALT

The corresponding decorated AST and entity descriptions are shown in Figure 7.4.

□

A function is translated in much the same way as a procedure. The only essential difference is in the code that returns the function result.

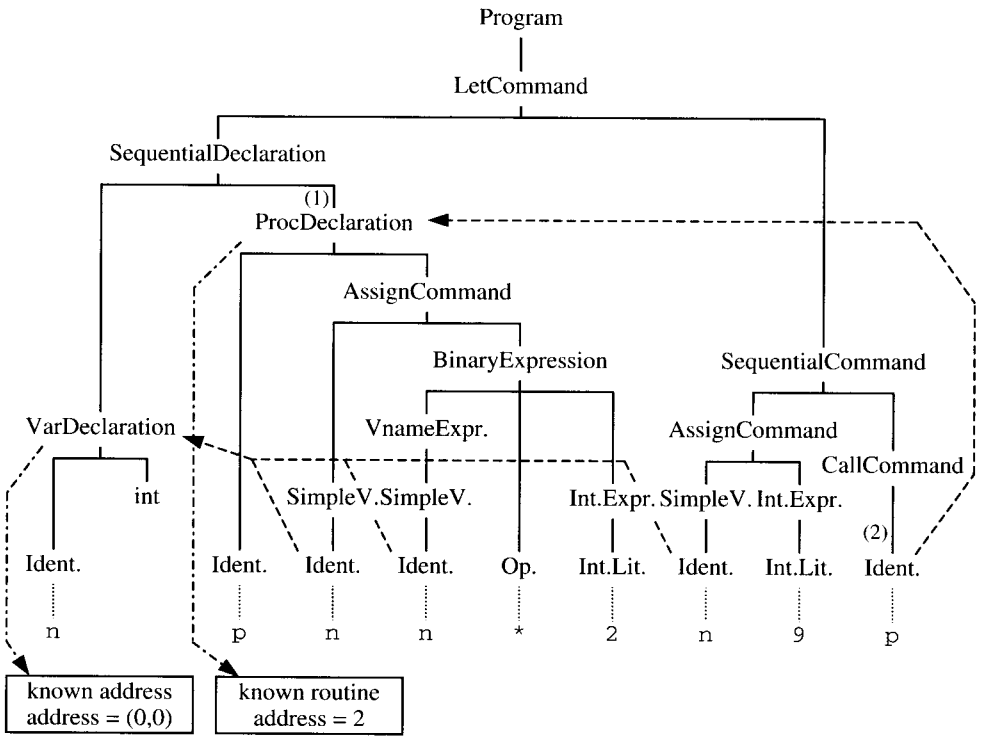

Figure 7.4 Entity description for a known routine.

Example 7.19 Code templates for Mini-Triangle plus global functions

Suppose that Mini-Triangle is to be extended with parameterless functions. The syntactic changes are as follows:

```
Declaration   ::=  ...
                |  func Identifier ( ) : Type-denoter ~ Expression   (7.25)

Expression    ::=  ...
                |  Identifier ( )                                   (7.26)
```

As in Example 7.17, we shall assume that all function declarations are global.

The code template specifying translation of a function declaration to TAM code would be:

```
elaborate [[func I ( ) : T ~ E]] =                                  (7.27)
        JUMP g
    e:  evaluate E
        RETURN(s)  0        where s = size T
    g:
```

This RETURN instruction returns a result of size s, that result being the value of E. The function has no arguments, so the RETURN instruction removes 0 words of arguments from the stack.

The code template specifying translation of a function call to TAM code would be:

```
evaluate [[I ( )]] =                                                (7.28)
        CALL(SB)  e         where e = entry address of routine bound to I
```

which is similar to (7.24).

□

7.4.2 Nested procedures and functions

Now consider a source language that allows procedures and functions to be nested, and allows them to access nonlocal variables. In this case the implementation needs static links, as explained in Section 6.4.2. The call instruction (or instruction sequence) must designate not only the entry address of the called routine but also an appropriate static link.

Suppose that a procedure is represented by a routine R in the object code. R's entry address is known to the code generator, as we have already seen. The appropriate static link for a call to R will be the base address of a frame somewhere in the stack. This base address is not known to the code generator. But the code generator can determine which display register will contain that static link, at the time when R is called. The appropriate register is determined entirely by a pair of routine levels known to the code generator: the routine level of R's declaration and the routine level of the code that calls R.

The address of routine *R* must, therefore, be held as a pair (*l*, *e*), where *l* is the routine level of *R*'s declaration (with global routines at level 0), and *e* is *R*'s entry address.

Example 7.20 Nested procedures

The Triangle program outline of Figure 7.3 is reproduced in Figure 7.5, with entity descriptions shown attached to the procedure declarations.

The entity description for procedure `P` describes it as a known routine with address (0, 3), signifying that `P` was declared at level 0, and its entry address is 3. The entity description for procedure `Q` describes it as a known routine with address (1, 6), signifying that `Q` was declared at level 1, and its entry address is 6. And so on.

□

The code template (7.24) in Example 7.17 assumed global procedures only. It must be modified to take account of nested procedures.

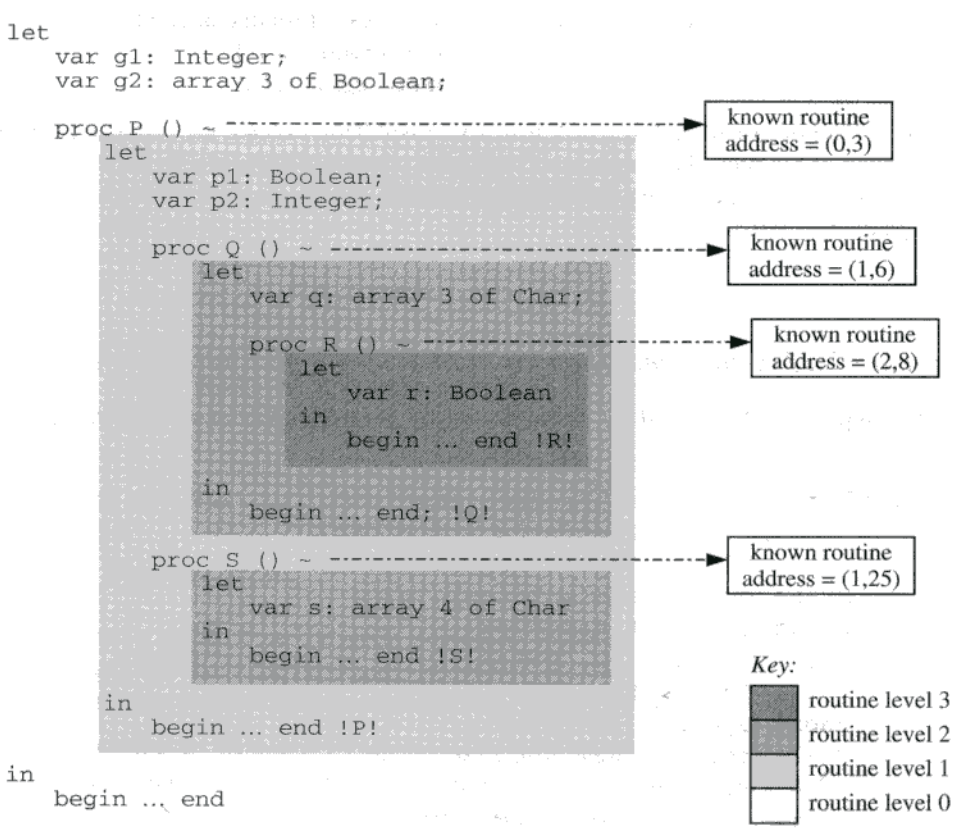

Figure 7.5 Entity descriptions for nested routines.

Example 7.21 Code templates for Mini-Triangle plus nested procedures

Consider again the language Mini-Triangle extended with parameterless procedures. The syntax is unchanged from Example 7.17, but now we shall allow nested procedure declarations.

The code template for a procedure declaration is unchanged:

elaborate ⟦`proc` *I* `(` `)` ~ *C*⟧ = (7.29)

```
        JUMP g
  e:    execute C
        RETURN(0) 0
  g:
```

but now the entity description bound to *I* must include the address pair (*l*, *e*), where *l* is the current routine level, and *e* is the entry address.

The code template for a procedure call becomes:

execute ⟦*I* `(` `)` ⟧ = (7.30)

```
        CALL(r)  e
```

where (*l*, *e*) = address of routine bound to *I*,
cl = current routine level,
r = *display-register*(*cl*, *l*)

The net effect of executing this `CALL` instruction will be to execute the command *C* that is the body of the procedure bound to *I*, using the content of register *r* as the static link. The latter is determined using the auxiliary function *display-register*, which is defined by equations (7.20).

□

Example 7.22 Code generation for Mini-Triangle plus nested procedures

Code template (7.29) would be implemented by the following new visitor/encoding method:

```
public Object visitProcDeclaration
               (ProcDeclaration decl,
                Object arg) {
  Frame outerFrame = (Frame) arg;
  short j = nextInstrAddr;
  emit(Instruction.JUMPop, 0,
       Instruction.CBr, 0);
  short e = nextInstrAddr;
  decl.entity = new KnownRoutine
       (2, outerFrame.level, e);
  Frame localFrame = new Frame
       (outerFrame.level + 1, 3);
  decl.C.visit(this, localFrame);
```

elaborate ⟦`proc` *I* `(` `)` ~ *C*⟧ =

```
j:
  JUMP g
e:
```

execute C

```
        emit(Instruction.RETURNop, 0, 0, 0);      RETURN(0) 0
        short g = nextInstrAddr;                  g:
        patch(j, g);
        return new Short(0);
    }
```

This assumes a new kind of entity description:

```
    public class KnownRoutine extends RuntimeEntity {
        public EntityAddress address;
        ...
    }
```

where `address.level` is the level of the routine and `address.displacement` is its entry address.

Code template (7.30) would be implemented by the following visitor/encoding method:

```
    public Object visitCallCommand          execute ⟦I ( )⟧ =
                    (CallCommand com,
                     Object arg) {
        Frame frame = (Frame) arg;
        EntityAddress address =
            ((KnownRoutine)
                com.I.decl.entity).address;
        emit(Instruction.CALLop,                  CALL(r) e
                displayRegister(
                    frame.level,
                    address.level),
                Instruction.CBr,
                address.displacement);
        return null;
    }
```

□

7.4.3 Parameters

Now let us consider how the code generator implements parameter passing. Every source language has one or more ***parameter mechanisms***, the means by which arguments are associated with the corresponding formal parameters.

As explained in Section 6.5.1, a routine protocol is needed to ensure that the calling code deposits the arguments in a place where the called routine expects to find them. If the operating system does not impose a routine protocol, the language implementor must design one, taking account of the source language's parameter mechanisms and the target machine architecture.

The routine protocol adopted in TAM is for the calling code to deposit the arguments at the stack top immediately before the call. Thus the called routine can address its own arguments using negative displacements relative to its own frame base. The code generator can represent the address of each argument in the usual way by a pair (l, d), where l is the routine level of the routine's body and d is the negative displacement.

Example 7.23 Addressing parameters

Recall the Triangle program of Example 6.23, whose run-time behavior was shown in Figure 6.21. The same program is reproduced in Figure 7.6, with appropriate entity descriptions attached to the declarations and formal parameters.

The constant parameter `i` will be bound to an argument *value* whenever procedure `S` is called. Therefore its entity description is that of an unknown value, stored at address (1, –1).

The variable parameter `n`, on the other hand, will be bound to an argument *address*. Therefore its entity description is that of an unknown address, which is itself stored at address (1, –2). This entity description implies that each applied occurrence of `n` must be implemented by indirect addressing.

In TAM, indirect addressing is supported by the instructions `LOADI` (load indirect) and `STOREI` (store indirect). These were illustrated in Example 6.23. □

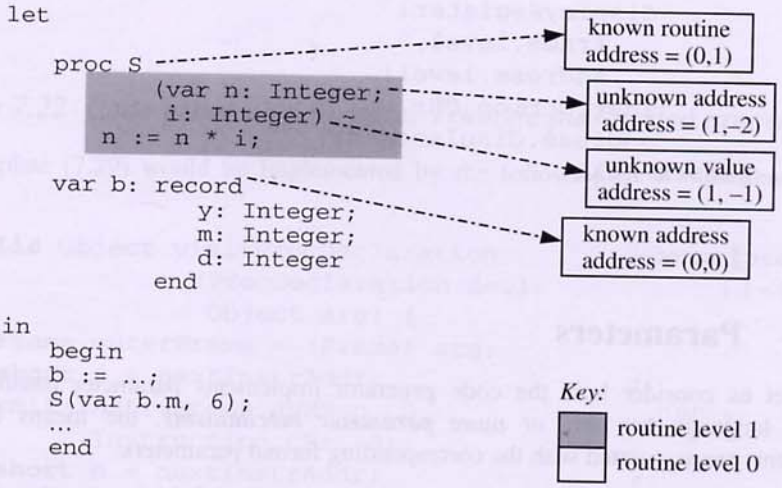

Figure 7.6 Entity descriptions for constant and variable parameters.

Now we have encountered all combinations of known and unknown values, and known and unknown addresses, in entity descriptions. We must therefore generalize our code templates for value-or-variable-names accordingly.

Example 7.24 Code templates for procedures with parameters

Consider Mini-Triangle extended with procedures, and constant and variable parameters. For simplicity we shall assume that each procedure has a single formal parameter. The syntactic changes, for procedure declarations and procedure calls, are as follows:

```
Declaration          ::= ...
                       | proc Identifier ( Formal-Parameter ) ~ Command   (7.31)

Formal-Parameter     ::= Identifier : Type-denoter                        (7.32a)
                       | var Identifier : Type-denoter                    (7.32b)

Command              ::= ...
                       | Identifier ( Actual-Parameter )                  (7.33)

Actual-Parameter     ::= Expression                                       (7.34a)
                       | var V-name                                       (7.34b)
```

Production rules (7.32a) and (7.34a) are concerned with constant parameters; production rules (7.32b) and (7.34b) are concerned with variable parameters.

The code template for a procedure declaration is now:

elaborate ⟦`proc` *I* `(` *FP* `)` ~ *C*⟧ = (7.35)

```
        JUMP g
e:      execute C
        RETURN(0) d        where d = size of formal parameter FP
g:
```

Since the TAM routine protocol requires the *caller* to push the argument on to the stack, the routine body itself contains no code corresponding to the formal parameter *FP*.

The code template specifying translation of a procedure call to TAM code is now:

execute ⟦*I* `(` *AP* `)` ⟧ = (7.36)

```
        pass-argument AP
        CALL(r) e          where (l, e) = address of routine bound to I,
                                cl = current routine level,
                                r = display-register(cl, l)
```

The code templates for actual parameters are:

pass-argument ⟦*E*⟧ = (7.37a)

```
        evaluate E
```

pass-argument ⟦`var` *V*⟧ = (7.37b)

```
        fetch-address V
```

Code template (7.37b) uses a new code function for value-or-variable-names:

fetch-address : V-name → Instruction* (7.38)

where '*fetch-address V*' is code that will push the address of the variable *V* on to the stack top.

The code templates for value-or-variable-names are generalized as follows:

fetch [[*I*]] = (7.39)

(i) if *I* is bound to a known value:

`LOADL` *v* where *v* = value bound to *I*

(ii) if *I* is bound to an unknown value or known address:

`LOAD(`*s*`)` *d*`[`*r*`]` where *s* = *size*(type of *I*),
(*l*, *d*) = address bound to *I*,
cl = current routine level,
r = *display-register*(*cl*, *l*)

(iii) if *I* is bound to an unknown address:

`LOAD(1)` *d*`[`*r*`]`
`LOADI(`*s*`)` where *s* = *size*(type of *I*),
(*l*, *d*) = address bound to *I*,
cl = current routine level,
r = *display-register*(*cl*, *l*)

assign [[*I*]] = (7.40)

(i) if *I* is bound to a known address:

`STORE(`*s*`)` *d*`[`*r*`]` where *s* = *size*(type of *I*),
(*l*, *d*) = address bound to *I*,
cl = current routine level,
r = *display-register*(*cl*, *l*)

(ii) if *I* is bound to an unknown address:

`LOAD(1)` *d*`[`*r*`]`
`STOREI(`*s*`)` where *s* = *size*(type of *I*),
(*l*, *d*) = address bound to *I*,
cl = current routine level,
r = *display-register*(*cl*, *l*)

fetch-address [[*I*]] = (7.41)

(i) if *I* is bound to a known address:

`LOADA` *d*`[`*r*`]` where (*l*, *d*) = address bound to *I*,
cl = current routine level,
r = *display-register*(*cl*, *l*)

(ii) if *I* is bound to an unknown address:

`LOAD(1)` *d*`[`*r*`]` where (*l*, *d*) = address bound to *I*,
cl = current routine level,
r = *display-register*(*cl*, *l*)

□

7.5 Case study: code generation in the Triangle compiler

The Triangle code generator consists of a package `Triangle.CodeGenerator` that contains the `Encoder`, and the classes for the various kinds of run-time entity. The `Encoder` class depends on the package `Triangle.AbstractSyntaxTrees`, which contains all of the class definitions for ASTs, and on the package `TAM`, which contains the definition of the Triangle abstract machine.

The Triangle code generator was designed and implemented using techniques similar to those described in this chapter. Some extensions were necessary to deal with particular features of Triangle. Here we briefly discuss some of these extensions.

7.5.1 Entity descriptions

The Triangle code generator deals with a wide variety of entities and entity descriptions, some of which we have not yet met. The following kinds of entity description are used:

- *Known value:* This describes a value bound in a constant declaration whose right side is a literal, e.g.:

```
const daysPerWeek ~ 7;
const currency    ~ '$'
```

- *Unknown value:* This describes a value bound in a constant declaration, if obtained by evaluating an expression at run-time, e.g.:[5]

```
const area ~ length * breadth;
const nul  ~ chr(0)
```

  It also describes an argument value bound to a constant parameter, e.g., the value bound to `n` in:

```
func odd (n: Integer) : Boolean ~ ...
```

- *Known address:* This describes an address allocated and bound in a variable declaration. The code generator represents each address by a (level, displacement) pair, as described in Section 7.3.3.

- *Unknown address:* This describes an argument address bound to a variable parameter, e.g., the address bound to `n` in:

```
proc inc (var n: Integer) ~ ...
```

---

[5] In principle, `nul` in this example could be treated as bound to a known value. However, the code generator would have to be enhanced to evaluate the expression '`chr(0)`' itself, using a technique called *constant folding*.

- *Known routine:* This describes a routine bound in a procedure or function declaration, e.g., the routines bound to `inc` and `odd` in the above examples.
- *Unknown routine:* This describes an argument routine bound to a procedural or functional parameter, e.g., the routine bound to `p` in:

```
proc filter (func p (x: Integer): Boolean;
             var l: IntegerList) ~ ...
```

- *Primitive routine:* This describes a primitive routine provided by the abstract machine. Primitive routines are bound in the standard environment to operators and identifiers, e.g., to '+', '<', `eof`, and `get`.
- *Equality routine:* This describes one of the primitive routines provided by the abstract machine for testing (in)equality of two values. Equality routines are generic, in that the values can be of any size. Equality routines are bound to the operators '=' and '\='.
- *Field:* This describes a field of a record type. Every record field has a known offset relative to the base of the record (see Section 6.1.2), and the field's entity description includes this offset.
- *Type representation:* This describes a type. Every type has a known size, which is constant for all values of the type (see Section 6.1), and the type's entity description includes that size.

7.5.2 Constants and variables

A value-or-variable-name in the Triangle program identifies a constant or variable. Either a constant or a variable may be used as an expression operand, but only a variable may be used on the left side of an assignment command. These two usages give rise to two different code functions on value-or-variable-names:

fetch : V-name → Instruction*
assign : V-name → Instruction*

In the little language Mini-Triangle used as a running example in this chapter, a value-or-variable-name was just an identifier (declared in a constant or variable declaration). Accordingly, *fetch* was defined by a single code template (plus a special-case code template), and *assign* by a single code template.

More realistic programming languages have composite types, and operations to select components of composite values and variables. In Triangle, a record value-or-variable-name can be subjected to field selection, and an array value-or-variable-name can be indexed.

Example 7.25 Addressing composite variables

Consider the following Triangle declarations:

```
type   Name      = array 15 of Char;
       TelNumber = array 10 of Char;
       Entry     = record
                      name: Name;
                      num:  TelNumber
                   end;
       Directory = record
                      count: Integer;
                      entry: array 100 of Entry
                   end;

var    dir: Directory
```

Now, the following are all value-or-variable-names:

```
dir
dir.count
dir.entry
dir.entry[i]
dir.entry[i].num
dir.entry[i].name
dir.entry[i].name[j]
```

The code generator will compute the following type sizes:

size⟦`Name`⟧	= 15×1	=	15 words
size⟦`TelNumber`⟧	= 10×1	=	10 words
size⟦`Entry`⟧	= $15 + 10$	=	25 words
size⟦`array 100 of Entry`⟧	= 100×25	=	2500 words
size⟦`Directory`⟧	= $1 + 2500$	=	2501 words

It will also compute the offsets of the fields of record type `Entry`:

offset⟦`name`⟧	=	0 words
offset⟦`num`⟧	=	15 words

and those of record type `Directory`:

offset⟦`count`⟧	=	0 words
offset⟦`entry`⟧	=	1 word

As in Section 6.1, we use the notation *address v* for the address of variable (or constant) *v*. For the various components of `dir` we find:

address⟦`dir.count`⟧	= *address*⟦`dir`⟧ + 0
address⟦`dir.entry`⟧	= *address*⟦`dir`⟧ + 1
address⟦`dir.entry[i]`⟧	= *address*⟦`dir`⟧ + 1 + 25*i*

$$
\begin{aligned}
\textit{address}[\![\texttt{dir.entry[i].num}]\!] &= \textit{address}[\![\texttt{dir}]\!] + 1 + 25i + 15 \\
&= \textit{address}[\![\texttt{dir}]\!] + 16 + 25i \\
\textit{address}[\![\texttt{dir.entry[i].name}]\!] &= \textit{address}[\![\texttt{dir}]\!] + 1 + 25i + 0 \\
&= \textit{address}[\![\texttt{dir}]\!] + 1 + 25i \\
\textit{address}[\![\texttt{dir.entry[i].name[j]}]\!] &= \textit{address}[\![\texttt{dir}]\!] + 1 + 25i + j \\
&= \textit{address}[\![\texttt{dir}]\!] + 1 + (25i + j)
\end{aligned}
$$

where i and j are the values of `i` and `j`.

In each case the address formula contains some constant terms. These constant terms are accumulated by the code generator, simplifying the address formula to the sum of (at most) three terms: the address of the entire variable (or constant), plus a known offset, plus an unknown value. The known offset is obtained by adding together the offsets of any record fields. The unknown value is determined by evaluating array indices at run-time.

The following instruction sequences illustrate how the Triangle code generator uses this information. To be concrete, assume that *address*[![dir]!] = (0, 100):

fetch [![dir.count]!]

```
LOAD(1)    100[SB]
```

fetch [![dir.entry[i]]!]

evaluate [![i]!]

```
LOAD(1)    i
LOADL      25
CALL       mult
LOADA      101[SB]
CALL       add
LOADI(25)
```

fetch [![dir.entry[i].num]!]

evaluate [![i]!]

```
LOAD(1)    i
LOADL      25
CALL       mult
LOADA      116[SB]
CALL       add
LOADI(10)
```

In each case, the known offset is added into the displacement part of *address*[![dir]!]. Thus the address arithmetic implied by field selection can be done entirely at compile-time. Only indexing must be deferred until run-time.

□

7.6 Further reading

The target machine architecture strongly influences the structure of a code generator. Once upon a time, machines were designed by engineers with no knowledge of code generation. Such machines often had cunning features that skilled assembly language programmers could exploit, but were very difficult for code generators to exploit. But

now nearly all programs – even operating systems – are written in high-level languages. So it makes more sense for the machine to support the code generator by, for example, providing a simple regular instruction set. A lucid discussion of the interaction between code generation and machine design may be found in Wirth (1986).

Almost all real machines have general-purpose and/or special-purpose registers; some have a stack as well. The number of registers is usually small and always limited. It is quite hard to generate object code that makes effective use of registers. Code generation for register machines is therefore beyond the scope of this introductory textbook. For a thorough treatment, see Chapter 9 of Aho *et al.* (1985).

The code generator described in this chapter works in the context of a multi-pass compiler: it traverses an AST that represents the entire source program. In the context of a one-pass compiler, the code generator would be structured rather differently: it would be a collection of methods, which can be called by the syntactic analyzer to generate code 'on the fly' as the source program is parsed. For a clear account of how to organize code generation in a one-pass compiler, see Welsh and McKeag (1980).

The sheer diversity of machine architectures is a problem for implementors. A common practice among software vendors is to construct a family of compilers, translating a single source language to several different target machine languages. These compilers will have a common syntactic analyzer and contextual analyzer, but a distinct code generator will be needed for each target machine. Unfortunately, a code generator suitable for one target machine might be difficult or impossible to adapt to a dissimilar target machine. *Code generation by pattern matching* is an attractive way to reduce the amount of work to be done. In this method the semantics of each machine instruction is expressed in terms of low-level operations. Each source-program command is translated to a combination of these low-level operations; code generation then consists of finding an instruction sequence that corresponds to the same combination of operations. A survey of code generation by pattern matching may be found in Ganapathi *et al.* (1982).

Fraser and Hansen (1995) describe in detail a C compiler with three alternative target machines. This gives a clear insight into the problems of code generation for dissimilar register machines.

Exercises

Section 7.1

7.1 The Triangle compiler uses code template (7.8e) for while-commands, but many compilers use the following alternative code template:

```
execute [[while E do C]] =
    g:  evaluate E
        JUMPIF(0) h
        execute C
        JUMP g
    h:
```

Convince yourself that the alternative code template is semantically equivalent to (7.8e).

Apply the alternative code template to determine the object code of:

execute [[`while n > 0 do n := n - 2`]]

Compare with Example 7.3, and show that the object code is less efficient.

Why, do you think, is the alternative code template commonly used?

7.2* Suppose that Mini-Triangle is to be extended with the following commands:

(a) V_1 , V_2 := E_1 , E_2

This is a simultaneous assignment: both E_1 and E_2 are to be evaluated, and then their values assigned to the variables V_1 and V_2, respectively.

(b) C_1 , C_2

This is a collateral command: the subcommands C_1 and C_2 are to be executed in any order chosen by the implementor.

(c) `if` E `then` C

This is a conditional command: if E evaluates to *true*, C is executed, otherwise nothing.

(d) `repeat` C `until` E

This is a loop command: E is evaluated at the end of each iteration (after executing C), and the loop terminates if its value is *true*.

(e) `repeat` C_1 `while` E `do` C_2

This is a loop command: E is evaluated in the middle of each iteration (after executing C_1 but before executing C_2), and the loop terminates if its value is *false*.

Write code templates for all these commands.

7.3* Suppose that Mini-Triangle is to be extended with the following expressions:

(a) `if` E_1 `then` E_2 `else` E_3

This is a conditional expression: if E evaluates to *true*, E_2 is evaluated, otherwise E_3 is evaluated. (E_2 and E_3 must be of the same type.)

(b) `let` *D* `in` *E*

This is a block expression: the declaration *D* is elaborated, and the resultant bindings are used in the evaluation of *E*.

(c) `begin` *C* `;` `yield` *E* `end`

Here the command *C* is executed (making side effects), and then *E* is evaluated.

Write code templates for all these expressions.

Section 7.2

7.4* Implement the visitor/encoding methods `visit...Expression` (along the lines of Example 7.8) for the expressions of Exercise 7.3.

7.5* Implement the visitor/encoding methods `visit...Command` (along the lines of Example 7.8) for the commands of Exercise 7.2. Use the technique illustrated in Example 7.9 for generating jump instructions.

Section 7.3

7.6 Classify the following declarations according to whether they bind identifiers to known or unknown values, variables, or routines.

(a) Pascal constant, variable, and procedure declarations, and Pascal value, variable, and procedural parameters.

(b) ML value and function declarations, and ML parameters.

(c) Java local variable declarations, and Java parameters.

7.7* Suppose that Mini-Triangle is to be extended with a for-command of the form '`for` *I* `from` E_1 `to` E_2 `do` *C*', with the following semantics. First, the expressions E_1 and E_2 are evaluated, yielding the integers *m* and *n*, respectively. Then the subcommand *C* is executed repeatedly, with *I* bound to the integers *m*, *m*+1, ..., *n* in successive iterations. If $m > n$, *C* is not executed at all. The scope of *I* is *C*, which may fetch *I* but may not assign to it.

(a) Write a code template for the for-command.

(b) Use it to implement a visitor/encoding method `visitForCommand` (along the lines of Example 7.13).

7.8* Suppose that Mini-Triangle is to be extended with array types, as found in Triangle itself. The relevant extensions to the Mini-Triangle grammar are:

```
V-name      ::=  ...
               |  V-name [ Expression ]
```

```
Type-denoter   ::=   ...
                   |   array Integer-Literal of Type-denoter
```

(a) Modify the Mini-Triangle code specification accordingly.

(b) Modify the Mini-Triangle code generator accordingly.

Section 7.4

7.9* Modify the Mini-Triangle code generator to deal with parameterized procedures, using the code templates of Example 7.24.

7.10* A hypothetical programming language's function declaration has the form '`func` *I* (*FP*) : *T* ~ *C*', i.e., its body is a command. A function body may contain one or more commands of the form '`result` *E*'. This command evaluates expression *E*, and stores its value in an anonymous variable associated with the function. On return from the function, the *latest* value stored in this way is returned as the function's result.

(a) Modify the Mini-Triangle code specification as if Mini-Triangle were extended with functions of this form.

(b) Modify the Mini-Triangle code generator accordingly.