13 USING COCO/R - CASE STUDIES

The best way to come to terms with the use of a tool like Coco/R is to try to use it, so in this chapter we make use of several case studies to illustrate how simple and powerful a tool it really is.

13.1 Case study - Understanding C declarations

It is generally acknowledged, even by experts, that the syntax of declarations in C and C++ can be quite difficult to understand. This is especially true for programmers who have learned Pascal or Modula-2 before turning to a study of C or C++. Simple declarations like

```
int x, list[100];
```

present few difficulties (`x` is a scalar integer, `list` is an array of 100 integers). However, in developing more abstruse examples like

```
char **a;      // a is a pointer to a pointer to a character
int *b[10];    // b is an array of 10 pointers to single integers
int (*c)[10];  // c is a pointer to an array of 10 integers
double *d();   // d is a function returning a pointer to a double
char (*e)();   // e is a pointer to a function returning a character
```

it is easy to confuse the placement of the various brackets, parentheses and asterisks, perhaps even writing syntactically correct declarations that do not mean what the author intended. By the time one is into writing (or reading) declarations like

```
short (*(*f())[])();
double (*(*g[50])())[15];
```

there may be little consolation to be gained from learning that C was designed so that the syntax of declarations (defining occurrences) should mirror the syntax for access to the corresponding quantities in expressions (applied occurrences).

Algorithms to help humans unravel such declarations can be found in many text books - for example, the recent excellent one by King (1996), or the original description of C by Kernighan and Ritchie (1988). In this latter book can be found a hand-crafted recursive descent parser for converting a subset of the possible declaration forms into an English description. Such a program is very easily specified in Cocol.

The syntax of the restricted form of declarations that we wish to consider can be described by

```
Decl      =  { name Dcl ";" } .
Dcl       =  { "*" } DirectDcl .
DirectDcl =    name
             | "(" Dcl ")"
             | DirectDcl "(" ")"
             | DirectDcl "[" [ number ] "]" .
```

if we base the productions on those found in the usual descriptions of C, but change the notation to match the one we have been using in this book. Although these productions are not in LL(1) form, it is easy to find a way of eliminating the troublesome left recursion. It also turns out to be expedient to rewrite the production for *Dcl* so as to use right recursion rather than iteration:

```
Decl      =  { name Dcl ";" } .
```

```
Dcl       =  "*" Dcl | DirectDcl .
DirectDcl =  ( name | "(" Dcl ")" ) { Suffix } .
Suffix    =  "(" ")" | "[" [ number ] "]" .
```

When adding attributes we make use of ideas similar to those already seen for the conversion of infix expressions into postfix form in section 11.1. We arrange to read the token stream from left to right, writing descriptions of some tokens immediately, but delaying the output of descriptions of others. The full Cocol specification follows readily as

```
$CX   /* Generate Main Module, C++ */
COMPILER Decl
#include <stdlib.h>
#include <iostream.h>

CHARACTERS
  digit =  "0123456789" .
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyx_" .

IGNORE CHR(9) .. CHR(13)

TOKENS
  number = digit { digit } .
  name = letter { letter } .

PRODUCTIONS
  Decl
  =                     (. char Tipe[100]; .)
    { name              (. LexString(Tipe, sizeof(Tipe) - 1); .)
      Dcl               (. cout << ' ' << Tipe << endl; .)
      ";" } .

  Dcl
  =   "*" Dcl           (. cout << " pointer to"; .)
    | DirectDcl .

  DirectDcl
  =                     (. char Name[100]; .)
    (   name            (. LexString(Name, sizeof(Name) - 1);
                           cout << ' ' << Name << " is"; .)
      | "(" Dcl ")"
    ) { Suffix } .

  Suffix
  =                     (. char buff[100]; .)
      "["               (. cout << " array ["; .)
       [ number         (. LexString(buff, sizeof(buff) - 1);
                           cout << atoi(buff); .)
       ]
      "]"               (. cout << "] of"; .)
    | "(" ")"           (. cout << " function returning"; .) .

END Decl.
```

Exercises

13.1 Perusal of the original grammar (and of the equivalent LL(1) version) will suggest that the following declarations would be allowed. Some of them are, in fact, illegal in C:

```
int f()[100];  // Functions cannot return arrays
int g()();     // Functions cannot return functions
int x[100]();  // We cannot declare arrays of functions
int p[12][20]; // We are allowed arrays of arrays
int q[][100];  // We are also allowed to declare arrays like this
int r[100][];  // We are not allowed to declare arrays like this
```

Can you write a Cocol specification for a parser that accepts only the valid combinations of suffixes? If not, why not?

13.2 Extend the grammar to cater for the declaration of more than one item based on the same type, as exemplified by

```
int f[100], *x, (*g)[100];
```

13.3 Extend the grammar and the parser to allow function prototypes to describe parameter lists, and to allow variable declarators to have initializers, as exemplified by

```
int x = 10, y[3] = { 4, 5, 6 };
int z[2][2] = {{ 4, 5 }, { 6, 7 }};
double f(int x, char &y, double *z);
```

13.4 Develop a system that will do the reverse operation - read in a description of a declaration (such as might be output from the program we have just discussed) and construct the C code that corresponds to this.

13.2 Case study - Generating one-address code from expressions

The simple expression grammar is, understandably, very often used in texts on programming language translation. We have already seen it used as the basis of a system to convert infix to postfix (section 11.1), and for evaluating expressions (section 11.2). In this case study we show how easy it is to attribute the grammar to generate one- address code for a multi-register machine whose instruction set supports the following operations:

```
LDI Rx,value     ;  Rx := value (immediate)
LDA Rx,variable  ;  Rx := value of variable (direct)
ADD Rx,Ry        ;  Rx := Rx + Ry
SUB Rx,Ry        ;  Rx := Rx - Ry
MUL Rx,Ry        ;  Rx := Rx * Ry
DVD Rx,Ry        ;  Rx := Rx / Ry
```

For this machine we might translate some example expressions into code as follows:

```
a + b       5 * 6       x / 12       (a + b) * (c - 5)

LDA R1,a    LDI R1,5    LDA R1,x     LDA  R1,a    ;  R1 := a
LDA R2,b    LDI R2,6    LDI R2,12    LDA  R2,b    ;  R2 := b
ADD R1,R2   MUL R1,R2   DVD R1,R2    ADD  R1,R2   ;  R1 := a+b
                                     LDA  R2,c    ;  R2 := c
                                     LDI  R3,5    ;  R3 := 5
                                     SUB  R2,R3   ;  R2 := c-5
                                     MUL  R1,R2   ;  R1 := (a+b)*(c-5)
```

If we make the highly idealized assumption that the machine has an inexhaustible supply of registers (so that any values may be used for *x* and *y*), then an expression compiler becomes almost trivial to specify in Cocol.

```
$CX /* Compiler, C++ */
COMPILER Expr

CHARACTERS
  digit    = "0123456789" .
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .

IGNORE CHR(9) .. CHR(13)

TOKENS
  number   = digit { digit } .
  variable = letter .

PRODUCTIONS
  Expr
  = { Expression<1> SYNC ";"   (. printf("\n"); .)
    } .

  Expression<int R>
  = Term<R>
    {   "+" Term<R+1>          (. printf("ADD R%d,R%d\n", R, R+1); .)
```

```
      | "-" Term<R+1>          (. printf("SUB R%d,R%d\n", R, R+1); .)
    } .

  Term<int R>
  = Factor<R>
    {   "*" Factor<R+1>        (. printf("MUL R%d,R%d\n", R, R+1); .)
      | "/" Factor<R+1>        (. printf("DVD R%d,R%d\n", R, R+1); .)
    } .

  Factor<int R>
  =                            (. char CH; int N; .)
      Identifier<CH>           (. printf("LDA R%d,%c\n", R, CH); .)
    | Number<N>                (. printf("LDI R%d,%d\n", R, N); .)
    | "(" Expression<R> ")" .

  Identifier<char &CH>
  =  variable                  (. char str[100];
                                  LexString(str, sizeof(str) - 1);
                                  CH = str[0]; .) .

  Number<int &N>
  =  number                    (. char str[100];
                                  LexString(str, sizeof(str) - 1);
                                  N = atoi(str); .) .

END Expr.
```

The formal attribute to each routine is the number of the register in which the code generated by that routine is required to store the value for whose computation it is responsible. Parsing starts by assuming that the final value is to be stored in register 1. A binary operation is applied to values in registers *x* and *x* + 1, leaving the result in register *x*. The grammar is factorized, as we have seen, in a way that correctly reflects the associativity and precedence of the parentheses and arithmetic operators as they are found in infix expressions, so that, where necessary, the register numbers increase steadily as the parser proceeds to decode complex expressions.

Exercises

13.5 Use Coco/R to develop a program that will convert infix expressions to postfix form.

13.6 Use Coco/R to develop a program that will evaluate infix arithmetic expressions directly.

13.7 The parser above allows only single character variable names. Extend it to allow variable names that consist of an initial letter, followed by any sequence of digits and letters.

13.8 Suppose that we wished to be able to generate code for expressions that permit leading signs, as for example + *x* * (- *y* + *z*). Extend the grammar to describe such expressions, and then develop a program that will generate appropriate code. Do this in two ways (a) assume that there is no special machine instruction for negating a register (b) assume that such an operation is available (`NEG Rx`).

13.9 Suppose the machine also provided logical operations:

```
AND Rx,Ry        ;  Rx := Rx AND Ry
OR  Rx,Ry        ;  Rx := Rx OR Ry
XOR Rx,Ry        ;  Rx := Rx XOR Ry
NOT Rx           ;  Rx := NOT Rx
```

Extend the grammar to allow expressions to incorporate infix and prefix logical operations, in addition to arithmetic operations, and develop a program to translate them into simple machine code. This will require some decision as to the relative precedence of all the operations. NOT always takes precedence over AND, which in turn takes precedence over OR. In Pascal and

Modula-2, NOT, AND and OR are deemed to have precedence equal to unary negation, multiplication and addition (respectively). However, in C and C++, NOT has precedence equal to unary negation, while AND and OR have lower precedence than the arithmetic operators - the 16 levels of precedence in C, like the syntax of declarations, are another example of baroque language design that cause a great difficulty to beginners. Choose whatever relative precedence scheme you prefer, or better still, attempt the exercise both ways.

13.10 (Harder). Try to incorporate short-circuit Boolean semantics into the language suggested by Exercise 13.9, and then use Coco/R to write a translator for it. The reader will recall that these semantics demand that

```
A AND B   is defined to mean   IF A THEN B ELSE FALSE
A OR  B   is defined to mean   IF A THEN TRUE ELSE B
```

that is to say, in evaluating the AND operation there is no need to evaluate the second operand if the first one is found to be FALSE, and in evaluating the OR operation there is no need to evaluate the second operand if the first is found to be TRUE. You may need to extend the instruction set of the machine to provide conditional and other branch instructions; feel free to do so!

13.11 It is unrealistic to assume that one can simply allocate registers numbered from 1 upwards. More usually a compiler has to select registers from a set of those known to be free at the time the expression evaluation commences, and to arrange to release the registers once they are no longer needed for storing intermediate values. Modify the grammar (and hence the program) to incorporate this strategy. Choose a suitable data structure to keep track of the set of available registers - in Pascal and Modula-2 this becomes rather easy; in C++ you could make use of the template class for set handling discussed briefly in section 10.3.

13.12 It is also unreasonable to assume that the set of available registers is inexhaustible. What sort of expression requires a large set of registers before it can be evaluated? How big a set do you suppose is reasonable? What sort of strategy do you suppose has to be adopted if a compiler finds that the set of available registers becomes exhausted?

13.3 Case study - Generating one-address code from an AST

It should not take much imagination to realize that code generation for expression evaluation using an "on-the fly" technique like that suggested in section 13.2, while easy, leads to very inefficient and bloated code - especially if, as is usually the case, the machine instruction set incorporates a wider range of operations. If, for example, it were to include direct and immediate addressing operations like

```
ADD Rx,variable  ;  Rx := Rx + value of variable
SUB Rx,variable  ;  Rx := Rx - value of variable
MUL Rx,variable  ;  Rx := Rx * value of variable
DVD Rx,variable  ;  Rx := Rx / value of variable

ADI Rx,constant  ;  Rx := Rx + value of constant
SBI Rx,constant  ;  Rx := Rx - value of constant
MLI Rx,constant  ;  Rx := Rx * value of constant
DVI Rx,constant  ;  Rx := Rx / value of constant
```

then we should be able to translate the examples of code shown earlier far more effectively as follows:

```
a + b       5 * 6       x / 12       (a + b) * (c - 5)

LDA R1,a    LDI R1,30   LDA R1,x     LDA  R1,a    ;  R1 := a
ADD R1,b                DVI R1,12    ADD  R1,b    ;  R1 := a + b
                                     LDA  R2,c    ;  R2 := c
                                     SBI  R2,5    ;  R2 := c - 5
                                     MUL  R1,R2   ;  R1 := (a+b)*(c-5)
```

To be able to generate such code requires that we delay the choice of instruction somewhat - we should no longer simply emit instructions as soon as each operator is recognized (once again we can see a resemblance to the conversion from infix to postfix notation). The usual strategy for achieving such optimizations is to arrange to build an abstract syntax tree (AST) from the expression, and then to "walk" it in LRN (post) order, emitting machine code apposite to the form of the operation associated with each node. An example may make this clearer. The tree corresponding to the expression (*a* + *b*) * (*c* - 5) is shown in Figure 13.1.

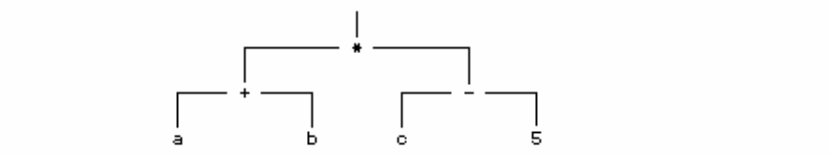


Figure 13.1 An AST corresponding to the expression (a + b) * (c - 5)

The code generating operations needed as each node is visited are depicted in Figure 13.2.

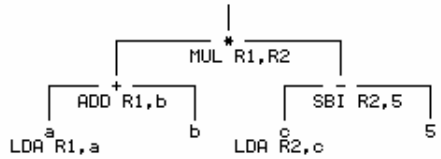


Figure 13.2 Code generation needed when visiting each node in an AST

It is, in fact, remarkably easy to attribute our grammar so as to incorporate tree-building actions instead of immediate code generation:

```
$CX /* Compiler, C++ */
COMPILER Expr
/* Convert infix expressions into machine code using a simple AST */

#include "trees.h"

CHARACTERS
  digit  =  "0123456789" .
  letter =  "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .

IGNORE CHR(9) .. CHR(13)

TOKENS
  number = digit { digit } .
  variable = letter .

PRODUCTIONS
  Expr
  =                           (. AST Exp; .)
    { Expression<Exp>
      SYNC ";"                (. if (Successful()) GenerateCode(Exp); .)
    } .

  Expression<AST &E>
  =                           (. AST T; .)
    Term<E>
    {   "+" Term<T>           (. E = BinOpNode(Plus, E, T); .)
      | "-" Term<T>           (. E = BinOpNode(Minus, E, T); .)
    } .
```

```
  Term<AST &T>
  =                          (. AST F; .)
    Factor<T>
    {   "*" Factor<F>        (. T = BinOpNode(Times, T, F); .)
      | "/" Factor<F>        (. T = BinOpNode(Slash, T, F); .)
    } .

  Factor<AST &F>
  =                          (. char CH; int N; .)
                             (. F = EmptyNode(); .)
  (   Identifier<CH>         (. F = VarNode(CH); .)
    | Number<N>              (. F = ConstNode(N); .)
    | "(" Expression<F> ")"
  ) .

  Identifier<char &CH>
  =  variable                (. char str[100];
                                LexName(str, sizeof(str) - 1);
                                CH = str[0]; .) .

  Number<int &N>
  =  number                  (. char str[100];
                                LexString(str, sizeof(str) - 1);
                                N = atoi(str); .) .

END Expr.
```

Here, rather than pass register indices as "value" parameters to the various parsing routines, we arrange that they each return an AST (as a "reference" parameter) - essentially a pointer to a structure created as each *Expression*, *Term* or *Factor* is recognized. The *Factor* parser is responsible for creating the leaf nodes, and these are stitched together to form larger trees as a result of the iteration components in the *Expression* and *Term* parsers. Once the tree has been built in this way - that is, after the goal symbol has been completely parsed - we can walk it so as to generate the code.

The reader may feel a bit cheated, as this does not reveal very much about how the trees are really constructed. However, that is in the spirit of "data abstraction"! The grammar above can be used unaltered with a variety of implementations of the AST tree handling module. In compiler technology terminology, we have succeeded in separating the "front end" or parser from the "back end" or tree-walker that generates the code. By providing machine specific versions of the tree-walker we can generate code for a variety of different machines, indulge in various optimization techniques, and so on. The AST tree-builder and tree-walker have the following interface:

```
enum optypes { Load, Plus, Minus, Times, Slash };

class NODE;

typedef NODE* AST;

AST BinOpNode(optypes op, AST left, AST right);
// Creates an AST for the binary operation "left op right"

AST VarNode(char name);
// Creates an AST for a variable factor with specified name

AST ConstNode(int value);
// Creates an AST for a constant factor with specified value

AST EmptyNode();
// Creates an empty node

void GenerateCode (AST A);
// Generates code from AST A
```

Here we are defining an `AST` type as a pointer to a (dynamically allocated) `NODE` object. The functions exported from this interface allow for the construction of several distinct varieties of nodes, of course, and in particular (a) an "empty" node (b) a "constant" node (c) a "variable" node and (d) a "binary operator" node. There is also a routine that can walk the tree, generating code as

each node is visited.

In traditional implementations of this module we should have to resort to constructing the `NODE` type as some sort of variant record (in Modula-2 or Pascal terminology) or union (in C terminology), and on the source diskette can be found examples of such implementations. In languages that support object-oriented programming it makes good sense to define the `NODE` type as an abstract base class, and then to derive the other types of nodes as sub- classes or derived classes of this type. The code below shows one such implementation in C++ for the generation of code for our hypothetical machine. On the source diskette can be found various class based implementations, including one that generates code no more sophisticated than was discussed in section 13.2, as well as one matching the same interface, but which generates code for the single-accumulator machine introduced in Chapter 4. There are also equivalent implementations that make use of the object-oriented extensions found in Turbo Pascal and various dialects of Modula-2.

```
// Abstract Syntax Tree facilities for simple expression trees
// used to generate reasonable one-address machine code.

#include <stdio.h>
#include "trees.h"

class NODE
{ friend AST BinOpNode(optypes op, AST left, AST right);
  friend class BINOPNODE;
  public:
    NODE()                              { defined = 0; }
    virtual void load(int R) = 0;
    // Generate code for loading value of a node into register R
  protected:
    int value;     // value derived from this node
    int defined;   // 1 if value is defined
    virtual void operation(optypes O, int R) = 0;
    virtual void loadreg(int R)        {;}
};

class BINOPNODE : public NODE
{ public:
    BINOPNODE(optypes O, AST L, AST R)   { op = O; left = L; right = R; }
    virtual void load(int R);
  protected:
    optypes op;
    AST left, right;
    virtual void operation(optypes O, int R);
    virtual void loadreg(int R)        { load(R); }
};

void BINOPNODE::operation(optypes op, int R)
{ switch (op)
  { case Load:  printf("LDA"); break;
    case Plus:  printf("ADD"); break;
    case Minus: printf("SUB"); break;
    case Times: printf("MUL"); break;
    case Slash: printf("DVD"); break;
  }
  printf(" R%d,R%d\n", R, R + 1);
}

void BINOPNODE::load(int R)
{ if (!left || !right) return;
  left->load(R); right->loadreg(R+1); right->operation(op, R);
  delete left; delete right;
}

AST BinOpNode(optypes op, AST left, AST right)
{ if (left && right && left->defined && right->defined)
  { // constant folding
    switch (op)
    { case Plus:  left->value += right->value; break;
      case Minus: left->value -= right->value; break;
      case Times: left->value *= right->value; break;
      case Slash: left->value /= right->value; break;
    }
    delete right; return left;
  }
  return new BINOPNODE(op, left, right);
```

```
}

class VARNODE : public NODE
{  public:
    VARNODE(char C)                    { name = C; }
    virtual void load(int R)           { operation(Load, R); }
  protected:
    char name;
    virtual void operation(optypes O, int R);
};

void VARNODE::operation(optypes op, int R)
{ switch (op)
  { case Load:  printf("LDA"); break;
    case Plus:  printf("ADD"); break;
    case Minus: printf("SUB"); break;
    case Times: printf("MUL"); break;
    case Slash: printf("DVD"); break;
  }
  printf(" R%d,%c\n", R, name);
}

AST VarNode(char name)
{ return new VARNODE(name); }

class CONSTNODE : public NODE
{ public:
    CONSTNODE(int V)                   { value = V; defined = 1; }
    virtual void load(int R)           { operation(Load, R); }
  protected:
    virtual void operation(optypes O, int R);
};

void CONSTNODE::operation(optypes op, int R)
{ switch (op)
  { case Load:  printf("LDI"); break;
    case Plus:  printf("ADI"); break;
    case Minus: printf("SBI"); break;
    case Times: printf("MLI"); break;
    case Slash: printf("DVI"); break;
  }
  printf(" R%d,%d\n", R, value);
}

AST ConstNode(int value)
{ return new CONSTNODE(value); }

AST EmptyNode()
{ return NULL; }

void GenerateCode(AST A)
{ A->load(1); printf("\n"); }
```

The reader's attention is drawn to several points that might otherwise be missed:

- We have deliberately chosen to implement a single `BINOPNODE` class, rather than using this as a base class from which were derived `ADDNODE, SUBNODE, MULNODE` and `DIVNODE` classes. The alternative approach makes for a useful exercise for the reader.

- When the `BinOpNode` routine constructs a binary node, some optimization is attempted. If both the left and right subexpressions are `defined`, that is to say, are represented by constant nodes, then arithmetic can be done immediately. This is known as **constant folding**, and, once again, is something that is far more easily achieved if an AST is constructed, rather than resorting to "on-the-fly" code generation. It often results in a saving of registers, and in shorter (and hence faster) object code.

- Some care must be taken to ensure that the integrity of the AST is preserved even if the source expression is syntactically incorrect. The *Factor* parser is arranged so as to return an empty node if it fails to recognize a valid member of FIRST(*Factor*), and there are various other checks in the code to ensure that tree walking is not attempted if such nodes have been incorporated into the tree (for example, in the `BINOPNODE::load` and `BinOpNode` routines).

Exercises

13.13 The constant folding demonstrated here is dangerous, in that it has assumed that arithmetic overflow will never occur. Try to improve it.

13.14 One disadvantage of the approach shown here is that the operators have been "hard wired" into the `optypes` enumeration. Extending the parser to handle other operations (such as AND and OR) would require modification in several places, which would be error-prone, and not in the spirit of extensibility that OOP techniques are meant to provide. If this strikes you as problematic, rework the `AST` handler to introduce further classes derived from `BINOPNODE`.

13.15 The tree handler is readily extended to perform other simple optimizations. For example, binary expressions like $x * 1$, $1 * x$, $x + 0$, $x * 0$ are quite easily detected, and the otherwise redundant operations can be eliminated. Try to incorporate some of these optimizations into the routines given earlier. Is it better to apply them while the tree is under construction, or when it is later walked?

13.16 Rework Exercises 13.8 through 13.12 to use abstract syntax trees for intermediate representations of source expressions.

13.4 Case study - How do parser generators work?

Our last case study aims to give the reader some insight into how a program like Coco/R might itself be developed. In effect, we wish to be able to develop a program that will take as input an LL(1) type grammar, and go on to construct a parser for that grammar. As we have seen, such grammars can be described in EBNF notation, and the same EBNF notation can be used to describe itself, rather simply, and in a form suitable for top- down parsing. In particular we might write

```
Syntax       =   { Production } "EOG" .
Production   =   NonTerminal "=" Expression "." .
Expression   =   Term { "|" Term } .
Term         =   [ Factor { Factor } ] .
Factor       =      NonTerminal
                |   Terminal
                |   "(" Expression ")" | "[" Expression "]"
                |   "{" Expression "}" .
```

where *NonTerminal* and *Terminal* would be chosen from a particular set of symbols for a grammar, and where the terminal "`EOG`" has been added to ease the task of recognizing the end of the grammar. It is left to the reader formally to show that this grammar is LL(1), and hence capable of being parsed by recursive descent.

A parser generator may be constructed by enriching this grammar, providing actions at appropriate points so as to construct, from the input data, some code (or similar structure which can later be "executed") either to parse other programs, or to construct parsers for those programs. One method of doing this, outlined by Wirth (1976b, 1986) and Rechenberg and Mössenböck (1989), is to develop the parser actions so that they construct a data structure that encapsulates a syntax diagram representation of the grammar as a graph, and then to apply a graph walker that traverses these syntax diagrams.

To take a particular example, consider the *ClassList* grammar of section 11.5, for which the productions are

```
ClassList  =  ClassName [ Group { ";" Group } ]  "." .
Group      =  Degree ":"  Student { "," Student } .
Degree     =  "BSc" | "BScS" .
ClassName  =  identifier .
Student    =  identifier .
```

A corresponding set of syntax diagrams for these productions is shown in Figure 13.3.

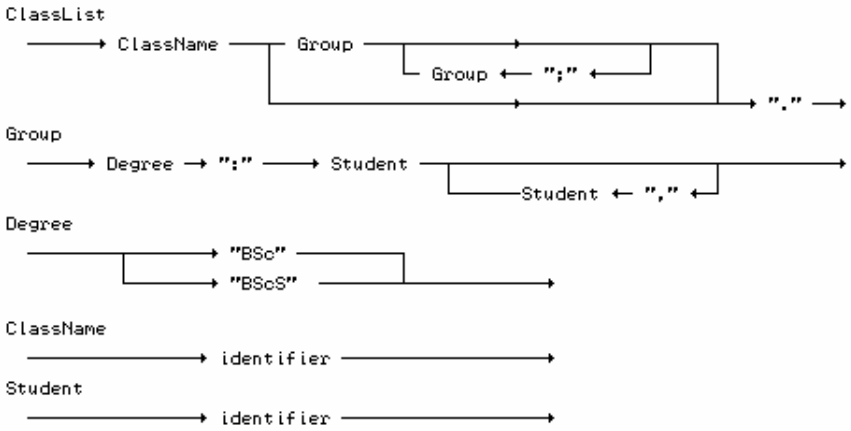


Figure 13.3 Syntax diagrams describing the Classlist grammar

Such graphs may be represented in programs by linked data structures. At the top level we maintain a linked list of nodes, each one corresponding to a non-terminal symbol of the grammar. For each such symbol in the grammar we then go on to introduce (for each of its alternative productions) a sub-graph of nodes linked together.

In these dependent graphs there are two basic types of nodes: those corresponding to terminal symbols, and those corresponding to non-terminals. Terminal nodes can be labelled by the terminal itself; non-terminal nodes can contain pointers back to the nodes in the non-terminal list. Both variants of graph nodes contain two pointers, one (`Next`) designating the symbol that follows the symbol "stored" at the node, and the other (`Alternate`) designating the next in a list of alternatives. Once again, the reader should be able to see that this lends itself to the fruitful adoption of OOP techniques - an abstract base class can be used for a node, with derived classes to handle the specializations.

As it turns out, one needs to take special cognizance of the empty terminal ε, especially in those situations where it appears implicitly through the "{" *Expression* "}" or "[" *Expression* "]" construction rather than through an explicit empty production.

The way in which the graphs are constructed is governed by four quite simple rules:

- A sequence of *Factors* generated by a *Term* gives rise to a list of nodes linked by their `Next` pointers, as shown in Figure 13.4(a);
- A succession of alternative *Terms* produced by an *Expression* gives rise to a list of nodes linked by their `Alternate` pointers, as shown in Figure 13.4(b);
- A loop produced by a factor of the form { *Expression* } gives rise to a structure of the form

shown in Figure 13.4(c);

- An option produced by a factor of the form [*Expression*] gives rise to a structure of the form shown in Figure 13.4(d).

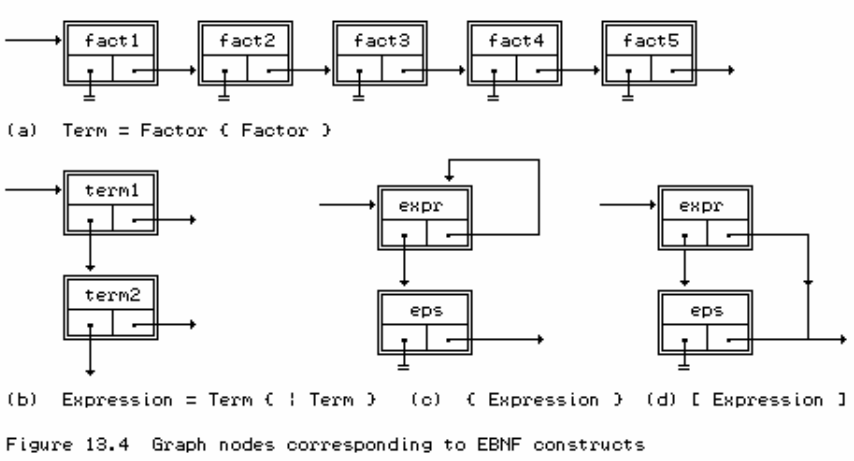


Figure 13.4 Graph nodes corresponding to EBNF constructs

As a complete example, the structures that correspond to our *ClassList* example lead to the graph depicted in Figure 13.5.

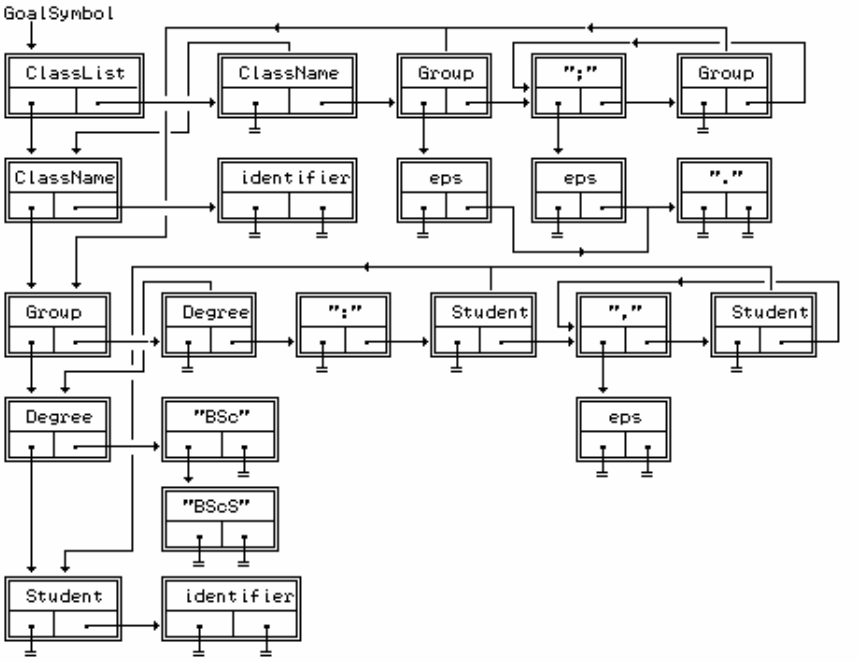


Figure 13.5 Graph depicting the ClassList grammar

Construction of the data structures is a non-trivial exercise - especially when they are extended further to allow for semantic attributes to be associated with the various nodes. As before, we have attempted to introduce a large measure of abstraction in the attributed Cocol grammar given below:

```
$CX /* compiler, C++ */
COMPILER EBNF
/* Augmented Coco/R grammar describing a set of EBNF productions
   and allowing the construction of a graph driven parser */

#include "misc.h"
#include "gp.h"
```

```
extern GP *GParser;

CHARACTERS
  cr       = CHR(13) .
  lf       = CHR(10) .
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  lowline  = "_" .
  digit    = "0123456789" .
  noquote1 = ANY - "'" - cr - lf .
  noquote2 = ANY - '"' - cr - lf .

IGNORE CHR(9) .. CHR(13)
IGNORE CASE

COMMENTS FROM "(*" TO "*)"  NESTED

TOKENS
  nonterminal = letter { letter | lowline | digit } .
  terminal    = "'" noquote1 { noquote1 } "'" | '"' noquote2 { noquote2 } '"' .
  EOG         = "$" .

PRODUCTIONS
  EBNF
  = { Production } EOG           (. bool haserrors = !Successful();
                                    GParser->checkgraph(stderr, haserrors);
                                    if (haserrors) SemError(200); .) .

  Production
  =                              (. GP_GRAPH rhs;
                                    GP_PROD lhs;
                                    char name[100]; .)
    NonTerminal<name>            (. GParser->startproduction(name, lhs); .)
    "=" Expression<rhs>          (. if (Successful())
                                      GParser->completeproduction(lhs, rhs); .)
    "." .

  Expression<GP_GRAPH &first>
  =                              (. GP_GRAPH next; .)
    Term<first>
    { "|" Term<next>             (. GParser->linkterms(first, next); .)
    } .

  Term<GP_GRAPH &first>
  =                              (. GP_GRAPH next; .)
    ( Factor<first>
      { Factor<next>             (. GParser->linkfactors(first, next); .)
      }
      |                          (. GParser->epsnode(first); .)
    ) .

  Factor<GP_GRAPH &node>
  =                              (. char name[100]; .)
      NonTerminal<name>          (. GParser->nonterminalnode(name, node); .)
    | Terminal<name>             (. GParser->terminalnode(name, node); .)
    | "[" Expression<node> "]"   (. GParser->optionalnode(node); .)
    | "{" Expression<node> "}"   (. GParser->repeatednode(node); .)
    | "(" Expression<node> ")" .

  NonTerminal<char *name>
  =  nonterminal                 (. LexName(name, 100); .) .

  Terminal<char *name>
  =  terminal                    (. char local[100];
                                    LexName(local, sizeof(local) - 1);
                                    int i = 0; /* strip quotes */
                                    while (local[i])
                                    { local[i] = local[i+1]; i++; }
                                    local[i-2] = '\0';
                                    strcpy(name, local); .) .

END EBNF.
```

The simplicity here is deceptive: this system has delegated control to various node creation and linker routines that are members of an instance `GParser` of a general graph parser class `GP`. It is predominantly the task of *Factor* (at the lowest point in the hierarchy) to call the routines to generate new actual nodes in the graph: the task of the routines called from other functions is to link them correctly (that called from *Term* uses the `Next` field, while *Expression* uses the `Alternate` field).

A non-terminal might appear in an *Expression* before it has appeared on the left side of a production. In this case it is still entered into the list of rules by a call to the *StartProduction* routine.

Once one has constructed these sorts of structures, what can be done with them? The idea of a graph-walker can be used in various ways. In Coco/R such graph-walkers are used in conjunction with the frame files, merging appropriately generated source code with these files to produce complete programs.

Further exploration

An implementation of the `GP` class, and of an associated scanner class `GS` has been provided on the source diskette, and will allow the reader to study these ideas in more detail. Be warned that the code, while quite concise, is not particularly easy to follow - and is still a long way short of being a program that can handle attributes and perform checks that the grammar submitted to it satisfies constraints like LL(1) conditions. Furthermore, the code does not demonstrate the construction of a complete parser generator, although it does show the development of a simple direct graph driven parser based on that suggested by Wirth (1976b, 1996).

This is actually a very naïve parsing algorithm, requiring rather special constraints on the grammar. It has the property of pursuing a new subgoal whenever it appears (by virtue of the recursive call to `ParseFrom`), without first checking whether the current symbol is in the set FIRST(*Goal->RightSide*). This means that the syntax must have been described in a rather special way - if a *NonTerminal* is nullable, then none of its right parts must start with a non-terminal, and each *Factor* (except possibly the last one) in the group of alternatives permitted by a *Term* must start with a distinct terminal symbol.

So, although this parser sometimes appears to work quite well - for example, for the *ClassList* grammar above it will correctly report that input sentences like

```
CS3 BSc : Tom, Dick ,, Harry .
CS3 BScS : Tom Dick .
```

are malformed - it will accept erroneous input like

```
CS3 BSc : .
```

as being correct. The assiduous reader might like to puzzle out why this is so.

The source code for Coco/R, its support modules, and the attributed grammar from which it is bootstrapped, are available from various Internet sites, as detailed in Appendix A. The really curious reader is encouraged to obtain copies of these if he or she wishes to learn more about Coco/R itself, or about how it is used in the construction of really large applications.

13.5 Project suggestions

Coco/R, like other parser generators, is a very powerful tool. Here are some suggestions for further projects that the reader might be encouraged to undertake.

13.17 The various expression parsers that have been used in earlier case studies have all assumed that the operands are simple integers. Suppose we wished to extend the underlying grammar to allow for comparison operations (which would operate on integer values but produce Boolean results), arithmetic operations (which operate on integer values and produce integer results) and logical operations (which act on Boolean values to produce Boolean results). A context-free grammar for such expressions, based on that used in Pascal and Modula-2, is given below. Incorporate this into an attributed Cocol grammar that will allow you to check whether expressions are semantically acceptable (that is, whether the operators have been applied in the correct context). Some examples follow

```
Acceptable                Not acceptable

3 + 4 * 6                 3 + 4 < 6
(x > y) AND (a < b)       x < y OR a < b
```

```
Expression       =  SimpleExpression [ RelOp SimpleExpression ] .
SimpleExpression =  Term { AddOp Term } .
Term             =  Factor { MulOp Factor } .
Factor           =   identifier | number  | "(" Expression ")"
                    | "NOT" Factor | "TRUE" | "FALSE" .
AddOp            =  "+" | "-" | "OR" .
MulOp            =  "*" | "/" | "AND"  .
RelOp            =  "<" | "<=" | ">" | ">=" | "=" | "<>" .
```

13.18 The "spreadsheet" has become a very popular tool in recent years. This projects aims to use Coco/R to develop a simple spreadsheet package.

A modern commercial package provides many thousands of features; we shall be less ambitious. In essence a simple two-dimensional spreadsheet is based on the concept of a matrix of cells, typically identified by a letter-digit pair (such as `E7`) in which the letter specifies a row, and the digit specifies a column. Part (or all) of this matrix is displayed on the terminal screen; one cell is taken as the *active cell*, and is usually highlighted in some way (for example, in inverse video).

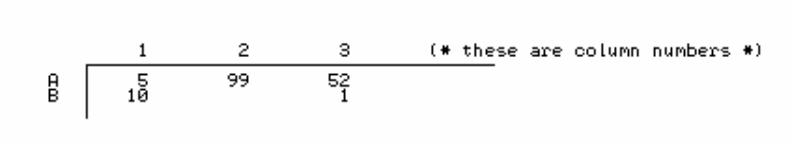
Input to a spreadsheet is then provided in the form of expressions typed by the user, interleaved with commands that can reselect the position of the active cell. Each time an expression is typed, its *formula* is associated with the active cell, and its *value* is displayed in the correct position. Changing the contents of one cell may affect the values of other cells. In a very simple spreadsheet implementation, each time one cell is assigned a new expression, the values of all the other cells are recomputed and redisplayed.

For this exercise assume that the expressions are confined to integer expressions of the sort exhaustively discussed in this text. The operands may be integer literals, or the designators of cells. No attempt need be made to handle string or character values.

A simple session with such a spreadsheet might be described as follows

```
(* we start in cell A1 *)
1 RIGHT              (* enter 1 in cell A1 and move on to cell A2 *)
99 RIGHT             (* enter 99 in cell A2 and move on to cell A3 *)
(A1 + A2) / 2  ENTER (* cell A3 contains the average of A1 and A2 *)
DOWN LEFT LEFT       (* move to cell B1 *)
2 * A1               (* cell B1 now contains twice the value of A1 *)
UP                   (* move back to cell A1 *)
5                    (* alter expression in A1 : A3 and B1 affected *)
GOTO B3              (* move to cell B3 *)
A3 % 3 ENTER         (* B3 contains remainder when A3 is divided by 3 *)
QUIT
```

At the point just before we quit, the grid displayed on the top left of the screen might display



It is possible to develop such a system using Coco/R in a number of ways, but it is suggested that you proceed as follows:

(a) Derive a context-free grammar that will describe the form of a session with the spreadsheet like that exemplified above.

(b) Enhance your grammar to provide the necessary attributes and actions to enable a complete system to be generated that will read and process a file of input and compute and display the spreadsheet, updating the display each time new expressions become associated with cells.

Make the following simplifying assumptions:

(a) A spreadsheet is normally run "interactively". However, Coco/R generates systems that most conveniently take their input from a disk file. If you want to work interactively you will need to modify the scanner frame file considerably.

(b) Assume that the spreadsheet has only 20 rows and 9 columns, extending from A1 through S9.

(c) Apart from accepting expressions typed in an obvious way, assume that the movement commands are input as `LEFT, RIGHT, UP, DOWN, HOME` and `GOTO Cell` as exemplified above. Assume that attempts to move too far in one direction either "wrap around" (so that a sequence like `GOTO A1  UP` results in cell `S1` becoming the active cell; `GOTO A12` actually moves to `A3`, and so on) or simply "stick" at the edge, as you please.

(d) An expression may also be terminated by `ENTER`, which does not affect the selection of the active cell.

(e) Input to the spreadsheet is terminated by the `QUIT` operation.

(f) The semantics of updating the spreadsheet display are captured in the following pseudo-code:

```
When Expression is recognized as complete
  Store Expression[CurrentRow, CurrentColumn] in a form
        that can be used for future interpretation
  Update value of Value[CurrentRow, CurrentColumn]
  FOR Row FROM A TO S DO
    FOR Column FROM 1 TO 9 DO
      Update Value[Row, Column] by
        evaluating Expression[Row, Column]
      Display new Value[Row, Column]
    END
  END
```

(g) Arrange that the spreadsheet starts with the values of each cell set to zero, and with no expressions associated with any cell.

(h) No facilities for "editing" an expression need be provided; if a cell’s expression is to be altered it must be typed afresh.

Hint: The most intriguing part of this exercise is deciding on a way to store an expression so that it can be evaluated again when needed. It is suggested that you associate a simple auxiliary data

structure with each cell of the spreadsheet. Each element of this structure can store an operation or operand for a simple interpreter.

13.19 A rather useful tool to have when dealing with large amounts of source code is a "cross reference generator". This is a program that will analyse the source text and produce a list of all the identifiers that appear in it, along with a list for each identifier of the line numbers on which it can be found. Construct a cross reference generator for programs written in Clang, for which a grammar was given in section 8.7, or for one of the variations on it suggested in Exercises 8.25 through 8.30. This can be done at various levels of sophistication; you should at least try to distinguish between the line on which an identifier is "declared", and those where it is "applied". A useful way to decompose the problem might be to develop a support module with an interface to a hidden data structure:

```
void Create();
// Initialize a new (empty) Table

void Add(char *Name, int Reference, bool Defining);
// Add Name to Table with given Reference, specifying whether
// this is a Defining (as opposed to an applied occurrence)

void List(FILE *lst);
// List out cross reference Table on lst file
```

You should then find that the actions needed to enhance the grammar are very straightforward, and the bulk of any programming effort falls on the development of a simple tree or queue-based data structure similar to those which you should have developed in other courses you have taken in Computer Science.

13.20 In case you have not met this concept before, a pretty printer is a "compiler" that takes a source program and "translates" the source into the same language. That probably does not sound very useful! However, the "object code" is formatted neatly and consistently, according to some simple conventions, making it far easier for humans to understand.

Develop a pretty printer for the simple Clang language for which the grammar was given in section 8.7. The good news is that you will not have to develop any semantic analysers, code generators, or symbol table handlers in this project, but can assume that the source program is semantically correct if it is syntactically correct. The bad news is that you may have some difficulty in retaining the comments. They can no longer be ignored, but should preferably be copied across to the output in some way.

An obvious starting point is to enhance the grammar with actions that simply write output as terminals are parsed. An example will make this clearer

```
CompoundStatement =
  "BEGIN"               (. Append("BEGIN"); IndentNewLine(); .)
     Statement
       { ";"            (. Append(";"); NewLine(); .)
         Statement }
  "END"                 (. ExdentNewLine(); Append("END"); .)   .
```

Of course, the productions for all the variations on *Statement* append their appropriate text as they are unravelled.

Once again, an external module might conveniently be introduced to give the support needed for these semantic actions, perhaps with an interface on the lines of

```
void Append(char *String);
// Append String to output
```

```
void IndentNewLine(void);
// Write line mark to output, and then prepare to indent further
// lines by a fixed amount more than before

void ExdentNewLine(void);
// Write line mark to output, and then prepare to indent further
// lines by a fixed amount less than before

void NewLine(void);
// Write line mark to output, but leave indentation as before

void Indent(void);
// Increment indentation level

void Exdent(void);
// Decrement indentation level

void SetIndentationStep(int Step);
// Set indentation step size to Step
```

13.21 If two high level languages are very similar, a translator from one to the other can often be developed by taking the idea of a pretty printer one stage further - rather than writing the same terminals as it reads, it writes slightly different ones. For example, a Clang *CompoundStatement* would be translated to the equivalent Topsy version by attributing the production as follows:

```
CompoundStatement =
  "BEGIN"               (. Append("{"); IndentNewLine(); .)
     Statement
       { ";"            (. NewLine(); .)
         Statement }
  "END"                 (. ExdentNewLine(); Append("}"); .)   .
```

Develop a complete Clang - Topsy translator in this way.

13.22 The Computer Centre has decided to introduce a system of charging users for electronic mail messages. The scale of charges will be as follows:

- Message charge: 20 units plus a charge per word of message text: 10 units for each word with at most 8 characters, 60 units for each word with more than 8 characters.
- The total charge is applied to every copy of the message transmitted - if a message is addressed to *N* multiple users, the sender's account is debited by *N* * *Charge*.

The program will be required to process data files exemplified by the following (read the messages - they give you some hints):

```
From: cspt@cs.ru.ac.za
To:   reader@in.bed, guru@sys-admin.uni-rhodes.ac.za
CC:   cslect@cs, pdterry@psg.com
This is a message containing twenty-seven words
The charge will be 20 plus 24 times 10 plus 3 times 60 units -
total 440 multiplied by 4
####
From: tutor@cs
To:   students@lab.somewhere
You should note that messages contain only words composed of plain
text or numbers or possible - signs

Assume for this project that no punctuation marks or other extra
characters will ever appear - this will make it much easier to do

User names and addresses may also contain digits and - characters
####
```

Each message has mandatory "`From`" and "`To`" lines, and an optional "`CC`" (carbon copy) line. Users are addressed in the usual Internet form, and case is insignificant. Ends of lines are, however, significant in addressing, and hence an EOL token must be catered for.

The chargeable text of a message starts after the `To` or `CC` line, and is terminated by the

(non-chargeable) #### line.

Describe this input by means of a suitable grammar, and then enhance it to provide the necessary attributes and actions to construct a complete charging system that will read and process a file of messages and then list the charges. In doing so you might like to consider developing a support module with an interface on the lines of that suggested below, and you should take care to incorporate error recovery.

```
void ChargeUser(char *Sender; int Charge);
// Pre:  Sender contains unique user name extracted from a From line
//       For example  cspt  extracted from  From: cspt@somewhere.com
//       Charge contains the charge for sending all copies of message
// Post: Database of charges updated to debit Charge to Sender

void ShowCharges(FILE *F);
// Pre:  Opened(F) AND the internal data base contains a list of user
//       names and accrued charges
// Post: The list has been displayed on file F
```

13.23 (This project requires some familiarity with music). "Tonic Solfa" is a notation sometimes used to help learn to play an instrument, or more frequently to sing, without requiring the use of expensive music printed in "staff notation". Many readers may have come across this as it applies to representing pitch. The notes of a major scale are named *doh, ray, me, fah, soh, lah, te* (and, as Julie Andrews taught us in *The Sound of Music*, that brings us back to *doh*). In the written notation these syllables are indicated by their initial letters only: *d r m f s l t*. Sharpened notes are indicated by adding the letter *e*, and flattened notes by adding the letter *a* (so that if the major scale were C major, *fe* would indicate F sharp and *la* would indicate A flat). Notes in octaves above the "starting" *doh* are indicated by superscript numbers, and notes below the "starting" *doh* are indicated by subscripts. Although the system is basically designed to indicate relative pitch, specific keys can be named at the beginning of the piece.

If, for the moment, we ignore timing information, the notes of the well-known jingle "Happy Birthday To You" could be represented by

$s_1\ s_1\ l_1\ s_1\ d\ t_1\ s_1\ s_1\ l_1\ s_1\ r\ d$
$s_1\ s_1\ s\ m\ d\ t_1\ l_1\ f\ f\ m\ d\ r\ d$

Of course we cannot really ignore timing information, which, unfortunately, complicates the picture considerably. In this notation, bar lines | and double bar lines || appear much as in staff notation. Braces { and } are used at the beginning and end of every line (except where a double bar line occurs).

The notation indicates relative note lengths, according to the basic pulse of the music. A bar line is placed before a strong pulse, a colon is placed before a weak pulse, and a shorter vertical line | indicates the secondary accent at the half bar in quadruple time. Horizontal lines indicate notes lasting longer than one beat (including dotted or tied notes). Pulses are divided in half by using dots as separators, and half pulses are further divided into quarter pulses by commas. Rests are indicated simply by leaving spaces. For example

| d : d | indicates duple time with notes on each pulse (two crotchets, if it were 2/4 time)

| d : - | d : d | indicates quadruple time (minim followed by two crotchets, in 4/4 time)

| d : - . d : | indicates triple time (dotted crotchet, quaver, crotchet rest, in 3/4 time)

| d : d . d : d,d . d | indicates triple time (crotchet, two quavers, two semiquavers, quaver, in 3/4
time)

"Happy Birthday To You" might then be coded fully as

{| : : s_1 . - , s_1 | l_1 : s_1 : d | t_1 : - : s_1 . - , s_1 }
{| l_1 : s_1 : r | d : - : s_1 . - , s_1 | s : m : d }
{| t_1 : l_1 : f . - , f | m : d : r | d : - : ||

Clearly this is fairly complex, and one suspects that singers may learn the rhythms "by ear" rather than by decoding this as they sing!

Write a Cocol grammar that describes this notation. Then go on to use it to develop a program that can read in a tune expressed this way and produce "machine code" for a device that is driven by a long stream of pairs of numbers, the first indicating the frequency of the note in Hz, and the second the duration of that note in milliseconds.

Recognizing superscripts and subscripts is clearly awkward, and it is suggested that you might initially use *d0 r0 m0 f0 s0 l0 t0 d r m f s l t d1 r1 m1 f1 s1 l1 t1* to give a range of three octaves, which will suffice for most songs.

Initially you might like to assume that a time signature (like the key) will preface the piece (which will simplify the computation of the duration of each note), and that the timing information has been correctly transcribed. As a later extension you might like to consider how varying time signatures and errors in transcription could be handled, while still assuming that each bar takes the same time to play.