Differences between the Linz and extended versions of Coco/R

P.D. Terry, Computer Science Department, Rhodes University

June 2005

These notes relate to the versions of Coco/R for C# and for Java released in June 2005. These differ from the earlier releases in that they produce scanners and parsers that have to be instantiated - that is, where the fields and methods are not "static" as they were previously. The earlier releases can be found at the website `http://www.scifac.ru.ac.za/resourcekit.`

1 Introduction

Software for the Coco/R compiler generator is distributed in accordance with the GNU General Public License (GPL), as outlined in a separate document in the resource kit. To comply with the GPL, the full source code for each of the Java and C# implementations of Coco/R has been included in the resource kit in two variations. One of these contains the versions as released by Hanspeter Mössenböck and his colleagues at the Institute for System Software of the Johannes Kepler Universität in Linz, Austria. The other set contains the versions modified and extended by the author to incorporate features that have been found useful for the case studies in this book. This document describes the differences between these versions, and is conveniently read in conjunction with the description of Coco/R that can be found in separate documents in the resource kit.

2 Differences in the user interface

Both the extended and the Linz versions of Coco/R are invoked from a command line prompt with a command that in its simplest form is exemplified by

```
Coco Grammar.ATG
```

where `Grammar.ATG` is the name of an ASCII file containing the Cocol description of the parser and scanner to be generated. The primary part of the name (exemplified here by `Grammar`) should match the name chosen for the goal symbol; the `.ATG` extension is simply conventional. The file name may include path information:

```
Coco D:\project\parva\Parva.ATG
```

Besides specifying the source file name, the command may incorporate optional directives, as exemplified by

```
Coco Calc.ATG -namespace Calc -frames C:\resource\cococ\frames -trace fs
```

Possible directives are as follows:

- The directives

  ```
  -namespace namespaceName
  -package   packageName
  ```

  are used to indicate that the code for the system is to belong to the specified namespace (C# versions) or package (Java versions). In the absence of such directives, the Linz versions generate code for the default namespace (or package), but the extended versions generate code for a namespace (or package) that has the same name as the goal symbol `Grammar` specified in the `Grammar.ATG` file.

- Besides the file containing the attributed grammar (specified as a command line parameter), Coco/R requires so-called frame files, which contain skeleton code for the scanner and parser classes. To this end, the directive

  ```
  -frames  frameDirectoryPath
  ```

  specifies the path to a directory where Coco/R can expect to locate the frame files `Parser.frame` and `Scanner.frame` and (for the extended system) the default `Driver.frame`. In the absence of this directive the frame files are assumed to reside in the same directory as the `Grammar.ATG` file. To eliminate proliferation of these files it is recommended that they be stored in a standard directory, as exemplified above.

- The directive

```
-trace  optionString
```

specifies a set of tracing options that can be used to generate extra output for diagnostic or informative purposes. The options are specified by a string concatenated from single letters, exemplified by

```
-trace  fsx
```

where the effects of the permissible letters in the `optionString` can be summarized:

`a`	Trace the construction of the finite state automaton used in the scanner
`f`	List the FIRST and FOLLOW sets for each non-terminal in the grammar
`g`	Summarize the syntax graph constructed from the grammar
`i`	Trace the construction of the FIRST sets for each non-terminal in the grammar
`j`	List the ANY and SYNC sets used in error recovery
`p`	Give statistics of the numbers of terminals, non-terminals and graph nodes
`s`	List the symbol table
`x`	Generate a cross-reference table for the terminals and non-terminals

Trace output is directed to a file `trace.txt`, created in the same directory as the file `Grammar.ATG`.

- In the extended version the directives

```
-trace   optionString
-options optionString
```

are equivalent. Further letters may be introduced to the `optionString`:

`c`	Generate code for a compiler driver class (`Grammar.cs` or `Grammar.java`)
`m`	Merge any error messages with the source code to create a listing file
`n`	Generate source code that uses names for the tokens and terminals
`t`	Test the grammar, but do not generate any code

- With the exception of "`m`", these various features may also be selected by pragmas placed within the `Grammar.ATG` file (conventionally at the start). Pragmas take the form

```
$optionString
```

This may be exemplified by

```
COMPILER Grammar  $cnf
...
```

and this may be the preferred route to follow for many applications

Some further observations should be made in respect of the extended features:

- If a compiler driver class is to be generated, a driver frame file is required. For very simple applications the default `Driver.frame` file supplied in the distribution might suffice, but for serious applications a customized frame file should be created in the same directory as the `Grammar.ATG` file and given the name `Grammar.frame`. It is suggested that this file can be derived fairly simply from the simple `Driver.frame`, and some hints for doing so can be found in section 10.7.2 of the text. In its simplest form a complete driver might look like this:

```
public class Compiler {

  public static void Main(string[] arg) {
    Scanner scanner = new Scanner(arg[0]);
    Errors errors = new Errors(scanner, arg[0], "", false);
    Parser parser = new Parser(scanner, errors);
    parser.Parse();
    Console.WriteLine(errors.count + " errors detected");
  }
}
```

- Error messages in Coco/R and in the compilers it generates are directed by default to the standard output file as parsing proceeds. In the Linz version these messages are of a form exemplified by

  ```
  -- line 6 col 9: "begin" expected
  ```

  but in the extended version they are of a form exemplified by

  ```
  file Grammar.atg : (11, 9) "end" expected
  ```

  which has proved to be useful for situations where editors such as UltraEdit from IDM Computer Solutions or the SemWare Professional editor are configured to act as an IDE (Interactive Development Environment) for Cocol programming. In either version the format of such messages is easily modified by redefining the value of a `public string` declared in the `Errors` class outlined in the `Parser.frame` file:

  ```
  Errors.errMsgFormat
  ```

  either by editing the `Parser.frame` file, or by dynamically overwriting this string from within the compiler driver class.

- Use of the `-options m` directive in the extended version of Coco/R creates a source listing in a file `listing.txt` in the same directory as the `Grammar.ATG` file, in which any error messages are merged with the original source in a style exemplified in section 10.6.8 of the text.

- Use of the `-options n` directive results in the generation of a scanner and parser that use names for all terminals. This makes the generated source code easier to comprehend than if the default simple integer enumeration of these tokens is used.

3 Differences in the specification of the Cocol grammar

The extended versions of Coco/R introduce some variations on the form of a Cocol specification:

- The extended version of Cocol allows for either single or double quotes to demarcate the strings used to specify terminals and character sets. Thus, for example, the extended version will allow

  ```
  CHARACTERS /* Extended */
    vowels = 'aeiou' .
    digits = "0123456789" .
    lf     = '\n' .
  PRODUCTIONS
    Sequence = "begin" { Statement } 'end' .
  ```

  which would have to be specified as follows in the Linz version:

  ```
  CHARACTERS /* Linz */
    vowels = "aeiou" .
    digits = "0123456789" .
    lf     = '\n' .
  PRODUCTIONS
    Sequence = "begin" { Statement } "end" .
  ```

  This may be further clarified by an extract from the extended Cocol description of Cocol itself:

  ```
  CHARACTERS
    cr        = '\r'.
    lf        = '\n'.
    stringCh  = ANY - '"' - '\\' - cr - lf.
    charCh    = ANY - "'" - '\\' - cr - lf.
    printable = '\u0020' .. '\u007e'.
  TOKENS
    string    = '"' { stringCh | '\\' printable } '"' | /* pdt */
                "'" { charCh | '\\' printable } "'"  .
  ```

- The extended version of Cocol allows the sequences `<.` and `.>` to be used as alternatives for `<` and `>` when specifying attributes. This is to allow for the simple invocation of parser methods in situations like

```
Method<. a > b .>
```

which have to be handled in the Linz version by the introduction of an extra local variable.

- The extended version allows the notation `CHR(n)` for specifying a character value when defining character sets. This may be less confusing than using escape sequences, and also provides a measure of compatibility with earlier versions of Cocol. Thus, for example, one can write

  ```
  CHARACTERS /* Extended */
    control = CHR(1) .. CHR(31) .
    lf      = CHR(10) .
  IGNORE CHR(9) .. CHR(13) .
  ```

  as an alternative to

  ```
  CHARACTERS /* Linz */
    control = '\u0001' .. '\u001f' .
    lf      = '\n' .
  IGNORE '\t' .. '\r' .
  ```

- By default the scanner and parser produced by Coco/R use small integer values to distinguish token kinds. As discussed above, in the extended system the use of the `-options n` directive results in the generation of source code that uses names for the tokens. By default these names have a rather stereotyped form (for example "..." would be named "`pointpointpointSym`"). A `NAMES` clause in the scanner specification may be used to prefer user-defined names, or to help resolve name clashes (for example, between the default names that would be generated for "`point`" and "`.`"). As an example:

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

4 Differences in the interfaces to the generated code

Both the Linz "instantiated" version and the extended version provide users with the following view of the generated `Scanner` class:

```
public class Scanner {

  public Buffer buffer;

  public Scanner(string fileName);
  // Opens and reads source file specified by fileName

  public Scanner(Stream s);
  // Opens and reads source file from stream s

  public Token Scan();
  // Returns next token from source file

  public Token Peek();
  // Returns next token from source file but does not consume it

  public Token ResetPeek();
  // Resets the scan position after calling Peek

} // end Scanner
```

The scanner is implemented by a system where the source is held within a `Buffer` class that also provides facilities for clients to retrieve characters or strings from it:

```
public class Buffer {

  public const char EOF = (char) 256;

  public Buffer(Stream s);
  // Constructor to use source stream s

  public int Read();
  // Returns the next character in the buffer, or 256
  // if the end of the buffer has been reached
```

```
    // and advances the internal buffer pointer

    public int Peek();
    // Returns the next character in the buffer, or 256
    // if the end of the buffer has been reached
    // but does not advance the internal buffer pointer

    public int Pos {get; set;}
    // Gets or sets the internal buffer pointer

    public string GetString(int beg, int end);
    // Returns the text held between positions beg and end

  } // end Buffer
```

For simple applications, users need not concern themselves with the details of the `Scanner` or `Buffer` classes, since Coco/R generates all calls needed to the public methods declared within them. However, the `Scanner` methods return objects of the `Token` class, and semantic actions may need to access the fields of such objects.

The `Token` class is defined in both versions by

```
  public class Token {

    public int kind;    // token code (EOF has the code 0)
    public int pos;     // token position in the source buffer (starting at 0)
    public int line;    // token line number (starting at 1)
    public int col;     // token column number (starting at 0)
    public string val;  // token lexeme

  } // end Token
```

Of more interest still is the `Parser` class. Users of the Linz system see this as defined by:

```
  public class Parser {

    public Scanner scanner; // the scanner for this parser
    public Errors errors;   // the error message handler

    public Parser(Scanner scanner, Errors errors);
    // Constructor

    public void Parse();
    // Parses the source

    public void SemErr(string msg);
    // Generates semantic error message described by msg

    public Token t;         // last recognized token
    public Token la;        // look ahead token

  } // end Parser
```

Some additional functionality has been provided in the extended system:

```
  public class Parser {

    public Scanner scanner; // the scanner for this parser
    public Errors errors;   // the error message handler

    public Parser(Scanner scanner, Errors errors);
    // Constructor

    public void Parse();
    // Parses the source

    public void SemErr(string msg);
    // Generates semantic error message described by msg

    private Token token;   // last recognized token
    private Token la;      // look ahead token

    public bool Successful();
    // Returns true if no errors occurred while parsing

    public void SemError(string msg);
    // Generates semantic error message described by msg
    public void Warning(string msg);
    // Generates warning message described by msg

    public string LexString();
    // Returns lexeme token.val

    public string LookAheadString();
    // Returns lexeme la.val

  } // end Parser
```

The use of `token` rather than `t` for the most recently parsed token (as used in the Linz versions) is compatible with some earlier releases of Coco/R and avoids the use of a very simple identifier that users might accidentally introduce into their attributes and actions, leading inevitably to some confusion.

The other additional methods also provide for a measure of compatibility with earlier versions of Coco/R.

The `Errors` class, normally used internally from the `Parser` class, is defined in the Linz version by

```
  public class Errors {

    public int count = 0;
    // Stores the number of errors reported while parsing

    public string errMsgFormat = "-- line {0} col {1}: {2}";
    // default formatter for reporting error messages to the console

    public void SynErr(int line, int col, int n);
    // Reports on a syntactic error (extended automatically)

    public void SemErr(int line, int col, int n);
    // Reports a semantic error denoted by integer n

    public void Error(int line, int col, string msg);
    // Reports an error message described by msg

    public void Exception(string msg);
    // Reports an exceptional error and terminates the application

  }
```

In the extended version the philosophy is slightly different:

```
  public class Errors {

    public int count = 0;
    // Stores the number of errors reported while parsing

    public string errMsgFormat = "file {0} : ({1}, {2}) {3}";
    // default formatter for reporting error messages to the console

    public void SynErr(int line, int col, int n);
    // Reports on a syntactic error (extended automatically)

    public void SemErr(int line, int col, int n);
    // Reports a semantic error denoted by integer n

    public void Error(int line, int col, string msg);
    // Reports an error message described by msg

    public void Exception(string msg);
    // Reports an exceptional error and terminates the application

    public Errors(Scanner scanner, string fileName, string dir, bool merge);
    // Constructor for error handler.  Prepares to create merged listing
    // if merge = true for source file fileName in directory dir

    public void Summarize();
    // Summarize errors at end of compilation

    public void StoreError(int line, int col, string msg);
    // Stores error report denoted by msg in internal data structure

    public void Warn(int line, int col, string msg);
    // Reports a warning message described by msg (does not increment count)

  }
```

Unless a merged source/error listing is required, error messages are simply reflected to the console using the format `errMsgFormat`, which can be changed by the user to obtain some custom format of error messages. The placeholder {0} is filled with the source file name {1} is filled with the line number, {2} is filled with the column number, and {3} is filled with the error message.

If a merged source/error listing is required, methods `SynErr`, `SemErr`, `SemError`, `Warn` and `Error` store error messages in an internal data structure. The production of the listing is initiated, after parsing is completed, by calling the method `Summarize`.

5 Other differences in the implementations

The extensions outlined above have given rise to quite a number of differences in the source code for the Linz and extended versions. As recommended in the GPL, these are marked in the source by comments of the form `/* pdt */` for readers who care to examine them. Other minor internal differences come about from an attempt to handle the different conventions for line separators in Unix, Mac and Dos systems consistently, from attempts to improve the labelling of the trace output and from attempts to perform internal I/O operations more effectively.