15 A SIMPLE COMPILER - THE BACK END

After the front end has analysed the source code, the back end of a compiler is responsible for synthesizing object code. The critical reader will have realized that code generation, of any form, implies that we consider the semantics of our language and of our target machine, and the interaction between them, in far more detail than we have done until now. Indeed, we have made no real attempt to define what programs written in Clang or Topsy "mean", although we have tacitly assumed that the reader has quite an extensive knowledge of imperative languages, and that we could safely draw on this.

15.1 The code generation interface

In considering the interface between analysis and code generation it will again pay to aim for some degree of machine independence. Generation of code should take place without too much, if any, knowledge of how the analyser works. A common technique for achieving this seemingly impossible task is to define a hypothetical machine, with instruction set and architecture convenient for the execution of programs of the source language, but without being too far removed from the actual system for which the compiler is required. The action of the interface routines will be to translate the source program into an equivalent sequence of operations for the hypothetical machine. Calls to these routines can be embedded in the parser without overmuch concern for how the final generator will turn the operations into object code for the target machine. Indeed, as we have already mentioned, some interpretive systems hand such operations over directly to an interpreter without ever producing real machine code.

The concepts of the meaning of an expression, and of assignment of the "values" of expressions to locations in memory labelled with the "addresses" of variables are probably well understood by the reader. As it happens, such operations are very easily handled by assuming that the hypothetical machine is stack-based, and translating the normal infix notation used in describing expressions into a postfix or Polish equivalent. This is then easily handled with the aid of an evaluation stack, the elements of which are either addresses of storage locations, or the values found at such addresses. These ideas will probably be familiar to readers already acquainted with stack- based machines like the Hewlett-Packard calculator. Furthermore, we have already examined a model of such a machine in section 2.4, and discussed how expressions might be converted into postfix notation in section 11.1.

A little reflection on this theme will suggest that the public interface of such a code generation class might take the form below.

```
enum CGEN_operators {
  CGEN_opadd, CGEN_opsub, CGEN_opmul, CGEN_opdvd, CGEN_opeql,
  CGEN_opneq, CGEN_oplss, CGEN_opgeq, CGEN_opgtr, CGEN_opleq
};

typedef short CGEN_labels;

class CGEN {
  public:
    CGEN_labels undefined;    // for forward references

    CGEN(REPORT *R);
    // Initializes code generator
```

```
    void negateinteger(void);
    // Generates code to negate integer value on top of evaluation stack

    void binaryintegerop(CGEN_operators op);
    // Generates code to pop two values A,B from stack and push value A op B

    void comparison(CGEN_operators op);
    // Generates code to pop two values A,B from stack; push Boolean value A op B

    void readvalue(void);
    // Generates code to read an integer; store on address found on top-of-stack

    void writevalue(void);
    // Generates code to pop and then output the value at top-of-stack

    void newline(void);
    // Generates code to output line mark

    void writestring(CGEN_labels location);
    // Generates code to output string stored from known location

    void stackstring(char *str, CGEN_labels &location);
    // Stores str in literal pool in memory and returns its location

    void stackconstant(int number);
    // Generates code to push number onto evaluation stack

    void stackaddress(int offset);
    // Generates code to push address for known offset onto evaluation stack

    void subscript(void);
    // Generates code to index an array and check that bounds are not exceeded

    void dereference(void);
    // Generates code to replace top-of-stack by the value stored at the
    // address currently stored at top-of-stack

    void assign(void);
    // Generates code to store value currently on top-of-stack on the
    // address stored at next-to-top, popping these two elements

    void openstackframe(int size);
    // Generates code to reserve space for size variables

    void leaveprogram(void);
    // Generates code needed as a program terminates (halt)

    void storelabel(CGEN_labels &location);
    // Stores address of next instruction in location for use in backpatching

    void jump(CGEN_labels &here, CGEN_labels destination);
    // Generates unconditional branch from here to destination

    void jumponfalse(CGEN_labels &here, CGEN_labels destination);
    // Generates branch from here to destination, conditional on the Boolean
    // value currently on top of the evaluation stack, popping this value

    void backpatch(CGEN_labels location);
    // Stores the current location counter as the address field of the branch
    // instruction assumed to be held in an incomplete form at location

    void dump(void);
    // Generates code to dump the current state of the evaluation stack

    void getsize(int &codelength, int &initsp);
    // Returns length of generated code and initial stack pointer

    int gettop(void);
    // Returns the current location counter
};
```

As usual, there are several points that need further comment and explanation:

- The code generation routines have been given names that might suggest that they actually *perform* operations like `jump`. They only *generate* code for such operations, of course.

- There is an unavoidable interaction between this class and the machine for which code is to be generated - the implementation will need to import the machine address type, and we have

seen fit to export a routine (`getsize`) that will allow the compiler to determine the amount of code generated.

- Code for data manipulation on such a machine can be generated by making calls on routines like `stackconstant, stackaddress, stackstring, subscript` and `dereference` for storage access; by calls to routines like `negateinteger` and `binaryintegerop` to generate code to perform simple arithmetic; and finally by calls to `assign` to handle the familiar assignment process.

  For example, compilation of the Clang assignment statement

```
A := 4 + List[5]
```

  (where `List` has 14 elements) should result in the following sequence of code generator routine calls

```
stackaddress(offset of A)
  stackconstant(4)
  stackaddress(offset of List[0])
  stackconstant(5)
  stackconstant(14)
  subscript
  dereference
  binaryintegerop(CGEN_opadd)
assign
```

- The address associated with an array in the symbol table will denote the offset of the first element of the array (the zero-subscript one) from some known "base" at run-time. Our arrays are very simple indeed. They have only one dimension, a size `N` fixed at compile-time, a fixed lower subscript bound of zero, and can easily be handled after allocating them `N` consecutive elements in memory. Addressing an individual element at run time is achieved by computing the value of the subscripting expression, and adding this to (or, on a stack implementation, subtracting it from) the address of the first element in the array. In the interests of safety we shall insist that all subscripting operations incorporate range checks (this is, of course, not done in C++).

- To generate code to handle simple I/O operations we can call on the routines `readvalue, writevalue, writestring` and `newline`.

- To generate code to allow comparisons to be effected we call on `comparison`, suitable parameterized according to the test to be performed.

- Control statements are a little more interesting. In the type of machine being considered it is assumed that machine code will be executed in the order in which it was generated, except where explicit "branch" operations occur. Although our simple language does not incorporate the somewhat despised `GOTO` statement, this maps very closely onto real machine code, and must form the basis of code generated by higher level control statements. The transformation is, of course, easily automated, save for the familiar problem of forward references. In our case there are two source statements that give rise to these. Source code like

```
IF Condition THEN Statement
```

  should lead to object code of the more fundamental form

```
        code for Condition
        IF NOT Condition THEN GOTO LAB END
        code for Statement
LAB     continue
```

and the problem is that when we get to the stage of generating `GOTO LAB` we do not know the address that will apply to `LAB`. Similarly, the source code

```
WHILE Condition DO Statement
```

should lead to object code of the form

```
LAB    code for Condition
       IF NOT Condition THEN GOTO EXIT END
       code for Statement
       GOTO LAB
EXIT   continue
```

Here we should know the address of `LAB` as we start to generate the code for *Condition*, but we shall not know the address of `EXIT` when we get to the stage of generating `GOTO EXIT`.

In general the solution to this problem might require the use of a two-pass system. However, we shall assume that we are developing a one-pass load-and-go compiler, and that the generated code is all in memory, or at worst on a random access file, so that modification of addresses in branch instructions can easily be effected. We generate branch instructions with the aid of `jump(here, label)` and `jumponfalse(here, label)`, and we introduce two auxiliary routines `storelabel(location)` and `backpatch(location)` to remember the location of an instruction, and to be able to repair the address fields of incompletely generated branch instructions at a later stage. The code generator exports a special value of the `CGEN_labels` type that can be used to generate a temporary target destination for such incomplete instructions.

- We have so far made no mention of the forward reference tables which the reader may be dreading. In fact we can leave the system to sort these out implicitly, pointing to yet another advantage of the recursive descent method. A little thought should show that side-effects of allowing only the structured *WhileStatement* and *IfStatement* are that we never need explicit labels, and that we need the same number of implicit labels for each instance of any construct. These labels may be handled by declaring appropriate variables local to parsing routines like *IfStatement*; each time a recursive call is made to *IfStatement* new variables will come into existence, and remain there for as long as it takes to complete parsing of the construction, after which they will be discarded. When compiling an *IfStatement* we simply use a technique like the following (shown devoid of error handling for simplicity):

```
void IfStatement(void)
// IfStatement = "IF" Condition "THEN" Statement .
{ CGEN_labels testlabel;   // must be declared locally
  getsym();                // scan past IF
  Condition();             // generates code to evaluate Condition
  jumponfalse(testlabel,
              undefined);  // remember address of incomplete instruction
  accept(thensym);         // scan past THEN
  Statement();             // generates code for intervening Statement(s)
  backpatch(testlabel);    // use local test value stored by jumponfalse
}
```

If the interior call to `Statement` needs to parse a further *IfStatement*, another instance of `testlabel` will be created for the purpose. Clearly, all variables associated with handling implicit forward references must be declared "locally", or chaos will ensue.

- We may need to generate special housekeeping code as we enter or leave a *Block*. This may not be apparent in the case of a single block program - which is all our language allows at present - but will certainly be the case when we extend the language to support procedures. This code can be generated by the routines `openstackframe` (for code needed as we enter the program) and `leaveprogram` (for code needed as we leave it to return, perhaps, to the charge

of some underlying operating system).

- `gettop` is provided so that a source listing may give details of the object code addresses corresponding to statements in the source.

We are now in a position to show a fully attributed phrase structure grammar for a complete Clang compiler. This could be submitted to Coco/R to generate such a compiler, or could be used to assist in the completion of a hand-crafted compiler such as the one to be found on the source diskette. The power and usefulness of this notation should now be very apparent.

```
PRODUCTIONS
  Clang
  =                           (. TABLE_entries entry; .)
     "PROGRAM"
     Ident<entry.name>        (. debug = (strcmp(entry.name, "DEBUG") == 0);
                                 entry.idclass = TABLE_progs;
                                 Table->enter(entry); .)
     WEAK ";" Block "." .

  Block
  =                           (. int framesize = 0; .)
     SYNC { (  ConstDeclarations | VarDeclarations<framesize> )
     SYNC }                   (. /* reserve space for variables */
                                 CGen->openstackframe(framesize); .)
     CompoundStatement        (. CGen->leaveprogram();
                                 if (debug) /* demonstration purposes */
                                   Table->printtable(stdout); .) .

  ConstDeclarations
  =  "CONST" OneConst { OneConst } .

  OneConst
  =                           (. TABLE_entries entry; .)
     Ident<entry.name>        (. entry.idclass = TABLE_consts; .)
     WEAK "="
     Number<entry.c.value>    (. Table->enter(entry); .)
     ";" .

  VarDeclarations<int &framesize>
  =  "VAR" OneVar<framesize> { WEAK "," OneVar<framesize> } ";" .

  OneVar<int &framesize>
  =                           (. TABLE_entries entry; .)
                              (. entry.idclass = TABLE_vars;
                                 entry.v.size = 1; entry.v.scalar = true;
                                 entry.v.offset = framesize + 1; .)
     Ident<entry.name>
     [ UpperBound<entry.v.size> (. entry.v.scalar = false; .)
     ]                        (. Table->enter(entry);
                                 framesize += entry.v.size; .) .

  UpperBound<int &size>
  =  "[" Number<size> "]"     (. size++; .) .

  CompoundStatement
  =  "BEGIN" Statement { WEAK ";" Statement } "END" .

  Statement
  =  SYNC [  CompoundStatement | Assignment
           | IfStatement       | WhileStatement
           | ReadStatement     | WriteStatement
           | "STACKDUMP"      (. CGen->dump(); .)
         ] .

  Assignment
  =  Variable ":="
     Expression SYNC          (. CGen->assign(); .) .

  Variable
  =                           (. TABLE_entries entry; .)
     Designator<classset(TABLE_vars), entry> .

  Designator<classset allowed, TABLE_entries &entry>
  =                           (. TABLE_alfa name;
                                 bool found; .)
     Ident<name>              (. Table->search(name, entry, found);
```

```
                              if (!found) SemError(202);
                              if (!allowed.memb(entry.idclass)) SemError(206);
                              if (entry.idclass != TABLE_vars) return;
                              CGen->stackaddress(entry.v.offset); .)
   ( "["                   (. if (entry.v.scalar) SemError(204); .)
     Expression            (. /* determine size for bounds check */
                              CGen->stackconstant(entry.v.size);
                              CGen->subscript(); .)
     "]"
     |                     (. if (!entry.v.scalar) SemError(205); .)
   ) .

IfStatement
=                          (. CGEN_labels testlabel; .)
   "IF" Condition "THEN"   (. CGen->jumponfalse(testlabel, CGen->undefined); .)
   Statement               (. CGen->backpatch(testlabel); .) .

WhileStatement
=                          (. CGEN_labels startloop, testlabel, dummylabel; .)
   "WHILE"                 (. CGen->storelabel(startloop); .)
   Condition "DO"          (. CGen->jumponfalse(testlabel, CGen->undefined); .)
   Statement               (. CGen->jump(dummylabel, startloop);
                              CGen->backpatch(testlabel); .) .

Condition
=                          (. CGEN_operators op; .)
   Expression
   (  RelOp<op> Expression (. CGen->comparison(op); .)
    | /* Missing op */     (. SynError(91); .)
   ) .

ReadStatement
=  "READ" "(" Variable     (. CGen->readvalue(); .)
   { WEAK "," Variable     (. CGen->readvalue(); .)
   } ")" .

WriteStatement
=  "WRITE" [ "(" WriteElement { WEAK "," WriteElement } ")" ]
                           (. CGen->newline(); .) .

WriteElement
=                          (. char str[600];
                              CGEN_labels startstring; .)
    String<str>            (. CGen->stackstring(str, startstring);
                              CGen->writestring(startstring); .)
  | Expression             (. CGen->writevalue(); .) .

Expression
=                          (. CGEN_operators op; .)
   (   "+" Term
     | "-" Term            (. CGen->negateinteger(); .)
     | Term
   )
   { AddOp<op> Term        (. CGen->binaryintegerop(op); .)
   } .

Term
=                          (. CGEN_operators op; .)
   Factor
   { (  MulOp<op>
      | /* missing op */   (. SynError(92); op = CGEN_opmul; .)
     ) Factor              (. CGen->binaryintegerop(op); .)
   } .

Factor
=                          (. TABLE_entries entry;
                              int value; .)
     Designator<classset(TABLE_consts, TABLE_vars), entry>
                           (. switch (entry.idclass)
                              { case TABLE_vars :
                                  CGen->dereference(); break;
                                case TABLE_consts :
                                  CGen->stackconstant(entry.c.value); break;
                              } .)
   | Number<value>         (. CGen->stackconstant(value); .)
   | "(" Expression ")" .

AddOp<CGEN_operators &op>
=    "+"                   (. op = CGEN_opadd; .)
   | "-"                   (. op = CGEN_opsub; .) .

MulOp<CGEN_operators &op>
=    "*"                   (. op = CGEN_opmul; .)
```

```
      | "/"                    (. op = CGEN_opdvd; .) .

    RelOp<CGEN_operators &op>
    =    "="                   (. op = CGEN_opeql; .)
       | "<>"                  (. op = CGEN_opneq; .)
       | "<"                   (. op = CGEN_oplss; .)
       | "<="                  (. op = CGEN_opleq; .)
       | ">"                   (. op = CGEN_opgtr; .)
       | ">="                  (. op = CGEN_opgeq; .) .

    Ident<char *name>
    =  identifier              (. LexName(name, TABLE_alfalength); .) .

    String<char *str>
    =  string                  (. char local[100];
                                  LexString(local, sizeof(local) - 1);
                                  int i = 0;
                                  while (local[i]) /* strip quotes */
                                  { local[i] = local[i+1]; i++; }
                                  local[i-2] = '\0';
                                  i = 0;
                                  while (local[i]) /* find internal quotes */
                                  { if (local[i] == '\'')
                                    { int j = i;
                                      while (local[j])
                                      { local[j] = local[j+1]; j++; }
                                    }
                                    i++;
                                  }
                                  strcpy(str, local); .) .

    Number <int &num>
    =  number                  (. char str[100];
                                  int i = 0, l, digit, overflow = 0;
                                  num = 0;
                                  LexString(str, sizeof(str) - 1);
                                  l = strlen(str);
                                  while (i <= l && isdigit(str[i]))
                                  { digit = str[i] - '0'; i++;
                                    if (num <= (maxint - digit) / 10)
                                      num = 10 * num + digit;
                                    else overflow = 1;
                                  }
                                  if (overflow) SemError(200); .) .

END Clang.
```

A few points call for additional comment:

- The reverse Polish (postfix) form of the expression manipulation is accomplished simply by delaying the calls for "operation" code generation until after the second "operand" code generation has taken place - this is, of course, completely analogous to the system developed in section 11.1 for converting infix expression strings to their reverse Polish equivalents.

- It turns out to be useful for debugging purposes, and for a full understanding of the way in which our machine works, to be able to print out the evaluation stack at any point in the program. This we have done by introducing another keyword into the language, `STACKDUMP`, which can appear as a simple statement, and whose code generation is handled by `dump`.

- The reader will recall that the production for *Factor* would be better expressed in a way that would introduce an LL(1) conflict into the grammar. This conflict is resolved within the production for *Designator* in the above Cocol grammar; it can be (and is) resolved within the production for *Factor* in the hand-crafted compiler on the source diskette. In other respects the semantic actions found in the hand-crafted code will be found to match those in the Cocol grammar very closely indeed.

Exercises

Many of the previous suggestions for extending Clang or Topsy will act as useful sources of inspiration for projects. Some of these may call for extra code generator interface routines, but many will be found to require no more than we have already discussed. Decide which of the following problems can be solved immediately, and for those that cannot, suggest the minimal extensions to the code generator that you can foresee might be necessary.

15.1 How do you generate code for the `REPEAT ... UNTIL` statement in Clang (or the `do` statement in Topsy)?

15.2 How do you generate code for an `IF ... THEN ... ELSE` statement, with the "dangling else" ambiguity resolved as in Pascal or C++? Bear in mind that the `ELSE` part may or may not be present, and ensure that your solution can handle both situations.

15.3 What sort of code generation is needed for the Pascal or Modula-2 style `FOR` loop that we have suggested adding to Clang? Make sure you understand the semantics of the `FOR` loop before you begin - they may be more subtle than you think!

15.4 What sort of code generation is needed for the C++ style `for` loop that we have suggested adding to Topsy?

15.5 Why do you suppose languages allow a `FOR` loop to terminate with its control variable "undefined"?

15.6 At present the `WRITE` statement of Clang is rather like Pascal's `WriteLn`. What changes would be needed to provide an explicit `WRITELN` statement, and, similarly, an explicit `READLN` statement, with semantics as used in Pascal.

15.7 If you add a "character" data type to your language, as suggested in Exercise 14.30, how do you generate code to handle `READ` and `WRITE` operations?

15.8 Code generation for the `LOOP ... EXIT ... END` construction suggested in Exercise 14.28 provides quite an interesting exercise. Since we may have several `EXIT` statements in a loop, we seem to have a severe forward reference problem. This may be avoided in several ways. For example, we could generate code of the form

```
           GOTO STARTLOOP
EXITPOINT  GOTO LOOPEXIT
STARTLOOP  code for loop body
           . . .
           GOTO EXITPOINT     (from an EXIT statement)
           . . .
           GOTO STARTLOOP
LOOPEXIT   code which follows loop
```

With this idea, all `EXIT` statements can branch back to `EXITPOINT`, and we have only to backpatch the one instruction at `EXITPOINT` when we reach the `END` of the `LOOP`. This is marginally inefficient, but the execution of one extra `GOTO` statement adds very little to the overall execution time.

Another idea is to generate code like

```
STARTLOOP  code for loop body
           . . .
           GOTO EXIT1    (from an EXIT statement)
           . . .
EXIT1      GOTO EXIT2    (from an EXIT statement)
```

```
            . . .
EXIT2       GOTO LOOPEXIT (from an EXIT statement)
            . . .
            GOTO STARTLOOP
LOOPEXIT    code which follows END
```

In this case, each time another `EXIT` is encountered the previously incomplete one is backpatched to branch to the incomplete instruction which is just about to be generated. When the `END` is encountered, the last one is backpatched to leave the loop. (A `LOOP ... END` structure may, unusually, have no `EXIT` statements, but this is easily handled.) This solution is even less efficient than the last. An ingenious modification can lead to much better code. Suppose we generate code which at first appears quite incorrect, on the lines of

```
STARTLOOP  code for loop body
           . . .
EXIT0      GOTO 0        (incomplete - from an EXIT statement)
           . . .
EXIT1      GOTO EXIT0    (from an EXIT statement)
           . . .
EXIT2      GOTO EXIT1    (from an EXIT statement)
           . . .
```

with an auxiliary variable `Exit` which contains the address of the most recent of the `GOTO` instructions so generated. (In the above example this would contain the address of the instruction labelled `EXIT2`). We have used only backward references so far, so no real problems arise. When we encounter the `END`, we refer to the instruction at `Exit`, alter its address field to the now known forward address, and use the old backward address to find the address of the next instruction to modify, repeating this process until the "`GOTO 0`" is encountered, which stops the chaining process - we are, of course, doing nothing other than constructing a linked list temporarily within the generated code.

Try out one or other approach, or come up with your own ideas. All of these schemes need careful thought when the possibility exists for having nested `LOOP ... END` structures, which you should allow.

15.9 What sort of code generation is needed for the translation of structured statements like the following?

```
IfStatement       =  "IF" Condition "THEN" StatementSequence
                       { "ELSIF" Condition "THEN" StatementSequence }
                       [ "ELSE" StatementSequence  ]
                     "END" .
WhileStatement    =  "WHILE" Condition "DO" StatementSequence  "END" .
StatementSequence =  Statement { ";" Statement } .
```

15.10 Brinch Hansen (1983) introduced an extended form of the `WHILE` loop into the language Edison:

```
WhileStatement    =  "WHILE" Condition "DO" StatementSequence
                       { "ELSE" Condition "DO" StatementSequence }
                     "END" .
```

The *Conditions* are evaluated one at a time in the order written until one is found to be true, when the corresponding *StatementSequence* is executed, after which the process is repeated. If no *Condition* is true, the loop terminates. How could this be implemented? Can you think of any algorithms where this statement would be useful?

15.11 Add a `HALT` statement, as a variation on the `WRITE` statement, which first prints the values of its parameters and then aborts execution.

15.12 How would you handle the `GOTO` statement, assuming you were to add it to the language?

What restrictions or precautions should you take when combining it with structured loops (and, in particular, `FOR` loops)?

15.13 How would you implement a `CASE` statement in Clang, or a `switch` statement in Topsy? What should be done to handle an `OTHERWISE` or `default`, and what action should be taken to be taken when the selector does not match any of the labelled "arms"? Is it preferable to regard this as an error, or as an implicit "do nothing"?

15.14 Add the `MOD` or `%` operator for use in finding remainders in expressions, and the `AND, OR` and `NOT` operations for use in forming more complex *Conditions*.

15.15 Add `INC(x)` and `DEC(x)` statements to Clang, or equivalently add `x++` and `x--` statements to Topsy - thereby turning it, at last, into Topsy++! The Topsy version will introduce an LL(1) conflict into the grammar, for now there will be three distinct alternatives for *Statement* that commence with an identifier. However, this conflict is not hard to resolve.

Further reading

The hypothetical stack machine has been widely used in the development of Pascal compilers. In the book by Welsh and McKeag (1980) can be found a treatment on which our own is partly based, as is the excellent treatment by Elder (1994). The discussion in the book by Wirth (1976b) is also relevant, although, as is typical in several systems like this, no real attempt is made to specify an interface to the code generation, which is simply overlaid directly onto the analyser in a machine dependent way. The discussion of the Pascal-P compiler in the book by Pemberton and Daniels (1982) is, as usual, extensive. However, code generation for a language supporting a variety of data types (something we have so far assiduously avoided introducing except in the exercises) tends to obscure many principles when it is simply layered onto an already large system.

Various approaches can be taken to compiling the `CASE` statement. The reader might like to consult the early papers by Sale (1981) and Hennessy and Mendelsohn (1982), as well as the descriptions in the book by Pemberton and Daniels (1982).

15.2 Code generation for a simple stack machine

The problem of code generation for a real machine is, in general, complex, and very specialized. In this section we shall content ourselves with completing our first level Clang compiler on the assumption that we wish to generate code for the stack machine described in section 4.4. Such a machine does not exist, but, as we saw, it may readily be emulated by a simple interpreter. Indeed, if the `interpret` routine from that section is invoked from the driver program after completing a successful parse, an implementation of Clang quite suitable for experimental work is readily produced.

An implementation of an "on-the-fly" code generator for this machine is almost trivially easy, and can be found on the source diskette. In studying this code the reader should note that:

- An external instance of the `STKMC` class is made directly visible to the code generator; as the code is generated it is stored directly in the code array `Machine->mem`.

- The constructor of the code generator class initializes two private members - a location counter (`codetop`) needed for storing instructions, and a top of memory pointer (`stktop`) needed for storing string literals.

- The `stackaddress` routine is passed a simple symbol table address value, and converts this into an offset that will later be computed relative to the `cpu.bp` register when the program is executed.

- The main part of the code generation is done in terms of calls to a routine `emit`, which does some error checking that the "memory" has not overflowed. Storing a string in the literal pool in high memory is done by routine `stackstring`, which is also responsible for overflow checking. As usual, errors are reported through the error reporting class discussed in section 14.3; the code generator suppresses further attempts to generate code if memory overflow occurs, while still allowing syntactic parsing to proceed.

- Since many of the routines in the code generator class are very elementary interfaces to `emit`, the reader might feel that we have taken modular decomposition too far - code generation as simple as this could surely be made more efficient if the parser simply evoked `emit` directly. This is certainly true, and many recursive descent compilers do this.

- Code generation is very easy for a stack-oriented language. It is much more difficult for a machine with no stack, and only a few registers and addressing modes. However, as the discussion in later sections will reveal, the interface we have developed is, in fact, capable of being used with only minor modification for code generation for more conventional machines. Developing the system in a highly modular way has many advantages if one is striving for portability, and for the ability to construct improved back ends easily.

It may be of interest to show the code generated for a simple program that incorporates several features of the language.

```
Clang 1.0 on 19/05/96 at 22:17:12

  0 : PROGRAM Debug;
  0 :   CONST
  0 :     VotingAge = 18;
  0 :   VAR
  0 :     Eligible, Voters[100], Age, Total;
  0 :   BEGIN
  2 :     Total := 0;
  7 :     Eligible := 0;
 12 :     READ(Age);
 15 :     WHILE Age > 0 DO
 23 :       BEGIN
 23 :         IF Age > VotingAge THEN
 29 :           BEGIN
 31 :             Voters[Eligible] := Age;
 43 :             Eligible := Eligible + 1;
 52 :             Total := Total + Voters[Eligible -1];
 67 :           END;
 71 :         READ(Age);
 74 :       END;
 76 :     WRITE(Eligible, ' voters. Average age = ', Total / Eligible);
 91 :   END.
```

The symbol table has entries

```
1 DEBUG          Program
2 VOTINGAGE      Constant      18
3 ELIGIBLE       Variable       1
4 VOTERS         Variable       2
5 AGE            Variable     103
6 TOTAL          Variable     104
```

and the generated code is as follows:

```
 0 DSP   104    Reserve variable space
 2 ADR  -104      address of Total
 4 LIT     0      push 0
 6 STO          Total := 0
 7 ADR    -1      address of Eligible
 9 LIT     0      push 0
11 STO          Eligible := 0
12 ADR  -103      address of Age
14 INN          READ(Age)
15 ADR  -103      address of Age
17 VAL            value of Age
18 LIT     0      push 0
20 GTR            compare
21 BZE    74    WHILE Age > 0 DO
23 ADR  -103      address of Age
25 VAL            value of Age
26 LIT    18      push VotingAge
28 GTR            compare
29 BZE    69      IF Age > VotingAge THEN
31 ADR    -2        address of Voters[0]
33 ADR    -1        address of Eligible
35 VAL              value of Eligible
36 LIT   101        array size 101
38 IND              address of Voters[Eligible]
39 ADR  -103        address of Age
41 VAL              value of Age
42 STO              Voters[Eligible] := Age
43 ADR    -1          address of Eligible
45 ADR    -1          address of Eligible
47 VAL                value of Eligible
48 LIT     1          push 1
50 ADD                value of Eligible + 1
51 STO              Eligible := Eligible + 1
52 ADR  -104          address of Total
54 ADR  -104          address of Total
56 VAL                value of Total
57 ADR    -2          address of Voters[0]
59 ADR    -1          address of Eligible
61 VAL                value of Eligible
62 LIT     1          push 1
64 SUB                value of Eligible - 1
65 LIT   101          array size 101
67 IND                address of Voters[Eligible-1]
68 VAL                value of Voters[Eligible - 1]
69 ADD                value of Total + Voters[Eligible - 1]
70 STO              Total := Total + Voters[Eligible - 1]
71 ADR  -103           address of Age
73 INN              READ(Age)
74 BRN    15        to start of WHILE loop
76 ADR    -1      address of Eligible
78 VAL            value of Eligible
79 PRN          WRITE(Eligible,
80 PRS  ' voters. Average age = '
82 ADR  -104      address of Total
84 VAL            value of Total
85 ADR    -1      address of Eligible
87 VAL            value of Eligible
88 DVD            value of Total / Eligible
89 PRN          WRITE(Total / Eligible)
90 NLN          output new line
91 HLT          END.
```

Exercises

15.16 If you study the code generated by the compiler you should be struck by the fact that the sequence `ADR x; VAL` occurs frequently. Investigate the possibilities for peephole optimization. Assume that the stack machine is extended to provide a new operation `PSH x` that will perform this sequence in one operation as follows:

```
case STKMC_psh:
  cpu.sp--;
  int ear = cpu.bp + mem[cpu.pc];
```

```
if (inbounds(cpu.sp) && inbounds(ear))
  { mem[cpu.sp] = mem[ear]; cpu.pc++; }
break;
```

Go on to modify the code generator so that it will replace any sequence `ADR x; VAL` with `PSH x` (be careful: not all `VAL` operations follow immediately on an `ADR` operation).

15.17 Augment the system so that you can declare constants to be literal strings and print these, for example

```
PROGRAM Debug;
  CONST
    Planet = 'World';
  BEGIN
    WRITE('Hello ', Planet)
  END.
```

How would you need to modify the parser, code generator, and run-time system?

15.18 Suppose that we wished to use relative branch instructions, rather than absolute branch instructions. How would code generation be affected?

15.19 (Harder) Once you have introduced a Boolean type into Clang or Topsy, along with `AND` and `OR` operations, try to generate code based on short-circuit semantics, rather than the easier Boolean operator approach. In the short-circuit approach the operators `AND` and `OR` are defined to have semantic meanings such that

```
A AND B       means      IF A THEN B ELSE FALSE END
A OR  B       means      IF A THEN TRUE ELSE B END
```

In the language Ada this has been made explicit: `AND` and `OR` alone have Boolean operator semantics, but `AND THEN` and `OR ELSE` have short-circuit semantics. Thus, in Ada

```
A AND THEN B       means      IF A THEN B ELSE FALSE END
A OR ELSE  B       means      IF A THEN TRUE ELSE B END
```

Can you make your system accept both forms?

15.20 Consider an extension where we allow a one-dimensional array with fixed bounds, but with the lower bound set by the user. For example, a way to declare such arrays might be along the lines of

```
CONST
  BC = -44;
  AD = 300;
VAR
  WWII[1939 : 1945], RomanBritain[BC : AD];
```

Modify the language, compiler, and interpreter to handle this idea, performing bounds checks on the subscript. Addressing an element is quite easy. If we declare an array

```
VAR Array[Min : Max];
```

then the offset of the *I* th element in the array is computed as

```
I - Min + offset of first element of array
```

which may give some hints about the checking problem too, if you think of it as

```
(offset of first element of array - Min)  +  I
```

15.21 A little more ingenuity is called for if one is to allow two-dimensional arrays. Again, if these

are of fixed size, addressing is quite easy. Suppose we declare a matrix

```
VAR Matrix[MinX : MaxX , MinY : MaxY];
```

Then we shall have to reserve `(MaxX-MinX+1) * (MaxY-MinY+1)` consecutive locations for the whole array. If we store the elements by rows (as in most languages, other than Fortran), then the offset of the *I,J* th element in the matrix will be found as

```
(I - MinX) * (MaxY - MinY + 1) + (J - MinY) + offset of first element
```

You will need a new version of the `STK_ind` opcode (incorporating bounds checking).

15.22 Extend your system to allow whole arrays (of the same length) to be assigned one to another.

15.23 Some users like to live dangerously. How could you arrange for the compiler to have an option whereby generation of subscript range checks could be suppressed?

15.24 Complete level 1 of your extended Clang or Topsy compiler by generating code for all the extra statement forms that you have introduced while assiduously exploring the exercises suggested earlier in this chapter. How many of these can only be completed if the instruction set of the machine is extended?

15.3 Other aspects of code generation

As the reader may have realized, the approach taken to code generation up until now has been rather idealistic. A hypothetical stack machine is, in many ways, ideal for our language - as witness the simplicity of the code generator - but it may differ rather markedly from a real machine. In this section we wish to look at other aspects of this large and intricate subject.

15.3.1 Machine tyranny

It is rather awkward to describe code generation for a real machine in a general text. It inevitably becomes machine specific, and the principles may become obscured behind a lot of detail for an architecture with which the reader may be completely unfamiliar. To illustrate a few of these difficulties, we shall consider some features of code generation for a relatively simple processor.

The Zilog Z80 processor that we shall use as a model is typical of several 8-bit microprocessors that were very popular in the early 1980's, and which helped to spur on the microcomputer revolution. It had a single 8-bit accumulator (denoted by `A`), several internal 8-bit registers (denoted by `B, C, D, E, H` and `L`), a 16-bit program counter (`PC`), two 16-bit index registers (`IX` and `IY`), a 16-bit stack pointer (`SP`), an 8-bit data bus, and a 16-bit address bus to allow access to 64KB of memory. With the exception of the `BRN` opcode (and, perhaps, the `HLT` opcode), our hypothetical machine instructions do not map one-for-one onto Z80 opcodes. Indeed, at first sight the Z80 would appear to be ill suited to supporting a high-level language at all, since operations on a single 8-bit accumulator only provide for handling numbers between -128 and +127, scarcely of much use in arithmetic calculations. For many years, however - even after the introduction of processors like the Intel 80x86 and Motorola 680x0 - 16-bit arithmetic was deemed adequate for quite a number of operations, as it allows for numbers in the range -32768 to +32767. In the Z80 a limited number of operations were allowed on 16-bit register pairs. These were denoted `BC, DE` and `HL`, and were formed by simply concatenating the 8-bit registers mentioned earlier. For example, 16-bit constants could be loaded immediately into a register pair, and such pairs could be pushed and popped from

the stack, and transferred directly to and from memory. In addition, the HL pair could be used as a 16-bit accumulator into which could be added and subtracted the other pairs, and could also be used to perform register-indirect addressing of bytes. On the Z80 the 16-bit operations stopped short of multiplication, division, logical operations and even comparison against zero, all of which are found on more modern 16 and 32-bit processors. We do not propose to describe the instruction set in any detail; hopefully the reader will be able to understand the code fragments below from the commentary given alongside.

As an example, let us consider Z80 code for the simple assignment statement

```
I := 4 + J - K
```

where I, J and K are integers, each stored in two bytes. A fairly optimal translation of this, making use of the HL register pair as a 16 bit accumulator, but not using a stack in any way, might be as follows:

```
LD    HL,4       ; HL := 4
LD    DE,(J)     ; DE := Mem[J]
ADD   HL,DE      ; HL := HL + DE          (4 + J)
LD    DE,(K)     ; DE := Mem[K]
OR    A          ; just to clear Carry
SBC   HL,DE      ; HL := HL - DE - Carry  (4 + J - K)
LD    (I),HL     ; Mem[I] := HL
```

On the Z80 this amounted to some 18 bytes of code. The only point worth noting is that, unlike addition, there was no simple 16-bit subtraction operation, only one which involved a carry bit, which consequently required unsetting before SBC could be executed. By contrast, the same statement coded for our hypothetical machine would have produced 13 words of code

```
ADR   I          ; push address of I
LIT   4          ; push constant 4
ADR   J          ; push address of J
VAL              ; replace with value of J
ADD              ; 4 + J
ADR   K          ; push address of K
VAL              ; replace with value of K
SUB              ; 4 + J - K
STO              ; store on I
```

and for a simple single-accumulator machine like that discussed in Chapter 4 we should probably think of coding this statement on the lines of

```
LDI   4          ; A := 4
ADD   J          ; A := 4 + J
SUB   K          ; A := 4 + J - K
STA   I          ; I := 4 + J - K
```

How do we begin to map stack machine code to these other forms? One approach might be to consider the effect of the opcodes, as defined in the interpreter in Chapter 4, and to arrange that code generating routines like stackaddress, stackconstant and assign generate code equivalent to that which would be obeyed by the interpreter. For convenience we quote the relevant equivalences again. (We use T to denote the virtual machine top of stack pointer, to avoid confusion with the SP register of the Z80 real machine.)

```
ADR address : T := T - 1; Mem[T] := address          (push an address)
LIT value   : T := T - 1; Mem[T] := value            (push a constant)
VAL         : Mem[T] := Mem[Mem[T]]                  (dereference)
ADD         : T := T + 1; Mem[T] := Mem[T] + Mem[T-1] (addition)
SUB         : T := T + 1; Mem[T] := Mem[T] - Mem[T-1] (subtraction)
STO         : Mem[Mem[T+1]] := Mem[T]; T := T + 2     (store top-of-stack)
```

It does not take much imagination to see that this would produce a great deal more code than we should like. For example, the equivalent Z80 code for an LIT opcode, obtained from a translation of

the sequence above, and generated by `stackconstant(Num)` might be

```
T := T - 1       :  LD    HL,(T)     ; HL := T
                    DEC   HL         ; HL := HL - 1
                    DEC   HL         ; HL := HL - 1
                    LD    (T),HL     ; T := HL
Mem[T] := Num    :  LD    DE,Num     ; DE := Num
                    LD    (HL),E     ; store low order byte
                    INC   HL         ; HL := HL + 1
                    LD    (HL),D     ; store high order byte
```

which amounts to some 14 bytes. We should comment that `HL` must be decremented twice to allow for the fact that memory is addressed in bytes, not words, and that we have to store the two halves of the register pair `DE` in two operations, "bumping" the `HL` pair (used for register indirect addressing) between these.

If the machine for which we are generating code does not have some sort of hardware stack we might be forced or tempted into taking this approach, but fortunately most modern processors do incorporate a stack. Although the Z80 did not support operations like `ADD` and `SUB` on elements of its stack, the pushing which is implicit in `LIT` and `ADR` is easily handled, and the popping and pushing implied by `ADD` and `SUB` are nearly as simple. Consequently, it would be quite simple to write code generating routines which, for the same assignment statement as before, would have the effects shown below.

```
ADR  I   :    LD    HL,I       ;     HL := address of I
              PUSH  HL         ; push address of I
LIT  4   :    LD    DE,4       ;     DE := 4
              PUSH  DE         ; push value of 4
ADR  J   :    LD    HL,J       ;     HL := address of J
              PUSH  HL         ; push address of J                  *
VAL      :    POP   HL         ;     HL := address of variable      *
              LD    E,(HL)     ;     E := Mem[HL]  low order byte
              INC   HL         ;     HL := HL + 1
              LD    D,(HL)     ;     D := Mem[HL]  high order byte
              PUSH  DE         ; replace with value of J            *
ADD      :    POP   DE         ;     DE := second operand           *
              POP   HL         ;     HL := first operand
              ADD   HL,DE      ;     HL := HL + DE
              PUSH  HL         ; 4 + J
ADR  K   :    LD    HL,K       ;     HL := address of K
              PUSH  HL         ; push address of K                  *
VAL      :    POP   HL         ;     HL := address of variable      *
              LD    E,(HL)     ;     E := low order byte
              INC   HL         ;     HL := HL + 1
              LD    D,(HL)     ;     D := high order byte
              PUSH  DE         ; replace with value of K            *
SUB      :    POP   DE         ;     DE := second operand           *
              POP   HL         ;     HL := first operand
              OR    A          ;     unset carry
              SBC   HL,DE      ;     HL := HL - DE - carry
              PUSH  HL         ; 4 + J - K                          **
STO      :    POP   DE         ;     DE := value to be stored       **
              POP   HL         ;     HL := address to be stored at
              LD    (HL),E     ;     Mem[HL] := E  store low order byte
              INC   HL         ;     HL := HL + 1
              LD    (HL),D     ;     Mem[HL] := D  store high order byte
                               ; store on I
```

We need not present code generator routines based on these ideas in any detail. Their intent should be fairly clear - the code generated by each follows distinct patterns, with obvious differences being handled by the parameters which have already been introduced.

For the example under discussion we have generated 41 bytes, which is still quite a long way from the optimal 18 given before. However, little effort would be required to reduce this to 32 bytes. It is easy to see that 8 bytes could simply be removed (the ones marked with a single asterisk), since the operations of pushing a register pair at the end of one code generating sequence and of popping the same pair at the start of the next are clearly redundant. Another byte could be removed by replacing

the two marked with a double asterisk by a one-byte opcode for exchanging the DE and HL pairs (the Z80 code EX DE,HL does this). These are examples of so-called "peephole" optimization, and are quite easily included into the code generating routines we are contemplating. For example, the algorithm for assign could be

```
PROCEDURE Assign;
(* Generate code to store top-of-stack on address stored next-to-top *)
  BEGIN
    IF last code generated was PUSH HL
      THEN replace this PUSH HL with EX DE,HL
      ELSIF last code generated was PUSH DE
        THEN delete PUSH DE
        ELSE generate code for POP DE
    END;
    generate code for POP HL; generate code for LD (HL),E
    generate code for INC HL; generate code for LD (HL),D
  END;
```

(The reader might like to reflect on the kinds of assignment statements which would give rise to the three possible paths through this routine.)

By now, hopefully, it will have dawned on the reader that generation of native code is probably strongly influenced by the desire to make this compact and efficient, and that achieving this objective will require the compiler writer to be highly conversant with details of the target machine, and with the possibilities afforded by its instruction set. We could pursue the ideas we have just introduced, but will refrain from doing so, concentrating instead on how one might come up with a better structure from which to generate code.

15.3.2 Abstract syntax trees as intermediate representations

In section 13.3 the reader was introduced to the idea of deriving an abstract syntax tree from an expression, by attributing the grammar that describes such expressions so as to add semantic actions that construct the nodes in these trees and then link them together as the parsing process is carried out. After such a tree has been completely constructed by the syntactic/semantic analysis phase (or even during its construction) it is often possible to carry out various transformations on it before embarking on the code generation phase that consists of walking it in some convenient way, generating code apposite to each node as it is visited.

In principle we can construct a single tree to represent an entire program. Initially we shall prefer simply to demonstrate the use of trees to represent the expressions that appear on both sides of *Assignments*, as components of *ReadStatements* and *WriteStatements* and as components of the *Condition* in *IfStatements* and *WhileStatements*. Later we shall extend the use of trees to handle the compilation of parameterized subroutine calls as well.

A tree is usually implemented as a structure in which the various nodes are linked by pointers, and so we declare an AST type to be a pointer to a NODE type. The nodes are inhomogeneous. When we declare them in traditional implementations we resort to using variant records or unions, but in a C++ implementation we can take advantage of inheritance to derive various node classes from a base class, declared as

```
struct NODE {
  int value;                    // value to be associated with this node
  bool defined;                 // true if value is predictable at compile time
  NODE()                        { defined = 0; }
  virtual void emit1(void) = 0;
  virtual void emit2(void) = 0;
  // ... further members as appropriate
};
```

where the emit member functions will be responsible for code generation as the nodes are visited

during the code generation phase. It makes sense to think of a value associated with each node - either a value that can be predicted at compile-time, or a value that will require code to be generated to compute it at run-time (where it will then be stored in a register, or on a machine stack, perhaps).

When constants form operands of expressions they give rise to nodes of a simple `CONSTNODE` class:

```
struct CONSTNODE : public NODE {
  CONSTNODE(int V)              { value = V; defined = true; }
  virtual void emit1(void);     // generate code to retrieve value of constant
  virtual void emit2(void)      {;}
};
```

Operands in expressions that are known to be associated with variables are handled by introducing a derived `VARNODE` class. Such nodes need to store the variable's offset address, and to provide at least two code generating member functions. These will handle the generation of code when the variable is associated with a *ValueDesignator* (as it is in a *Factor*) and when it is associated with a *VariableDesignator* (as it is on the left side of an *Assignment*, or in a *ReadStatement*).

```
struct VARNODE : public NODE {
  int offset;                   // offset of variable assigned by compiler
  VARNODE() {;}                 // default constructor
  VARNODE(int O)                { offset = O; }
  virtual void emit1(void);     // generate code to retrieve value of variable
  virtual void emit2(void);     // generate code to retrieve address of variable
};
```

To handle access to the elements of an array we derive an `INDEXNODE` class from the `VARNODE` class. The member function responsible for retrieving the address of an array element has the responsibility of generating code to perform run-time checks that the index remains in bounds, so we need further pointers to the subtrees that represent the index expression and the size of the array.

```
struct INDEXNODE : public VARNODE {
  AST size;                     // for range checking
  AST index;                    // subscripting expression
  INDEXNODE(int O, AST S, AST I) { offset = O; size = S; index = I; }
  //      void emit1(void)         is inherited from VARNODE
  virtual void emit2(void);     // code to retrieve address of array element
};
```

Finally, we derive two node classes associated with unary (prefix) and binary (infix) arithmetic operations

```
struct MONOPNODE : public NODE {
  CGEN_operators op;
  AST operand;
  MONOPNODE(CGEN_operators O, AST E) { op = O; operand = E }
  virtual void emit1(void);     // generate code to evaluate "op operand"
  virtual void emit2(void)      {;}
};

struct BINOPNODE : public NODE {
  CGEN_operators op;
  AST left, right;
  BINOPNODE(CGEN_operators O, AST L, AST R) { op = O; left = L; right = R; }
  virtual void emit1(void);     // generate code to evaluate "left op right"
  virtual void emit2(void)      {;}
};
```

The structures we hope to set up are exemplified by considering an assignment statement

```
  A[X + 4] := (A[3] + Z) * (5 - 4 * 1) - Y
```

We use one tree to represent the address used for the destination (left side), and one for the value of the expression (right side), as shown in Figure 15.1, where for illustration the array `A` is assumed to have been declared with an size of 8.

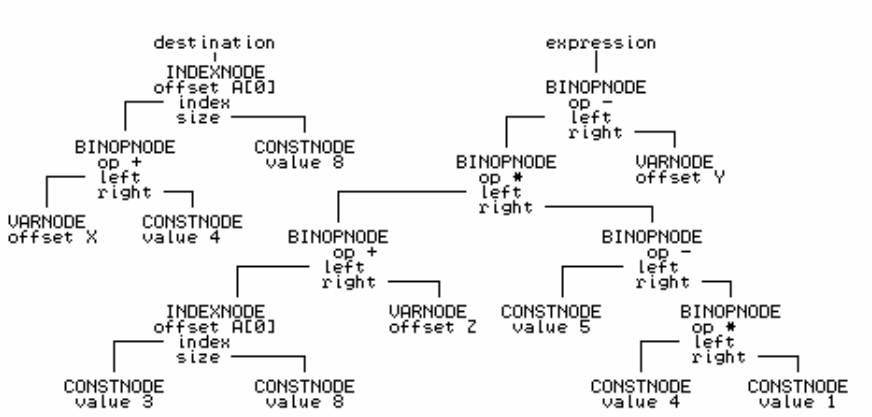

Figure 15.1 AST structures for the statement
A[X + 4] := (A[3] + Z) * (5 - 4 * 1) - Y

Tree-building operations may be understood by referring to the attributes with which a Cocol grammar would be decorated:

```
Assignment
=                           (. AST dest, exp; .)
   Variable<dest> ":="
   Expression<exp> SYNC     (. CGen->assign(dest, exp); .) .

Variable<AST &V>
=                           (. TABLE_entries entry; .)
   Designator<V, classset(TABLE_vars), entry>.

Designator<AST &D, classset allowed, TABLE_entries &entry>
=                           (. TABLE_alfa name;
                               AST index, size;
                               bool found;
                               D = CGen->emptyast(); .)
   Ident<name>              (. Table->search(name, entry, found);
                               if (!found) SemError(202);
                               if (!allowed.memb(entry.idclass)) SemError(206);
                               if (entry.idclass != TABLE_vars) return;
                               CGen->stackaddress(D, entry.v.offset); .)
   ( "["                    (. if (entry.v.scalar) SemError(204); .)
     Expression<index>      (. if (!entry.v.scalar)
                               /* determine size for bounds check */
                               { CGen->stackconstant(size, entry.v.size);
                                 CGen->subscript(D, entry.v.offset,
                                                 size, index); } .)
     "]"
     |                      (. if (!entry.v.scalar) SemError(205); .)
   ) .

Expression<AST &E>
=                           (. AST T;
                               CGEN_operators op;
                               E = CGen->emptyast(); .)
   (   "+" Term<E>
     | "-" Term<E>          (. CGen->negateinteger(E); .)
     | Term<E>
   )
   { AddOp<op> Term<T>      (. CGen->binaryintegerop(op, E, T); .)
   } .

Term<AST &T>
=                           (. AST F;
                               CGEN_operators op; .)
   Factor<T>
   { (  MulOp<op>
      | /* missing op */    (. SynError(92); op = CGEN_opmul; .)
     ) Factor<F>            (. CGen->binaryintegerop(CGEN_op, T, F); .)
   } .

Factor<AST &F>
=                           (. TABLE_entries entry;
                               int value;
                               F = CGen->emptyast(); .)
     Designator<F, classset(TABLE_consts, TABLE_vars), entry>
```

```
                          (. switch (entry.idclass)
                             { case TABLE_consts :
                                 CGen->stackconstant(F, entry.c.value);
                                 break;
                               default : break;
                             } .)
    | Number<value>         (. CGen->stackconstant(F, value); .)
    | "(" Expression<F> ")" .
```

The reader should note that:

- This grammar is very close to that presented earlier. The code generator interface is changed only in that the various routines need extra parameters specifying the subtrees that they manipulate.

- The productions for *Designator*, *Expression* and *Factor* take the precaution of initializing their formal parameter to point to an "empty" node, so that if a syntax error is detected, the nodes of a tree will still be well defined.

In a simple system, the various routines like `stackconstant, stackaddress` and `binaryintegerop` do little more than call upon the appropriate class constructors. As an example, the routine for `binaryintegerop` is merely

```
void binaryintegerop(CGEN_operators op, AST &left, AST &right)
{ left = new BINOPNODE(op, left, right); }
```

where we note that the `left` parameter is used both for input and output (this is done to keep the code generation interface as close as possible to that used in the previous system). These routines simply build the tree, and do not actually generate code.

Code generation is left in the charge of routines like `assign`, `jumponfalse` and `readvalue`, which take new parameters denoting the tree structures that they are required to walk. This may be exemplified by code for the `assign` routine, as it would be developed to generate code for our simple stack machine

```
void CGEN::assign(AST dest, AST expr)
{ if (dest)                // beware of corrupt trees
  { dest->emit2();         // generate code to push address of destination
    delete dest;           // recovery memory used for tree
  }
  if (expr)                // beware of corrupt trees
  { expr->emit1();         // generate code to push value of expression
    delete expr;           // recovery memory used for tree
    emit(int(STKMC_sto));  // generate the store instruction
  }
}
```

In typical OOP fashion, each subtree "knows" how to generate its own code! For a `VARNODE`, for example, and for our stack machine, we would define the `emit` members as follows:

```
void VARNODE::emit1(void)   // load variable value onto stack
{ emit2(); CGen->emit(int(STKMC_val)); }

void VARNODE::emit2(void)   // load variable address onto stack
{ CGen->emit(int(STKMC_adr)); CGen->emit(-offset); }
```

15.3.3 Simple optimizations - constant folding

The reader may need to be convinced that the construction of a tree is of any real value, especially when used to generate code for a simple stack machine. To back up the assertion that transformations on a tree are easily effected and can lead to the generation of better code, let us reconsider the statement

```
A[X + 4] := (A[3] + Z) * (5 - 4 * 1) - Y
```

It is easy to identify opportunities for code improvement:

- `A[3]` represents an array access with a constant index. There is no real need to compute the additional offset for `A[3]` at run-time. It can be done at compile-time, along with a compile-time (rather than run-time) check that the subscript expression is "in bounds".

- Similarly, the subexpression (5 - 4 * 1) only has constant operands, and can also be evaluated at compile-time.

Before any code is generated, the trees for the above assignment could be reduced to those shown in Figure 15.2.

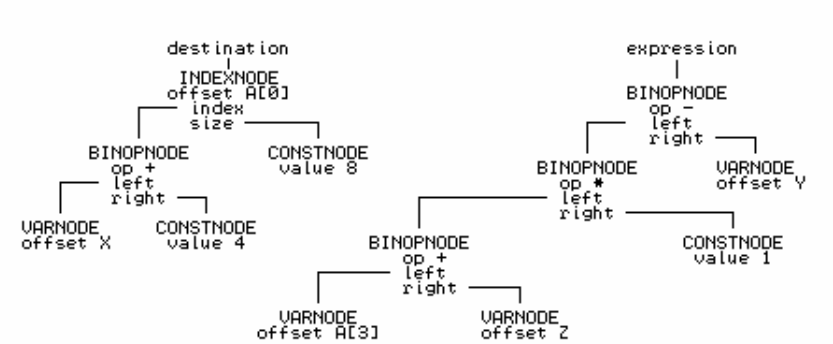

Figure 15.2 Improved AST structures for the statement A[X + 4] := (A[3] + Z) * (5 - 4 * 1) - Y

These sorts of manipulations fall into the category known as *constant folding*. They are easily added to the tree- building process, but are rather harder to do if code is generated on the fly. Constant folding is implemented by writing tree-building routines modelled on the following:

```
void CGEN::subscript(AST &base, int offset, AST &size, AST &index)
{ if (!index || !index->defined           // check for well defined
      || !size || !size->defined)         // trees and constant index
  { base = new INDEXNODE(offset, size, index); return; }
  if (unsigned(index->value) >= size->value)
    Report->error(223);                   // report range error immediately
  else                                    // simple variable designator
    base = new VARNODE(offset + index->value);
  delete index; delete size;              // and delete the unused debris
}

void CGEN::binaryop(CGEN_operators op, AST &left, AST &right)
{ if (left && right)                      // beware of corrupt trees
  { if (left->defined && right->defined)  // both operands are constant
    { switch (op)                         // so do compile-time evaluation
      { case CGEN_opadd: left->value += right->value; break;
        case CGEN_opsub: left->value -= right->value; break;
                                          // ... others like this
      }
      delete right; return;               // discard one operand
    }
  }
  left = new BINOPNODE(op, left, right);  // construct proper bin op node
}
```

The reader should notice that such constant folding is essentially machine independent (assuming that the arithmetic can be done at compile-time to the required precision). Tree construction represents the last phase of a machine-independent front end to a compiler; the routines that walk the tree become machine dependent.

Recognition and evaluation of expressions in which every operand is constant is useful if one

wishes to extend the language in other ways. For example, we may now easily extend our Clang language to allow for constant expressions within *ConstDeclarations*:

```
ConstDeclarations =  "CONST" OneConst { OneConst } .
OneConst          =  identifier "=" ConstExpression ";" .
ConstExpression   =  Expression .
```

We can make use of the existing parsing routines to handle a *ConstExpression*. The attributes in a Cocol specification would simply incorporate a constraint check that the expression was, indeed, "defined", and if so, store the "value" in the symbol table.

15.3.4 Simple optimizations - removal of redundant code

Production quality compilers often expend considerable effort in the detection of structures for which no code need be generated at all. For example, a source statement of the form

```
WHILE TRUE DO Something
```

does not require the generation of code like

```
LAB  IF NOT TRUE GOTO EXIT END
     Something
     GOTO LAB
EXIT
```

but can be reduced to

```
LAB  Something
     GOTO LAB
```

and, to take a more extreme case, if it were ever written, source code like

```
WHILE 15 < 6 DO Something
```

could be disregarded completely. Once again, optimizations of this sort are most easily attempted after an internal representation of the source program has been created in the form of a tree or graph. A full discussion of this fascinating subject is beyond the scope of this text, and it will suffice merely to mention a few improvements that might be incorporated into a simple tree-walking code generator for expressions. For example, the remaining multiplication by 1 in the expression we have used for illustration is redundant, and is easily eliminated. Similarly, multiplications by small powers of 2 could be converted into shift operations if the machine supports these, and multiplication by zero could be recognized as a golden opportunity to load a constant of 0 rather than perform any multiplications at all. To exemplify this, consider an extract from an improved routine that generates code to load the value resulting from a binary operation onto the run-time stack of our simple machine:

```
void BINOPNODE::emit1(void)
// load value onto stack resulting from binary operation
{ bool folded = false;
  if (left && right)                        // beware of corrupt trees
  { switch (op)                             // redundant operations?
    { case CGEN_opadd:
        if (right->defined && right->value == 0) // x + 0 = x
          { left->emit1(); folded = true; }
                                            // ... other special cases
        break;
      case CGEN_opsub:
                                            // ... other special cases
      case CGEN_opmul:
        if (right->defined && right->value == 1) // x * 1 = x
          { left->emit1(); folded = true; }
        else if (right->defined && right->value == 0) // x * 0 = 0
          { right->emit1(); folded = true; }
                                            // ... other special cases
```

```
          break;
        case CGEN_opdvd:
                                          // ... other special cases
      }
    }
    if (!folded)                          // still have to generate code
    { if (left)  left->emit1();           // beware of corrupt trees
      if (right) right->emit1();
      CGen->emit(int(STKMC_add) + int(op));   // careful - ordering used
    }
    delete left; delete right;            // remove debris
  }
```

These sorts of optimizations can have a remarkable effect on the volume of code that is generated - assuming, of course, that the expressions are littered with constants.

So far we have assumed that the structures set up as we parse expressions are all binary trees - each node has subtrees that are disjoint. Other structures are possible, although creating these calls for routines more complex than we have considered up till now. If we relax the restriction that subtrees must be disjoint, we introduce the possibility of using a so-called **directed acyclic graph** (DAG). This finds application in optimizations in which common subexpressions are identified, so that code for them is generated as few times as possible. For example, the expression *(a * a + b * b) / (a * a - b * b)* could be optimized so as to compute each of *a * a* and *b * b* only once. A binary tree structure and a DAG for this expression are depicted in Figure 15.3, but further treatment of this topic is beyond the scope of this text.

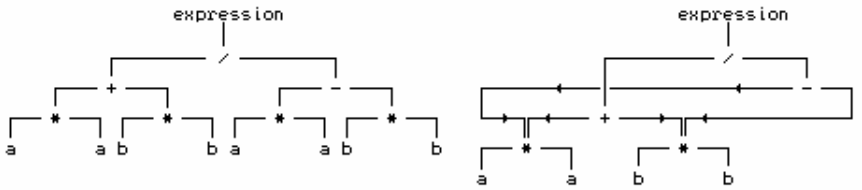

Figure 15.3 (a) Binary tree corresponding to expression
(a * a + b * b) / (a * a - b * b)
(b) DAG corresponding to expression
(a * a + b * b) / (a * a - b * b)

15.3.5 Generation of assembler code

We should also mention another approach often taken in providing native code compilers, especially on small machines. This is to generate output in the form of assembler code that can then be processed in a second pass using a macro assembler. Although carrying an overhead in terms of compilation speed, this approach has some strong points - it relieves the compiler writer of developing intensely machine dependent bit manipulating code (very tiresome in some languages, like the original Pascal), handling awkward forward referencing problems, dealing with operating system and linkage conventions, and so forth. It is widely used on Unix systems, for example.

On the source diskette can be found such a code generator. This can be used to construct a compiler that will translate Clang programs into the ASSEMBLER language for the tiny single-accumulator machine discussed in Chapter 4, and for which assemblers were developed in Chapter 6. Clearly there is a very real restriction on the size of source program that can be handled by this system, but the code generator employs several optimizations of the sort discussed earlier, and is an entertaining example of code that the reader is encouraged to study. Space does not permit of a full description, but the following points are worth emphasizing:

- An on-the-fly code generator for this machine would be very difficult to write, but the Cocol description of the phrase structure grammar can remain exactly the same as that used for the

stack machine. Naturally, the internal definitions of some members of the node classes are different, as are the implementations of the tree- walking member functions.

- The single-accumulator machine has conditional branching instructions that are very different from those used in the stack machine; it also has a rather non-orthogonal set of these. This calls for some ingenuity in the generation of code for *Conditions*, *IfStatements* and *WhileStatements*.

- The problem of handling the forward references needed in conditional statements is left to the later assembler stage. However, the code generator still has to solve the problem of generating a self-consistent set of labels for those instructions that need them.

- The input/output facilities of the two machines are rather disparate. In particular the single-accumulator machine does not have an special operation for writing strings. This is handled by arranging for the code generator to create and call a standard output subroutine for this purpose when it is required. The approach of generating calls to standardized library routines is, of course, very widespread in real compilers.

- Although capable of handling access to array elements, the code generator does not generate any run-time subscript checks, as these would be prohibitively expensive on such a tiny machine.

- The machine described in Chapter 4 does not have any operations for handling multiplication and division. A compiler error is reported if it appears that such operations are needed.

Exercises

Implementations of tree-based code generators for our simple stack machine can be found on the source diskette, as can the parsers and Cocol grammars that match these. The Modula-2 and Pascal implementations make use of variant records for discriminating between the various classes of nodes; C++ versions of these are also available.

15.25 If you program in Modula-2 or Pascal and have access to an implementation that supports OOP extensions to these languages, derive a tree-walking code generator based on the C++ model.

15.26 The constant folding operations perform little in the way of range checks. Improve them.

15.27 Adapt the tree-walking code generator for the stack machine to support the extensions you have made to Clang or Topsy.

15.28 Adapt the tree-walking code generator for the single-accumulator machine to support the extensions you have made to Clang or Topsy.

15.29 Extend the single-accumulator machine to support multiplication and division, and extend the code generator for Clang or Topsy to permit these operations (one cannot do much multiplication and division in an 8-bit machine, but it is the principle that matters here).

15.30 Follow up the suggestion made earlier, and extend Clang or Topsy to allow constant expressions to appear in constant declarations, for example

```
CONST
  Max = 100;
  Limit = 2 * Max + 1;
  NegMax = - Max;
```

15.31 Perusal of our example assignment should suggest the possibility of producing a still smaller tree for the right-hand side expression (Figure 15.4(a)). And, were the assignment to have been

```
A[X + 4] := (A[3] + Z) * (4 - 4 * 1) - Y
```

perhaps we could do better still (see Figure 15.4(b)). How would you modify the tree-building routines to achieve this sort of improvement? Can you do this in a way that still allows your compiler to support the notion of a constant expression as part of a constant declaration?

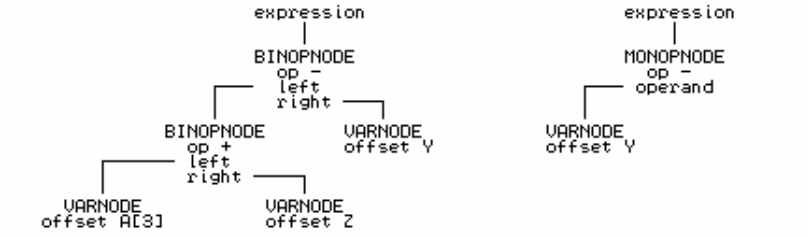

Figure 15.4 Optimal AST structures for the expressions
(a) (A[3] + Z) * (5 - 4 * 1) - Y (b) (A[3] + Z) * (4 - 4 * 1) - Y

15.32 (More extensive) Modify the attributed grammar and tree-building code generator so that node classes are introduced for the various categories of *Statement*. Then develop code generator routines that can effect the sorts of optimizations hinted at earlier for removing redundant code for unreachable components of *IfStatements* and *WhileStatements*. Sophisticated compilers often issue warnings when they discover code that can never be executed. Can you incorporate such a feature into your compiler?

15.33 (Harder) Use a tree-based representation to generate code for Boolean expressions that require short- circuit semantics (see Exercises 13.10 and 15.19).

15.34 (More extensive) Develop a code generator for a register-based machine such as that suggested in section 13.3. Can you do this without altering the Cocol specification, as we claim is possible for the single-accumulator machine of Chapter 4?

15.35 Many development environments incorporate "debuggers" - sophisticated tools that will trace the execution of a compiled program in conjunction with the source code, referring run-time errors to source code statements, allowing the user to interrogate (and even alter) the values of variables by using the identifiers of the source code, and so on. Development of such a system could be a very open- ended project. As a less ambitious project, extend the interpretive compiler for Clang or Topsy that, in the event of a run-time error, will relate this to the corresponding line in the source, and then print a post-mortem dump showing the values of the variables at the time the error occurred. A system of this sort was described for the well-known subset of Pascal known as Pascal-S (Wirth, 1981; Rees and Robson, 1987), and is also used in the implementation of the simple teaching language Umbriel (Terry, 1995).

15.36 Develop a code generator that produces correct C or C++ code from Clang or Topsy source.

Further reading

Our treatment of code generation has been dangerously superficial. "Real" code generation tends to become highly machine dependent, and the literature reflects this. Although all of the standard texts have a lot to say on the subject, those texts which do not confine themselves to generalities (by stopping short of showing how it is actually done) inevitably relate their material to one or other real machine, which can become confusing for a beginner who has little if any experience of that machine. Watson (1989) has a very readable discussion of the use of tree structures. Considerably more detail is given in the comprehensive books by Aho, Sethi and Ullman (1986) and Fischer and LeBlanc (1988, 1991). Various texts discuss code generation for familiar microprocessors. For example, the book by Mak (1991) develops a Pascal compiler that generates assembler code for the Intel 80x86 range of machines, and the book by Ullman (1994) develops a subset Modula-2 compiler that generates a variant of Intel assembler. The recent book by Holmes (1995) uses object orientation to develop a Pascal compiler, discussing the generation of assembler code for a SUN SPARC workstation. Wirth (1996) presents a tightly written account of developing a compiler for a subset of Oberon that generates code for a slightly idealized processor, modelled on the hypothetical RISC processor named DLX by Hennessy and Patterson (1990) to resemble the MIPS processor. Elder (1994) gives a thorough description of many aspects of code generation for a more advanced stack-based machine than the one described here.