12 USING COCO/R - OVERVIEW

One of the main reasons for developing attributed grammars like those discussed in the last chapter is to be able to use them as input to compiler generator tools, and so construct complete programs. It is the aim of this chapter and the next to illustrate how this process is achieved with Coco/R, and to discuss the Cocol specification language in greater detail than before. Our discussion will, as usual, focus mainly on C++ applications, but a study of the documentation and examples on the diskette should allow Modula-2, Pascal and "traditional C" readers to develop in those languages just as easily.

12.1 Installing and running Coco/R

On the diskette that accompanies this book can be found three implementations of Coco/R that can generate applications in C/C++, Modula-2, or Turbo Pascal. These have been configured for easy use on MS-DOS based systems. Versions of Coco/R are also available for use with many other compilers and operating systems. These can be obtained from several sites on the Internet; a list of some of these appears in Appendix A.

The installation and execution of Coco/R is rather system-specific, and readers will be obliged to make use of the documentation that is provided on the diskette. Nevertheless, a brief overview of the process can usefully be given here.

12.1.1 Installation

The MS-DOS versions of Coco/R are supplied as compressed, self-extracting executable files, and for these the installation process requires a user to

- create a system directory to store the system files [`MKDIR C:\COCO`];
- make this the active directory [`CD C:\COCO`];
- copy the distribution file to the system directory [`COPY A:COCORC.EXE C:\COCO`];
- start the decompression process [`COCORC`] (this process will extract the files, and create further subdirectories to contain Coco/R and its support files and library modules);
- add the system directory to the MS-DOS "path" (this may often most easily be done by modifying the `PATH` statement in the `AUTOEXEC.BAT` file);
- compile the library support modules;
- modify the host compiler and linker parameters, so that applications created by Coco/R can easily be linked to the support modules;
- set an "environment variable", so that Coco/R can locate its "frame files" (this may often most easily be done by adding a line like `SET CRFRAMES = C:\COCO\FRAMES` to the `AUTOEXEC.BAT` file).

12.1.2 Input file preparation

For each application, the user has to prepare a text file to contain the attributed grammar. Points to be aware of are that

- it is sensible to work within a "project directory" (say `C:\WORK`) and not within the "system directory" (`C:\COCO`);
- text file preparation must be done with an ASCII editor, and not with a word processor;
- by convention the file is named with a primary name that is based on the grammar's goal symbol, and with an "`ATG`" extension, for example `CALC.ATG`.

Besides the grammar, Coco/R needs to be able to read **frame files**. These contain outlines of the scanner, parser, and driver files, to which will be added statements derived from an analysis of the attributed grammar. Frame files for the scanner and parser are of a highly standard form; the ones supplied with the distribution are suitable for use in many applications without the need for any customization. However, a complete compiler consists of more than just a scanner and parser - in particular it requires a driver program to call the parser. A basic driver frame file (`COMPILER.FRM`) comes with the kit. This will allow simple applications to be generated immediately, but it is usually necessary to copy this basic file to the project directory, and then to edit it to suit the application. The resulting file should be given the same primary name as the grammar file, and a `FRM` extension, for example `CALC.FRM`.

12.1.3 Execution

Once the input files have been prepared, generation of the application is started with a command like

```
COCOR CALC.ATG
```

A number of compiler options may be specified in a way that is probably familiar, for example

```
COCOR -L -C CALC.ATG
```

The options depend on the particular version of Coco/R in use. A summary of those available may be obtained by issuing the `COCOR` command with no parameters at all, or with only a `-H` parameter. Compiler options may also be selected by **pragmas** embedded in the attributed grammar itself, and this is probably the preferred approach for serious applications. Examples of such pragmas can be found in the case studies later in this chapter.

12.1.4 Output from Coco/R

Assuming that the attributed grammar appears to be satisfactory, and depending on the compiler switches specified, execution of Coco/R will typically result in the production of header and implementation files (with names derived from the goal symbol name) for

- a FSA scanner (for example `CALCS.HPP` and `CALCS.CPP`)
- a recursive descent parser (for example `CALCP.HPP` and `CALCP.CPP`)
- a driver routine (for example `CALC.CPP`)
- a list of error messages (for example `CALCE.H`)
- a file relating the names of tokens to the integer numbers by which they will be known to the parser (for example `CALCC.H`)

12.1.5 Assembling the generated system

After they have been generated, the various parts of an application can be compiled and linked with one another, and with any other components that they need. The way in which this is done depends very much on the host compiler. For a very simple MS-DOS application using the Borland C++ system, one might be able to use commands like

```
    BCC -ml -IC:\COCO\CPLUS2 -c CALC.CPP CALCS.CPP CALCP.CPP
    BCC -ml -LC:\COCO\CPLUS2 -eCALC.EXE CALC.OBJ CALCS.OBJ CALCP.OBJ CR_LIB.LIB
```

but for larger applications the use of a **makefile** is probably to be preferred. Examples of makefiles are found on the distribution diskette.

12.2 Case study - a simple adding machine

Preparation of completely attributed grammars suitable as input to Coco/R requires an in-depth understanding of the Cocol specification language, including many features that we have not yet encountered. Sections 12.3 and 12.4 discuss these aspects in some detail, and owe much to the original description by Mössenböck (1990a).

The discussion will be clarified by reference to a simple example, chosen to illustrate as many features as possible (as a result, it may appear rather contrived). Suppose we wish to construct an adding machine that can add numbers arranged in various groups into subtotals, and then either add these subtotals to a running grand total, or reject them. Our numbers can have fractional parts; just to be perverse we shall allow a shorthand notation for handling ranges of numbers. Typical input is exemplified by

```
   clear                        // start the machine
   10 + 20 + 3 .. 7 accept      // one subtotal 10+20+3+4+5+6+7, accepted
   3.4 + 6.875..50 cancel       // another one, but rejected
   3 + 4 + 6 accept             // and a third, this time accepted
   total                        // display grand total and then stop
```

Correct input of this form can be described by a simple LL(1) grammar that we might try initially to specify in Cocol on the lines of the following:

```
   COMPILER Calc

     CHARACTERS
       digit    = "0123456789" .

     TOKENS
       number   = digit { digit } [ "." digit { digit } ] .

     PRODUCTIONS
       Calc     = "clear" { Subtotal } "total" .
       Subtotal = Range { "+" Range } ( "accept" | "cancel" ) .
       Range    = Amount [ ".." Amount ] .
       Amount   = number .

   END Calc.
```

In general a grammar like this can itself be described in EBNF by

```
   Cocol =   "COMPILER" GoalIdentifier
                 ArbitraryText
                 ScannerSpecification
                 ParserSpecification
             "END" GoalIdentifier "." .
```

We note immediately that the identifier after the keyword `COMPILER` gives the grammar name, and must match the name after the keyword `END`. The grammar name must also match the name chosen for the non-terminal that defines the goal symbol of the phrase structure grammar.

Each of the productions leads to the generation of a corresponding parsing routine. It should not take much imagination to see that the routines in our case study will also need to perform operations like

- converting the string that defines a `number` token into a corresponding numerical value. Thus we need mechanisms for extracting attributes of the various tokens from the scanner that recognizes them.
- adding such numbers into variables declared for the purpose of recording totals and subtotals, and passing these values between the routines. Thus we need mechanisms for declaring parameters and local variables in the generated routines, and for incorporating arithmetic statements.
- displaying the values of the variables on an output device. Thus we need mechanisms for interfacing our parsing routines to external library routines.
- reacting sensibly to input data that does not conform to the proper syntax. Thus we need mechanisms for specifying how error recovery should be accomplished.
- reacting sensibly to data that is syntactically correct, but still meaningless, as might happen if one was asked to process numbers in the range 6 .. 2. Thus we need mechanisms for reporting semantic and constraint violations.

These mechanisms are all incorporated into the grammar by attributing it with extra information, as discussed in the next sections. As an immediate example of this, arbitrary text may follow the *GoalIdentifier,* preceding the *ScannerSpecification.* This is not checked by Coco/R, but is simply incorporated directly in the generated parser. This offers the facility of providing code for `IMPORT` clauses in Modula-2, `USES` clauses in Turbo Pascal, or `#include` directives in C++, and for the declaration of global objects (constants, types, variables or functions) that may be needed by later semantic actions.

12.3 Scanner specification

A scanner has to read source text, skip meaningless characters, and recognize tokens that can be handled by the parser. Clearly there has to be some way for the parser to retrieve information about these tokens. The most fundamental information can be returned in the form of a simple integer, unique to the type of token recognized. While a moment's thought will confirm that the members of such an enumeration will allow a parser to perform syntactic analysis, semantic properties (such as the numeric value of the `number` that appears in our example grammar) may require a token to be analysed in more detail. To this end, the generated scanner allows the parser to retrieve the **lexeme** or textual representation of a token.

Tokens may be classified either as literals or as token classes. As we have already seen, literals (like "`END`" and "`!=`") may be introduced directly into productions as strings, and do not need to be named. Token classes (such as *identifiers* or *numbers*) must be named, and have structures that are specified by regular expressions, defined in EBNF.

In Cocol, a scanner specification consists of six optional parts, that may, in fact, be introduced in arbitrary order.

```
ScannerSpecification =  {  CharacterSets
                             | Ignorable
                             | Comments
                             | Tokens
                             | Pragmas
                             | UserNames
                           } .
```

12.3.1 Character sets

The *CharacterSets* component allows for the declaration of names for character sets like letters or digits, and defines the characters that may occur as members of these sets. These names may then be used in the other sections of the scanner specification (but not, it should be noted, in the parser specification).

```
CharacterSets   =  "CHARACTERS" { NamedCharSet } .
NamedCharSet    =  SetIdent "=" CharacterSet "." .
CharacterSet    =  SimpleSet { ( "+" | "-" ) SimpleSet } .
SimpleSet       =  SetIdent  |  string | SingleChar [ ".." SingleChar ] | "ANY" .
SingleChar      =  "CHR" "(" number ")" .
SetIdent        =  identifier .
```

Simple character sets are denoted by one of

SetIdent	a previously declared character set with that name
String	a set consisting of all characters in the string
`CHR(`*i*`)`	a set of one character with ordinal value *i*
`CHR(`*i*`) .. CHR(`*j*`)`	a set consisting of all characters whose ordinal values are in the range *i* ... *j*.
`ANY`	the set of all characters acceptable to the implementation

Simple sets may then be combined by the union (+) and difference operators (-).

As examples we might have

```
digit     = "0123456789" .      /* The set of all digits */
hexdigit  = digit + "ABCDEF" .  /* The set of all hexadecimal digits */
eol       = CHR(10) .           /* Line feed character */
noDigit   = ANY - digit .       /* Any character that is not a digit */
ctrlChars = CHR(1) .. CHR(31) . /* The ASCII control characters */
InString  = ANY - '"' - eol .   /* Strings may not cross line boundaries */
```

12.3.2 Comments and ignorable characters

Usually spaces within the source text of a program are irrelevant, and in scanning for the start of a token, a Coco/R generated scanner will simply ignore them. Other separators like tabs, line ends, and form feeds may also be declared irrelevant, and some applications may prefer to ignore the distinction between upper and lower case input.

Comments are difficult to specify with the regular expressions used to denote tokens - indeed, nested comments may not be specified at all in this way. Since comments are usually discarded by a parsing process, and may typically appear in arbitrary places in source code, it makes sense to have a special construct to express their structure.

Ignorable aspects of the scanning process are defined in Cocol by

```
Comments  = "COMMENTS" "FROM" TokenExpr "TO" TokenExpr [ "NESTED" ] .
Ignorable = "IGNORE" ( "CASE" | CharacterSet ) .
```

where the optional keyword `NESTED` should have an obvious meaning. A practical restriction is that comment brackets must not be longer than 2 characters. It is possible to declare several kinds of comments within a single grammar, for example, for C++:

```
COMMENTS FROM "/*" TO "*/"
COMMENTS FROM "//" TO eol
IGNORE CHR(9) .. CHR(13)
```

The set of ignorable characters in this example is that which includes the standard white space separators in ASCII files. The null character `CHR(0)` should not be included in any ignorable set. It is used internally by Coco/R to mark the end of the input file.

12.3.3 Tokens

A very important part of the scanner specification declares the form of terminal tokens:

```
Tokens       =  "TOKENS" { Token } .
Token        =  TokenSymbol [ "=" TokenExpr "." ] .
TokenExpr    =  TokenTerm { "|" TokenTerm } .
TokenTerm    =  TokenFactor { TokenFactor } [ "CONTEXT" "(" TokenExpr ")" ] .
TokenFactor  =  SetIdent | string
                  | "(" TokenExpr ")"
                  | "[" TokenExpr "]"
                  | "{" TokenExpr "}" .
TokenSymbol  =  TokenIdent | string .
TokenIdent   =  identifier .
```

Tokens may be declared in any order. A token declaration defines a *TokenSymbol* together with its structure. Usually the symbol on the left-hand side of the declaration is an identifier, which is then used in other parts of the grammar to denote the structure described on the right-hand side of the declaration by a regular expression (expressed in EBNF). This expression may contain literals denoting themselves (for example "`END`"), or the names of character sets (for example *letter*), denoting an arbitrary character from such sets. The restriction to regular expressions means that it may not contain the names of any other tokens.

While token specification is usually straightforward, there are a number of subtleties that may need emphasizing:

- Since spaces are deemed to be irrelevant when they come *between* tokens in the input for most languages, one should not attempt to declare literal tokens that have spaces *within* them.

- Our case study has introduced but one explicit token class:

  ```
  number = digit { digit } [ "." digit { digit } ] .
  ```

  However it has also introduced tokens like "`clear`", "`cancel`" and "..". This last one is particularly interesting. A scanner might have trouble distinguishing the tokens in input like

  ```
  3 .. 5.4  +  5.4..16.4  + 50..80
  ```

  because in some cases the periods form part of a real literal, in others they form part of an ellipsis. This sort of situation arises quite frequently, and Cocol makes special provision for it. An optional `CONTEXT` phrase in a *TokenTerm* specifies that this term only be recognized when its right-hand context in the input stream is the *TokenExpr* specified in brackets. Our case study example requires alteration:

  ```
  TOKENS
    number =   digit { digit } [ "." digit { digit } ]
             | digit { digit } CONTEXT ( ".." ) .
  ```

- The grammar for tokens allows for empty right-hand sides. This may seem strange, especially as no scanner is generated if the right-hand side of a declaration is missing. This facility is used if the user wishes to supply a hand-crafted scanner, rather than the one generated by Coco/R. In this case, the symbol on the left- hand side of a token declaration may also simply be specified by a *string*, with no right-hand side.

- Tokens specified without right-hand sides are numbered consecutively starting from 0, and the hand-crafted scanner has to return token codes according to this numbering scheme.

12.3.4 Pragmas

A pragma, like a comment, is a token that may occur anywhere in the input stream, but, unlike a comment, it cannot be ignored. Pragmas are often used to allow programmers to select compiler switches dynamically. Since it becomes impractical to modify the phrase structure grammar to handle this, a special mechanism is provided for the recognition and treatment of pragmas. In Cocol they are declared like tokens, but may have an associated semantic action that is executed whenever they are recognized by the scanner.

```
Pragmas     =  "PRAGMAS" { Pragma } .
Pragma      =  Token [ Action ] .
Action      =  "(." arbitraryText ".)" .
```

As an example, we might add to our case study

```
PRAGMAS
  page = "page" .  (. printf("\f"); .)
```

to allow the word `page` to appear anywhere in the input data; each appearance would have the effect of moving to a new page on the output.

12.3.5 User names

The scanner and parser produced by Coco/R use small integer values to distinguish tokens. This makes their code harder to understand by a human reader (some would argue that humans should never need to read such code anyway). When used with appropriate options, Coco/R can generate code that uses names for the tokens. By default these names have a rather stereotyped form (for example "..." would be named "`pointpointpointSym`"). The *UserNames* section may be used to prefer user-defined names, or to help resolve name clashes (for example, between the default names that would be chosen for "`point`" and ".").

```
UserNames  = "NAMES" { UserName } .
UserName   = TokenIdent  "=" ( identifier | string ) "." .
```

As examples we might have

```
NAMES
  period   = "." .
  ellipsis = "..." .
```

12.3.6 The scanner interface

The scanner generated by Coco/R declares various procedures and functions that may be called from the parser whenever it needs to obtain a new token, or to analyse one that has already been recognized. As it happens, a user rarely has to make direct use of this interface, as the generated parser incorporates all the necessary calls to the scanner routines automatically, and also provides facilities for retrieving lexemes.

The form of the interface depends on the host system. For example, for the C++ version, the interface is effectively that shown below, although there is actually an underlying class hierarchy, so that the declarations are not exactly the same as those shown. The reader should take note that there are various ways in which source text may be retrieved from the scanner (to understand these in full it will be necessary to study the class hierarchy, but easier interfaces are provided for the parser; see section 12.4.6).

```
class grammarScanner
{ public:
    grammarScanner(int SourceFile, int ignoreCase);
    // Constructs scanner for grammar and associates this with a
    // previously opened SourceFile.  Specifies whether to IGNORE CASE

    int Get();
    // Retrieves next token from source

    void GetString(Token *Sym, char *Buffer, int Max);
    // Retrieves at most Max characters from Sym into Buffer

    void GetName(Token *Sym, char *Buffer, int Max);
    // Retrieves at most Max characters from Sym into Buffer
    // Buffer is capitalized if IGNORE CASE was specified

    long GetLine(long Pos, char *Line, int Max);
    // Retrieves at most Max characters (or until next line break)
    // from position Pos in source file into Line

};
```

12.4 Parser specification

The parser specification is the main part of the input to Coco/R. It contains the productions of an attributed grammar specifying the syntax of the language to be recognized, as well as the action to be taken as each phrase or token is recognized.

12.4.1 Productions

The form of the parser specification may itself be described in EBNF as follows. For the Modula-2 and Pascal versions we have:

```
ParserSpecification =  "PRODUCTIONS" { Production } .
Production          =  NonTerminal [ FormalAttributes ]
                         [ LocalDeclarations ]        (* Modula-2 and Pascal *)
                         "=" Expression "." .
FormalAttributes    =  "<"  arbitraryText ">" | "<."  arbitraryText ".>" .
LocalDeclarations   =  "(." arbitraryText ".)" .
NonTerminal         =  identifier .
```

For the C and C++ versions the *LocalDeclarations* follow the "=" instead:

```
Production          =  NonTerminal [ FormalAttributes ]
                         "=" [ LocalDeclarations ]  /* C and C++ */
                         Expression "." .
```

Any identifier appearing in a production that was not previously declared as a terminal token is considered to be the name of a *NonTerminal*, and there must be exactly one production for each *NonTerminal* that is used in the specification (this may, of course, specify a list of alternative right sides).

A production may be considered as a specification for creating a routine that parses the *NonTerminal*. This routine will constitute its own scope for parameters and other local components like variables and constants. The left-hand side of a *Production* specifies the name of the *NonTerminal* as well as its *FormalAttributes* (which effectively specify the formal parameters of the routine). In the Modula-2 and Pascal versions the optional *LocalDeclarations* allow the declaration of local components to precede the block of statements that follow. The C and C++ versions define their local components within this statement block, as required by the host language.

As in the case of tokens, some subtleties in the specification of productions should be emphasized:

- The productions may be given in any order.
- A production must be given for a *GoalIdentifier* that matches the name used for the grammar.
- The formal attributes enclosed in angle brackets "<" and ">" (or "`<.`" and "`.>`") simply consist of parameter declarations in the host language. Similarly, where they are required and permitted, local declarations take the form of host language declarations enclosed in "`(.`" and "`.)`" brackets. However, the syntax of these components is not checked by Coco/R; this is left to the responsibility of the compiler that will actually compile the generated application.
- All routines give rise to "regular procedures" (in Modula-2 terminology) or "void functions" (in C++ terminology). Coco/R cannot construct true functions that can be called from within other expressions; any return values must be transmitted using reference parameter mechanisms.
- The goal symbol may not have any *FormalAttributes*. Any information that the parser is required to pass back to the calling driver program must be handled in other ways. At times this may prove slightly awkward.
- While a production constitutes a scope for its formal attributes and its locally declared objects, terminals and non-terminals, globally declared objects, and imported modules are visible in any production.
- It may happen that an identifier chosen as the name of a *NonTerminal* may clash with one of the internal names used in the rest of the system. Such clashes will only become apparent when the application is compiled and linked, and may require the user to redefine the grammar to use other identifiers.

The *Expression* on the right-hand-side of each *Production* defines the context-free structure of some part of the source language, together with the attributes and semantic actions that specify how the parser must react to the recognition of each component. The syntax of an *Expression* may itself be described in EBNF (albeit not in LL(1) form) as

```
Expression   =  Term { "|" Term } .
Term         =  Factor { Factor }  .
Factor       =     [ "WEAK" ] TokenSymbol
                |  NonTerminal [ Attributes ]
                |  Action
                |  "ANY"
                |  "SYNC"
                |  "(" Expression ")"
                |  "[" Expression "]"
                |  "{" Expression "}" .
Attributes   =  "<"  arbitraryText ">" |  "<."  arbitraryText ".>" .
Action       =  "(." arbitraryText ".)" .
```

The *Attributes* enclosed in angle brackets that may follow a *NonTerminal* effectively denote the actual parameters that will be used in calling the corresponding routine. If a *NonTerminal* is defined on the left-hand side of a *Production* to have *FormalAttributes*, then every occurrence of that *NonTerminal* in a right-hand side *Expression* must have a list of actual attributes that correspond to the *FormalAttributes* according to the parameter compatibility rules of the host language. However, the conformance is only checked when the generated parser is itself compiled.

An *Action* is an arbitrary sequence of host language statements enclosed in "`(.`" and "`.)`". These are simply incorporated into the generated parser *in situ*; once again, no syntax is checked at that stage.

These points may be made clearer by considering a development of part of our case study, which hopefully needs little further explanation:

```
PRODUCTIONS
  Calc                                                        /* goal */
  =                 (. double total = 0.0, sub; .)            /* locals */
    "clear"
    { Subtotal<sub> (. total += sub; .)                       /* add to total */
    }
    "total"         (. printf("   total: %5.2f\n", total); .) /* display */
    .

  Subtotal<double &s>                                         /* ref param */
  =                 (. double r; .)                           /* local */
    Range<s>
    { "+" Range<r>  (. s += r; .)                             /* add to s */
    }
    (   "accept"    (. printf("subtotal: %5.2f\n", s); .)     /* display */
      | "cancel"    (. s = 0.0; .)                            /* nullify */
    ) .
```

Although the input to Coco/R is free-format, it is suggested that the regular EBNF appear on the left, with the actions on the right, as in the example above.

Many aspects of parser specification are straightforward, but there are some subtleties that call for comment:

- Where it appears, the keyword `ANY` denotes any terminal that cannot follow `ANY` in that context. It can conveniently be used to parse structures that contain arbitrary text.

- The `WEAK` and `SYNC` keywords are used in error recovery, as discussed in the next section.

- In earlier versions of Coco/R there was a potential pitfall in the specification of attributes. Suppose the urge arises to attribute a *NonTerminal* as follows:

  ```
  SomeNonTerminal< record->field >
  ```

  where the parameter uses the right arrow selection operator "`->`". Since the "`>`" would normally have been taken as a Cocol meta-bracket, this had to be recoded in terms of other operators as

  ```
  SomeNonTerminal< (*record).field >
  ```

  The current versions of Coco/R allow for attributes to be demarcated by "`<.`" and "`.>`" brackets to allow for this situation, and for other operators that involve the > character.

- Close perusal of the grammar for *Expression* will reveal that it is legal to write a *Production* in which an *Action* appears to be associated with an alternative for an *Expression* that contains no terminals or non- terminals at all. This feature is often useful. For example we might have

  ```
  Option =   "push" (. stack[++top] = item; .)
           | "pop"  (. item = stack[top--]; .)
           |        (. for (int i = top; i > 0; i--) cout << stack[i]; .) .
  ```

- Another useful feature that can be exploited is the ability of an *Action* to drive the parsing process "semantically". For example, the specification of assignment statements and procedure calls in a simple language might be defined as follows so as to conform to LL(1) restrictions

```
AssignmentOrCall = Identifier [ ":=" Expression ] .
```

Clearly the semantics of the two statement forms are very different. To handle this we might write the grammar on the lines of

```
AssignmentOrCall
= Identifier<name>          (. Lookup(name);
                               if (IsProcedure(name))
                                 { HandleCall(name); return; } .)
  ":=" Expression<value>    (. HandleAssignment(name, value); .) .
```

12.4.2 Syntax error recovery

Compiler generators vary tremendously in the way in which they provide for recovery from syntactic errors, a subject that was discussed in section 10.3.

The technique described there, although systematically applicable, slows down error-free parsing, inflates the parser code, and is relatively difficult to automate. Coco/R uses a simpler technique, as suggested by Wirth (1986), since this has proved to be almost as effective, and is very easily understood. Recovery takes place only at a rather small number of **synchronization points** in the grammar. Errors at other points are reported, but cause no recovery - parsing simply continues up to the next synchronization point. One consequence of this simplification is that many spurious errors are then likely to be detected for as long as the parser and the input remain out of step. An effective technique for handling this is to arrange that errors are simply not reported if they follow too closely upon one another (that is, a minimum amount of text must be correctly parsed after one error is detected before the next can be reported).

In the simplest approach to using this technique, the designer of the grammar is required to specify synchronization points explicitly. As it happens, this does not usually turn out to be a difficult task: the usual heuristic is to choose locations in the grammar where especially safe terminals are expected that are hardly ever missing or mistyped, or appear so often in source code that they are bound to be encountered again at some stage. In most Pascal-like languages, for example, good candidates for synchronization points are the beginning of a statement (where keywords like `IF` and `WHILE` are expected), the beginning of a declaration sequence (where keywords like `CONST` and `VAR` are expected), or the beginning of a type definition (where keywords like `RECORD` and `ARRAY` are expected).

In Cocol, a synchronization point is specified by the keyword `SYNC`, and the effect is to generate code for a loop that is prepared simply to consume source tokens until one is found that would be acceptable at that point. The sets of such terminals can be precomputed at parser generation time. They are always extended to include the end-of-file symbol (denoted by the keyword `EOF`), thus guaranteeing that if all else fails, synchronization will succeed at the end of the source text.

For our case study we might choose the end of the routine for handling a subtotal as such a point:

```
Subtotal = Range { "+" Range } SYNC ( "accept" | "cancel" ) .
```

This would have the effect of generating code on the following lines:

```
PROCEDURE Subtotal;
  BEGIN
    Range;
    WHILE Sym = plus DO GetSym; Range END;
    WHILE Sym ∉ { accept, cancel, EOF } DO GetSym END;
    IF Sym ∈ { accept, cancel } THEN GetSym END;
    END
```

The union of all the synchronization sets (which we shall denote by *AllSyncs*) is also computed by Coco/R, and is used in further refinements on this idea. A terminal can be designated to be *weak* in a certain context by preceding its appearance in the phrase structure grammar with the keyword `WEAK`. A weak terminal is one that might often be mistyped or omitted, such as the semicolon between statements. When the parser expects (but does not find) such a terminal, it adopts the strategy of consuming source tokens until it recognizes either a legal successor of the weak terminal, or one of the members of *AllSyncs* - since terminals expected at synchronization points are considered to be very "strong", it makes sense that they never be skipped in any error recovery process.

As an example of how this could be used, consider altering our case study grammar to read:

```
  Calc     = WEAK "clear" Subtotal { Subtotal } WEAK "total" .
  Subtotal = Range { "+" Range } SYNC ( "accept" | "cancel" ) .
  Range    = Amount [ ".." Amount ] .
  Amount   = number .
```

This would give rise to code on the lines of

```
  PROCEDURE Calc;
    BEGIN
      ExpectWeak(clear, FIRST(Subtotal)); (* ie { number } *)
      Subtotal; WHILE Sym = number DO Subtotal END;
      ExpectWeak(total, { EOF })
    END
```

The `ExpectWeak` routine would be internal to the parser, implemented on the lines of:

```
  PROCEDURE ExpectWeak (Expected : TERMINAL; WeakFollowers : SYMSET);
    BEGIN
      IF Sym = Expected
        THEN GetSym
        ELSE
          ReportError(Expected);
          WHILE sym ∉ (WeakFollowers + AllSyncs) DO GetSym END
      END
    END
```

Weak terminals give the parser another chance to synchronize in case of an error. The `WeakFollower` sets can be precomputed at parser generation time, and the technique causes no run-time overhead if the input is error-free.

Frequently iterations start with a weak terminal, in situations described by EBNF of the form

```
    Sequence =  FirstPart { "WEAK" ExpectedTerminal  IteratedPart } LastPart .
```

Such terminals will be called *weak separators* and can be handled in a special way: if the *ExpectedTerminal* cannot be recognized, source tokens are consumed until a terminal is found that is contained in one of the following three sets:

FOLLOW(*ExpectedTerminal*) (that is, FIRST(*IteratedPart*))
FIRST(*LastPart*)
AllSyncs

As an example of this, suppose we were to modify our case study grammar to read

```
    Subtotal = Range { WEAK "+" Range } ( "accept" | "cancel" ) .
```

The generated code would then be on the lines of

```
PROCEDURE Subtotal;
  BEGIN
    Range;
    WHILE WeakSeparator(plus, { number }, { accept, cancel } ) DO
      Range
    END;
    IF Sym ∈ {accept, cancel } THEN GetSym END;
  END
```

The `WeakSeparator` routine would be implemented internally to the parser on the lines of

```
BOOLEAN FUNCTION WeakSeparator (Expected : TERMINAL;
                                WeakFollowers, IterationFollowers : SYMSET);
  BEGIN
    IF Sym = Expected THEN GetSym; RETURN TRUE
      ELSIF Sym ∈ IterationFollowers THEN RETURN FALSE
      ELSE
        ReportError(Expected);
        WHILE Sym ∉ (WeakFollowers + IterationFollowers + AllSyncs) DO
          GetSym
        END;
        RETURN Sym ∈ WeakFollowers
    END
  END
```

Once again, all the necessary sets can be precomputed at generation time. Occasionally, in highly embedded grammars, the inclusion of *AllSyncs* (which tends to be "large") may detract from the efficacy of the technique, but with careful choice of the placing of `WEAK` and `SYNC` keywords it can work remarkably well.

12.4.3 Grammar checks

Coco/R performs several tests to check that the grammar submitted to it is well-formed. In particular it checks that

- each non-terminal has been associated with exactly one production;
- there are no useless productions (in the sense discussed in section 8.3.1);
- the grammar is cycle free (in the sense discussed in section 8.3.3);
- all tokens can be distinguished from one another (that is, no two terminals have been declared to have the same structure).

If any of these tests fail, no code generation takes place. In other respects the system is more lenient. Coco/R issues warnings if analysis of the grammar reveals that

- a non-terminal is nullable (this occurs frequently in correct grammars, but may sometimes be indicative of an error);
- the LL(1) conditions are violated, either because at least two alternatives for a production have FIRST sets with elements in common, or because the FIRST and FOLLOWER sets for a nullable string have elements in common.

If Coco/R reports an LL(1) error for a construct that involves alternatives or iterations, the user should be aware that the generated parser is highly likely to misbehave. As simple examples, productions like the following

```
P = "a" A | "a" B .
Q = [ "c" B ] "c"  .
R = { "d" C } "d" .
```

result in generation of code that can be described algorithmically as

```
IF Sym = "a" THEN Accept("a"); A ELSIF Sym = "a" THEN Accept("a"); B END;
```

```
      IF Sym = "c" THEN Accept("c"); B END; Accept("c");
      WHILE Sym = "d" DO Accept("d"); C END; Accept("d");
```

Of these, only the second can possibly ever have any meaning (as it does in the case of the "dangling else"). If these situations arise it may often be necessary to redesign the grammar.

12.4.4 Semantic errors

The parsers generated by Coco/R handle the reporting of syntax errors automatically. The default driver programs can summarize these errors at the end of compilation, along with source code line and column references, or produce source code listings with the errors clearly marked with explanatory messages (an example of such a listing appears in section 12.4.7). Pure syntax analysis cannot reveal static semantic errors, but Coco/R supports a mechanism whereby the grammar designer can arrange for such errors to be reported in the same style as is used for syntactic errors. The parser class has routines that can be called from within the semantic actions, with an error number parameter that can be associated with a matching user-defined message.

In the grammar of our case study, for example, it might make sense to introduce a semantic check into the actions for the non-terminal `Range`. The grammar allows for a range of values to be summed; clearly this will be awkward if the "upper" limit is supplied as a lower value than the "lower" limit. The code below shows how this could be detected, resulting in the reporting of the semantic error 200.

```
    Range<double &r>
    =                       (. double low, high; .)
      Amount<low>           (. r = low; .)
      [ ".." Amount<high>   (. if (low > high) SemError(200);
                               else while (low < high) { low++; r += low; } .)
      ] .
```

(Alternatively, we could also arrange for the system to run the loop in the appropriate direction, and not regard this as an error at all.) Numbers chosen for semantic error reporting must start at some fairly large number to avoid conflict with the low numbers chosen internally by Coco/R to report syntax errors.

12.4.5 Interfacing to support modules

It will not have escaped the reader’s attention that the code specified in the actions of the attributed grammar will frequently need to make use of routines that are not defined by the grammar itself. Two typical situations are exemplified in our case study.

Firstly, it has seen fit to make use of the `printf` routine from the `stdio` library found in all standard C and C++ implementations. To make use of such routines - or ones defined in other support libraries that the application may need - it is necessary simply to incorporate the appropriate `#define, IMPORT` or `USES` clauses into the grammar before the scanner specification, as discussed in section 12.2.

Secondly, the need arises in routines like `Amount` to be able to convert a string, recognized by the scanner as a number, into a numerical value that can be passed back via a formal parameter to the calling routine (`Range`). This situation arises so frequently that the parser interface defines several routines to simplify the extraction of this string. The production for `Amount`, when fully attributed, might take the form

```
    Amount<double &a>
    = number                (. char str[100];
                               LexString(str, 100);
```

```
                                  a = atof(str); .) .
```

The `LexString` routine (defined in the parser interface) retrieves the string into the local string `str`, whence it is converted to the `double` value `a` by a call to the `atof` function that is defined in the `stdlib` library. If the functionality of routines like `LexString` and `LexName` is inadequate, the user can incorporate calls to the even lower level routines defined in the scanner interface, such as were mentioned in section 12.3.6.

12.4.6 The parser interface

The parser generated by Coco/R defines various routines that may be called from an application. As for the scanner, the form of the interface depends on the host system. For the C++ version, it effectively takes the form below. (As before, there is actually an underlying class hierarchy, and the declarations are really slightly different from those presented here).

The functionality provides for the parser to

- initiate the parse for the goal symbol by calling `Parse()`.
- investigate whether the parse succeeded by calling `Successful()`.
- report on the presence of syntactic and semantic errors by calling `SynError` and `SemError`.
- obtain the lexeme value of a particular token in one of four ways (`LexString`, `LexName`, `LookAheadString` and `LookAheadName`). Calls to `LexString` are most common; the others are used for special variations.

```
class grammarParser
{ public:
    grammarParser(AbsScanner *S, CRError *E);
    // Constructs parser associated with scanner S and error reporter E

    void Parse();
    // Parses the source

    int Successful();
    // Returns 1 if no errors have been recorded while parsing

  private:
    void LexString(char *lex, int size);
    // Retrieves at most size characters from the most recently parsed
    // token into lex

    void LexName(char *lex, int size);
    // Retrieves at most size characters from the most recently parsed
    // token into lex, converted to upper case if IGNORE CASE was specified

    void LookAheadString(char *lex, int size);
    // Retrieves at most size characters from the lookahead token into lex

    void LookAheadName(char *lex, int size);
    // Retrieves at most size characters from the lookahead token into lex,
    // converted to upper case if IGNORE CASE was specified

    void SynError(int errorcode);
    // Reports syntax error denoted by errorcode

    void SemError(int errorcode);
    // Reports semantic error denoted by errorcode

    // ... Prototypes of functions for parsing each non-terminal in grammar
};
```

12.4.7 A complete example

To place all of the ideas of the last sections in context, we present a complete version of the attributed grammar for our case study:

```
$CX     /* pragmas - generate compiler, and use C++ classes */
```

```
COMPILER Calc

  #include <stdio.h>
  #include <stdlib.h>

  CHARACTERS
    digit =  "0123456789" .

  IGNORE CHR(9) .. CHR(13)

  TOKENS
    number =   digit { digit } [ "." digit { digit } ]
             | digit { digit } CONTEXT ( ".." ) .

  PRAGMAS
    page   = "page" .         (. printf("\f"); .)

  PRODUCTIONS
    Calc
    =                         (. double total = 0.0, sub; .)
      WEAK "clear"
      { Subtotal<sub>         (. total += sub; .)
      } SYNC "total"          (. printf("   total: %5.2f\n", total); .)
      .

    Subtotal<double &s>
    =                         (. double r; .)
      Range<s>
      { WEAK "+" Range<r>     (. s += r; .)
      } SYNC
      (   "accept"            (. printf("subtotal: %5.2f\n", s); .)
        | "cancel"            (. s = 0.0; .)
      ) .

    Range<double &r>
    =                         (. double low, high; .)
      Amount<low>             (. r = low; .)
      [ ".." Amount<high>     (. if (low > high) SemError(200);
                                 else while (low < high)
                                 { low++; r += low; } .)
      ] .

    Amount<double &a>
    = number                  (. char str[100];
                                 LexString(str, 100);
                                 a = atof(str); .) .

END Calc.
```

To show how errors are reported, we show the output from applying the generated system to input that is fairly obviously incorrect.

```
    1  clr
***** ^  clear  expected (E2)
    2  1 + 2 + 3 .. 4 + 4..5 accep
*****                       ^  +  expected (E4)
    3  3.4 5 cancel
*****      ^  +  expected (E4)
    4  3 + 4 .. 2 + 6 accept
*****           ^ High < Low (E200)
    5  TOTAL
*****  ^ unexpected symbol in Calc (E10)
```

12.5 The driver program

The most important tasks that Coco/R has to perform are the construction of the scanner and parser. However, these always have to be incorporated into a complete program before they become useful.

12.5.1 Essentials of the driver program

Any main routine for a driver program must be a refinement of ideas that can be summarized:

```
BEGIN
  Open(SourceFile);
  IF Okay THEN
    InstantiateScanner;
    InstantiateErrorHandler;
    InstantiateParser;
    Parse();
    IF Successful() THEN ApplicationSpecificAction END
  END
END
```

Much of this can be automated, of course, and Coco/R can generate such a program, consistent with its other components. To do so requires the use of an appropriate frame file. A generic version of this is supplied with the distribution. Although it may be suitable for constructing simple prototypes, it acts best as a model from which an application-specific frame file can easily be derived.

12.5.2 Customizing the driver frame file

A customized driver frame file generally requires at least three simple additions:

- It is often necessary to declare global or external variables, and to add application specific `#include, USES` or `IMPORT` directives so that the necessary library support will be provided.

- The section dealing with error messages may need extension if the grammar has made use of the facility for adding errors to those derived by the parser generator, as discussed in section 12.4.4. For example, the default C++ driver frame file has code that reads

  ```
  char *MyError::GetUserErrorMsg(int n)
  { switch (n) {
      // Put your customized messages here
      default:  return "Unknown error";
    }
  }
  ```

  To tailor this to the case study application we should need to add an option to the `switch` statement:

  ```
  char *MyError::GetUserErrorMsg(int n)
  { switch (n) {
      case 200: return "High < Low";
      default:  return "Unknown error";
    }
  }
  ```

- Finally, at the end of the default frame file can be found code like

  ```
     // instantiate Scanner, Parser and Error handler
     Scanner = new -->ScanClass(S_src, -->IgnoreCase);
     Error   = new MyError(SourceName, Scanner);
     Parser  = new -->ParserClass(Scanner, Error);

     // parse the source
     Parser->Parse();
     close(S_src);

     // Add to the following code to suit the application
     if (Error->Errors) fprintf(stderr, "Compilation errors\n");
     if (Listinfo) SourceListing(Error, Scanner);
     else if (Error->Errors) Error->SummarizeErrors();

     delete Scanner;
     delete Parser;
     delete Error;
   }
  ```

  the intention of which should be almost self explanatory. For example, in the case of a compiler/interpreter such as we shall discuss in a later chapter, we might want to modify this to read

```
// generate source listing
FILE *lst = fopen("listing");
Error->SetOutput(lst);
Error->PrintListing(Scanner);
fclose(lst);

if (Error->Errors)
  fprintf(stderr, "Compilation failed - see %s\n", ListName);
else {
  fprintf(stderr, "Compilation successful\n");
  CGen->getsize(codelength, initsp);
  Machine->interpret(codelength, initsp);
}
```

Exercises

12.1 Study the code produced by Coco/R from the grammar used in this case study. How closely does it correspond to what you might have written by hand?

12.2 Experiment with the grammar suggested in the case study. What happens if the `CONTEXT` clause is omitted in the scanner specification? What happens if the placement of the `WEAK` and `SYNC` keywords is changed?

12.3 Extend the system in various ways. For example, direct output to a file other than `stdout`, use the `iostreams` library rather than the `stdio` library, develop the actions so that they conform to "traditional" C (rather than using reference parameters), or arrange that ranges can be correctly interpreted in either order.

Further reading

The text by Rechenberg and Mössenböck (1989) describes the original Coco system in great detail. This system did not have an integrated scanner generator, but made use of one known as Alex (Mössenböck, 1986). Dobler and Pirklbauer (1990) and Dobler (1991) discuss Coco-2, a variant of Coco that incorporated automatic and sophisticated error recovery into table-driven LL(1) parsers. Literature on the inner workings of Coco/R is harder to come by, but the reader is referred to the papers by Mössenböck (1990a, 1990b).