The Compiler Generator Coco/R

Extended User Manual – Version 2

Pat Terry
Rhodes University
September 2005

This document has been modified, with permission, from the original by Hanspeter Mössenböck, so as to describe the extensions to Coco/R implemented to support the text "Compiling with C# and Java". The important modifications have been clearly marked in red.

This version describes the system that generates "instantiable" scanners and parsers, rather than the "static" ones described in the book, which were the sort produced by earlier releases of Coco/R

User Manual

Hanspeter Mössenböck
Johannes Kepler University Linz
Institute of System Software
June 2004

Coco/R[1] is a compiler generator, which takes an attributed grammar of a source language and generates a scanner and a parser for this language. The scanner works as a deterministic finite automaton. The parser uses recursive descent. LL(1) conflicts can be resolved by a multi-symbol lookahead or by semantic checks. Thus the class of accepted grammars is LL(*k*) for an arbitrary *k*.

There are versions of Coco/R for Java, C#, C++, Delphi, Modula-2, Oberon and other languages. This manual describes the versions for C# and Java implemented at the University of Linz.

Compiler Generator Coco/R,
Copyright © 1990, 2005 Hanspeter Mössenböck, University of Linz

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

As an exception, it is allowed to write an extension of Coco/R that is used as a plugin in non-free software.

If not otherwise stated, any source code generated by Coco/R (other than Coco/R itself) does not fall under the GNU General Public License.

[1] Coco/R stands for *compiler compiler* generating *recursive descent* parsers.

Contents

1. Overview

Coco/R is a compiler generator, which takes an attributed grammar for a source language and generates a scanner and a recursive descent parser for this language. The user has to supply a main class that calls the parser as well as semantic classes (e.g. a symbol table handler and a code generator) that are used by semantic actions in the parser. This is shown in Figure 1.

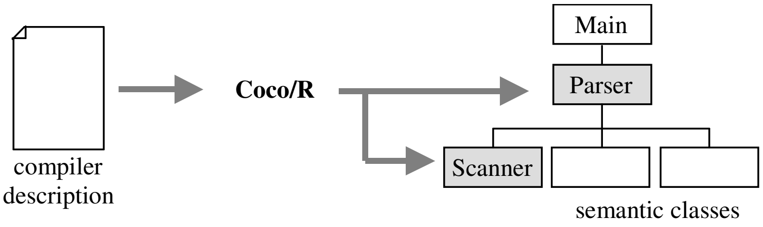

Figure 1 Input and output of Coco/R

1.1 Sample Production

In order to give you an idea of how attributed grammars look like in Coco/R, let us look at a sample production for variable declarations in a Pascal-like language.

```
VarDeclaration<ref int adr>  (. string name; TypeDesc typ; .)
= Ident<out name>            (. Obj x = symTab.Enter(name);
                                int n = 1; .)
  { ',' Ident<out name>      (. Obj y = symTab.Enter(name);
                                x.next = y; x = y;
                                n++; .)
  }
  ':' Type<out typ>          (. adr += n * typ.size;
                                for (int a = adr; x != null; x = x.next) {
                                  a -= typ.size;
                                  x.adr = a;
                                } .)
  ';' .
```

The core of this specification is the *EBNF production*

```
VarDeclaration = Ident { ',' Ident } ':' Type ';'.
```

It is augmented with attributes and semantic actions. The *attributes* (e.g. `<out name>`) specify the parameters of the symbols. There are input attributes (e.g. `<x, y>`) and output attributes (e.g. `<out z>` or `<ref z>`). A *semantic action* is a piece of code that is written in the target language of Coco/R (e.g. in C# or Java) and is executed by the generated parser at its position in the production.

1.2 Sample Parsing Method

Every production is translated by Coco/R into a parsing method. The method for `VarDeclaration`, for example, looks like this in C# (code parts originating from attributes or semantic actions are shown in grey):

```
static void VarDeclaration(ref int adr) {
   string name; TypeDesc typ;
   Ident(out name);
   Obj x = symTab.Enter(name);
   int n = 1;
   while (la.kind == comma) {
      Get();
      Ident(out name);
      Obj y = symTab.Enter(name);
      x.next = y; x = y;
      n++;
   }
   Expect(colon);
   Type(out typ);
   adr += n * typ.size;
   for (int a = adr; x != null; x = x.next) {
      a -= typ.size;
      x.adr = a;
   }
   Expect(semicolon);
}
```

Coco/R also generates a scanner that reads the input stream and returns a stream of tokens to the parser.

1.3 Summary of Features

Scanner

- The scanner is specified by a list of token declarations. Literals (e.g. `"if"` or `"while"`) do not have to be explicitly declared as tokens, but can be used directly in the productions of the grammar.
- The scanner is implemented as a deterministic finite automaton (DFA). Therefore the terminal symbols (or tokens) have to be described by a regular EBNF grammar.
- One can specify one or more kinds of comments that a scanner will ignore. Provision is made for allowing comments to be nested.
- Tokens must be made up of characters from the extended ASCII set (i.e. 256 values).
- The scanner can be made case-sensitive or case-insensitive.
- The scanner can recognize tokens that are distinguishable only on their context in the input stream.
- The scanner can read from any input stream (not just from a file). However, all input must come from a single stream (no includes).
- The scanner can handle so-called *pragmas*, which are tokens that are not part of the syntax but can occur anywhere in the input stream (e.g. compiler directives or end-of-line characters).
- The user can suppress the generation of a scanner and can provide a hand-written scanner instead.

Parser

- The parser is specified by a set of EBNF productions with attributes and semantic actions. The productions allow for alternatives, repetition and optional parts. Coco/R translates the productions into a recursive descent parser which is small and efficient.
- Nonterminal symbols can have a number of input and output attributes (the Java version allows just one output attribute, which may, however, be an object of a suitable composite class). Terminal symbols do not have explicit attributes, but the tokens returned by the scanner contain information that can be viewed as attributes. All attributes are evaluated during parsing (i.e. the grammar is processed as an L-attributed grammar).
- Semantic actions can be placed anywhere in the grammar (not just at the end of productions). They may contain arbitrary statements or declarations written in the language of the generated parser (e.g. C# or Java).
- The special symbol ANY can be used to denote a set of complementary tokens.
- In principle, the grammar must be LL(1). However, Coco/R can also handle non-LL(1) grammars by using so-called *resolvers* that make a parsing decision based on a multi-symbol lookahead or on semantic information.
- Every production can have its own local variables. In addition to these, one can declare global variables or methods, which are incorporated as static fields and methods of the parser. Semantic actions can also access other objects or methods from user-written classes or from library classes.
- Coco/R checks the grammar for completeness, consistency and non-redundancy. It also reports LL(1) conflicts.
- The error messages printed by the generated parser can be configured to conform to a user-specific format.
- The generated parser and scanner can be specified to belong to a certain namespace (or package).

2. Input Language

This section specifies the compiler description language *Cocol/R* that is used as the input language for Coco/R . A compiler description consists of a set of grammar rules that describe the lexical and syntactical structure of a language as well as its translation to a target language.

2.1 Vocabulary

The basic elements of Cocol/R are identifiers, numbers, characters and strings, which are defined as follows:

```
ident  = letter { letter | digit }.
number = digit { digit }.
string = '"' {anyButQuote|escape} '"' | "'" {anyButApostrophe|escape} "'" .
char   = '"' (anyButQuote|escape) '"' | "'" (anyButApostrophe|escape) "'" .
```

Upper case letters are distinct from lower case letters. Strings may not extend across multiple lines. Strings and characters may contain the following escape sequences:

`\\`	backslash	`\r`	carriage return	`\f`	form feed
`\'`	apostrophe	`\n`	new line	`\a`	bell
`\"`	quote	`\t`	horizontal tab	`\b`	backspace
`\0`	null character	`\v`	vertical tab	`\uxxxx`	hex char value

The following identifiers are reserved keywords (in the C# version of Cocol/R the identifier `using` is also a keyword, in the Java version the identifier `import`):

```
ANY           COMPILER      IF            NESTED        SYNC
CHARACTERS    CONTEXT       IGNORE        out           TO
CHR           END           IGNORECASE    PRAGMAS       TOKENS
COMMENTS      FROM          NAMES         PRODUCTIONS   WEAK
```

Comments are enclosed in `/*` and `*/` and may be nested. Alternatively they can start with // and go to the end of the line.

EBNF

All syntax descriptions in Cocol/R are written in Extended Backus-Naur Form (EBNF) [Wirth77]. By convention, identifiers starting with a lower case letter denote terminal symbols, identifiers starting with an upper case letter denote nonterminal symbols. Strings denote themselves. The following meta-characters are used:

symbol	meaning	example	
=	separates the sides of a production	A = a b c .	
.	terminates a production	A = a b c .	
|	separates alternatives	a b | c | d e	means a b or c or d e
( )	groups alternatives	(a | b) c	means a c or b c
[ ]	option	[a] b	means a b or b
{ }	iteration (0 or more times)	{a} b	means b or a b or a a b or ...

Attributes are written between `<` and `>` or between `<.` and `.>`. Semantic actions are enclosed in `(.` and `.)`. The operators `+` and `-` are used to form character sets.

2.2 Overall Structure

A Cocol/R compiler description has the following structure:

```
Cocol =
  [ Imports ]
  "COMPILER" ident
  [ GlobalFieldsAndMethods ]
  ScannerSpecification
  ParserSpecification
  "END" ident '.'
  .
```

The name after the keyword `COMPILER` is the *grammar name* and must match the name after the keyword `END`. The grammar name also denotes the topmost nonterminal symbol (the *start* or *goal symbol*). The parser specification **must** contain a production for this symbol.

Imports. In front of the keyword `COMPILER` one can import namespaces (in C#) or packages (in Java), for example:

```
using System;
using System.Collections;
```

GlobalFieldsAndMethods. After the grammar name one may declare additional fields and methods to be incorporated into the generated parser, for example:

```
int sum;

void Add(int x) {
   sum = sum + x;
}
```

These declarations are written in the language of the generated parser (i.e. in C# or in Java) and are not checked by Coco/R. Such fields and methods can then be used in the semantic actions of the parser specification.

The remaining parts of the compiler description specify the scanner and the parser that are to be generated. They are now described in more detail.

2.3 Scanner Specification

A scanner has to read source text, skip meaningless characters, recognize tokens and pass them to the parser. The scanner specification consists of six optional parts:

```
ScannerSpecification =
  [ "IGNORECASE"]
  [ "CHARACTERS" { SetDecl } ]
  [ "TOKENS"   { TokenDecl } ]
  [ "NAMES"    { NameDecl } ]
  [ "PRAGMAS" { PragmaDecl } ]
  { CommentDecl }
  { WhiteSpaceDecl }.
```

2.3.1 Character sets

This section allows the user to declare character sets such as letters or digits. Their names can then be used in the other sections of the scanner specification. Coco/R grammars are expressed in an extended ASCII character set (256 characters).

```
SetDecl   = ident '=' Set.
Set       = BasicSet { ('+' | '-') BasicSet }.
BasicSet = string | ident | Char [ ".." Char ] | "ANY".
Char      = "CHR" '(' number ')' | char .
```

`SetDecl` associates a name with a character set. Basic character sets are denoted as:

`string`	a set consisting of all the characters in the string
`ident`	a previously declared character set with this name
`char`	a set containing the character `char`
`char1 .. char2`	the set of all characters from `char1` to `char2` `inclusive`
`ANY`	the set of all characters in the range 0 .. 255

Character sets may be formed from basic sets using the operators

`+`	set union
`-`	set difference

Examples

```
CHARACTERS

  digit    = "0123456789".      /* the set of all digits */
  hexDigit = digit + "ABCDEF".  /* the set of all hexadecimal digits */
  letter   = 'A' .. 'Z'.        /* the set of all upper case letters */
  eol      = '\r'.              /* the end-of-line character */
  noDigit  = ANY - digit.       /* any character that is not a digit */
```

```
    control  = CHR(0) .. CHR(31). /* ASCII control characters */
```

2.3.2 Tokens

This is the main section of the scanner specification, in which the tokens (or terminal symbols) of the language are declared. Tokens may be divided into literals and token classes.

- *Literals* (such as `while` or `>=`) have a fixed representation in the source language. In the grammar they are written as strings (e.g. `"while"` or `">="`) and denote themselves. They do not have to be declared in the tokens section, but can be implicitly declared at their first use in the productions of the grammar.
- *Token classes* (such as identifiers or numbers) have a certain structure that must be explicitly declared by a regular expression in EBNF. There are usually many instances of a token class (e.g. many different identifiers), which have the same token code, but different lexeme values.

The syntax of token declarations is as follows:

```
  TokenDecl   = Symbol [ '=' TokenExpr '.' ].
  TokenExpr   = TokenTerm { '|' TokenTerm }.
  TokenTerm   = TokenFactor { TokenFactor } [ "CONTEXT" '(' TokenExpr ')' ].
  TokenFactor =   Symbol
                | '(' TokenExpr ')'
                | '[' TokenExpr ']'
                | '{' TokenExpr '}'.
  Symbol      = ident | string | char.
```

A token declaration defines the syntax of a terminal symbol by a regular EBNF expression. This expression may contain strings or character constants denoting themselves (e.g. `">="` or `';'`) as well as names of character sets (e.g. `letter`) denoting an arbitrary character from this set. It must not contain other token names, which implies that EBNF expressions in token declarations cannot be recursive.

Examples

```
  TOKENS

    ident  = letter { letter | digit | '_' }.
    number =   digit { digit }
             | "0x" hexDigit hexDigit hexDigit hexDigit.
    float  = digit {digit} '.' {digit} [ 'E' ['+'|'-'] digit {digit} ].
```

The token declarations need not be LL(1), as can be seen in the declaration of `number`, where both alternatives can start with a `'0'`. Coco/R automatically resolves any ambiguities and generates a deterministic finite scanner automaton.

Tokens may be declared in any order. However, if a token is declared as a literal that might match an instance of a more general token, the literal has to be declared *after* the more general token.

Example

```
  ident = letter { letter | digit }.
  while = "while".
```

Since the string `"while"` matches both the tokens `while` and `ident`, the declaration of `while` must come after the declaration of `ident`. In principle, literal tokens do not have to be declared in the token declarations at all, but can simply be introduced directly in the productions of the grammar. In some situations, however, it makes sense to declare them explicitly, perhaps to get a token name for them that can be used in resolver methods (see Section 2.4.6).

Context-dependent tokens. The `CONTEXT` phrase in a `TokenTerm` means that the term is only recognized if its context (i.e. the characters that follow the term in the input stream) matches the `TokenExpr` specified in brackets. Note that the `TokenExpr` is *not* part of the token.

Example

```
number = digit { digit }
       | digit { digit } CONTEXT ("..").
float  = digit { digit } '.' { digit } [ 'E' ['+'|'-'] digit { digit } ].
```

The `CONTEXT` phrase in this example allows the scanner to distinguish between `float` tokens (e.g. `1.23`) and integer ranges (e.g. `1..2`) that could otherwise not be scanned with a single character lookahead. This works as follows: after having read `"1."` the scanner still works on both tokens. If the next character is a `'.'` the characters `".."` are pushed back to the input stream and a `number` token with the value `1` is returned to the parser. If the next character is not a `'.'` the scanner continues with the recognition of a `float` token.

Hand-written scanners. If the right-hand sides of the token declarations are missing no scanner is generated. This gives the user the chance to provide a hand-written scanner, which must conform to the interface described in Section 3.3.1.

Example

```
TOKENS
  ident
  number
  "if"
  "while"
  ...
```

Tokens are assigned numbers in the order of their declaration. The first token gets the number `1`, the second the number `2`, and so on. The number `0` is reserved for the end-of-file token. The hand-written scanner must return the token numbers according to these conventions. In particular, it must return an end-of-file token if no more input is available.

It is hardly ever necessary to supply a hand-written scanner, because the scanner generated by Coco/R is highly optimized. A user-supplied scanner would be needed, for example, if the scanner were required to process *include* directives.

2.3.3 Pragmas

Pragmas are tokens that may occur anywhere in the input stream (for example, end-of-line symbols or compiler directives). It would be too tedious to handle all their possible occurrences in the grammar. Therefore they are excluded from the token stream that is passed to the parser. Pragmas are declared like tokens, but they may have a semantic action associated with them that is executed whenever they are recognized by the scanner.

```
PragmaDecl = TokenDecl [SemAction].
SemAction  = "(." ArbitraryStatements ".)".
```

Example

```
PRAGMAS
  option = '$' { letter }.    (. foreach (char ch in la.val)
                                   if (ch == 'A') ...
                                   else if (ch == 'B') ...
                                   ...    .)
```

This pragma defines a compiler option that can be written, for example, as `$A`. Whenever it occurs in the input stream it is *not* forwarded to the parser but immediately processed by executing its associated semantic action. Note that `la.val` accesses the value of the lookahead token `la`, which is in this case the pragma that was just read (see Section 3.3.4).

2.3.4 Comments

Comments are difficult to specify with regular expressions; nested comments are quite impossible to specify in that way. This makes it necessary to have a special construct to define their structure.

Comments are declared by specifying their opening and closing brackets. The keyword `NESTED` denotes that they can be nested.

```
CommentDecl = "COMMENTS" "FROM" TokenExpr "TO" TokenExpr ["NESTED"].
```

Comment delimiters must be sequences of 1 or 2 characters, which can be specified as literals or as single-element character sets. They must not be structured (for example with alternatives). It is possible to declare more than one structure for comments.

Example

```
COMMENTS FROM "/*" TO "*/" NESTED
COMMENTS FROM "//" TO eol
```

Alternatively, if comments are not to be nested one can define them as pragmas, eg.

```
CHARACTERS
  other = ANY - '/' - '*'.
PRAGMAS
  comment = "/*" {'/' | other | '*' {'*'} other} '*' {'*'} '/'.
```

This has the advantage that such comments can be processed semantically, for example, by counting them or by processing compiler options within them.

2.3.5 White space

Some characters that appear between tokens, such as blanks, tabulators or end-of-line separators are usually considered as white space that should simply be ignored by the scanner. Blanks are ignored by default. If other inter-token characters are to be ignored the user has to specify them in the following way:

```
WhiteSpaceDecl = "IGNORE" Set.
```

Example

```
IGNORE '\t' + '\r' + '\n'
```

2.3.6 Case sensitivity

Some languages such as Pascal or XML are case insensitive. In Pascal, for example, one can also write the keyword `while` as `While` or `WHILE`. By default, Coco/R generates scanners that are case sensitive. If this is not desired, one has to include the directive `IGNORECASE` at the beginning of the scanner specification.

The effect of `IGNORECASE` is that all input to the scanner is treated in a case-insensitive way. The production

```
WhileStatement = "while" '(' Expr ')' Statement.
```

would then also recognize while statements that start with `While` or `WHILE`. Similarly, the declaration:

```
TOKENS
  float = digit { digit } '.' [ 'E' ( '+'|'-' ) digit { digit } ].
```

would cause the scanner to recognize not only `1.2E2` but also `1.2e2` as a `float` token. However, the original casing of tokens is preserved in the `val` field of every token (see Section 3.3.2), so that the lexical value of tokens such as identifiers and strings is delivered exactly as it was written in the input text.

2.3.7 Named tokens

The scanner and parser produced by Coco/R use small integer values to distinguish token kinds. 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 (typically a `$N` pragma), Coco/R can generate code that uses names for all the tokens. By default these names have a rather stereotyped form (for example "..." would be named "`pointpointpointSym`"). The facility exists 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 = ident "=" ( ident | string ) "." .
```

Example

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

2.4 Parser Specification

The parser specification is the main part of a compiler description. It contains the productions of an attributed grammar, which specify the syntax of the language to be parsed as well as its translation.

```
ParserSpecification = "PRODUCTIONS" { Production }.
Production = ident [ FormalAttributes ] [ LocalDecl ] '=' Expression '.'.
Expression = Term {'|' Term}.
Term       = [ [ Resolver ] Factor { Factor } ].
Factor     = [ "WEAK" ] Symbol [ ActualAttributes ]
           | '(' Expression ')'
           | '[' Expression ']'
           | '{' Expression '}'
```

```
              | "ANY"
              | "SYNC"
              | SemAction.
Symbol       = ident | string | char.
SemAction    = "(." ArbitraryStatements ".)".
LocalDecl    = SemAction.
FormalAttributes = '<' ArbitraryText '>' | '<.' ArbitraryText '.>'
ActualAttributes = '<' ArbitraryText '>' | '<.' ArbitraryText '.>'.
Resolver     = "IF" '(' { ANY } ')'.
```

2.4.1 Productions

A production specifies the syntactical structure of a nonterminal symbol. It consists of a left-hand side and a right-hand side which are separated by an equal sign. The left-hand side specifies the name of the nonterminal together with its formal attributes and the local variables of the production. The right-hand side consists of an EBNF expression that specifies the structure of the nonterminal, as well as its translation in form of attributes and semantic actions.

The productions may be given in any order. References to as yet undeclared nonterminals are allowed (any name that has not previously been declared is considered to be a forward reference to a nonterminal symbol). For every nonterminal there must be exactly one production (which may specify alternative right-hand sides). In particular, there must be a production for the grammar name, which is the start symbol of the grammar.

2.4.2 Semantic Actions

A semantic action is a piece of code written in the target language of Coco/R (i.e. in C# or in Java). It is executed by the generated parser at the position where it has been specified in the grammar. Semantic actions are simply copied to the generated parser without being checked by Coco/R.

A semantic action can also contain the declarations of local variables. Every production has its own set of local variables, which are retained in recursive productions. The optional semantic action on the left-hand side of a production (`LocalDecl`) is intended for such declarations, but variables can also be declared in any other semantic action.

Here is an example that counts the number of identifiers in an identifier list:

```
IdentList =
  ident                 (. int n = 1; .)
  { ',' ident           (. n++; .)
  }                     (. Console.WriteLine("n = " + n); .)
  .
```

As a matter of style, it is good practice to write all syntax parts of each production on the left side of a Cocol file, with all semantic actions on the right, as illustrated above. This makes a production better readable because the syntax is separated from its processing.

Semantic actions can access not only local variables but also fields and methods declared at the beginning of the attributed grammar (see Section 2.2) as well as fields and methods of other classes.

2.4.3 Attributes

Productions are considered as (and are actually translated to) parsing methods. The occurrence of a nonterminal on the right-hand side of a production can be viewed as a call of that nonterminal's parsing method.

Nonterminals may have attributes, which correspond to parameters of the nonterminal's parsing method. There are *input attributes*, which are used to pass values to the production of a nonterminal, and *output attributes*, which are used to return values from the production of a nonterminal to its caller (i.e. to the place where this nonterminal occurs in some other production).

As with parameters, we distinguish between *formal attributes*, which are specified at the nonterminal's declaration on the left-hand side of a production, and *actual attributes*, which are specified at the nonterminal's occurrence on the right-hand side of a production.

Attributes in C#. A formal attribute looks like a parameter declaration. In C#, output attributes must be preceded by the keyword `out` or `ref`. The following example declares a nonterminal `S` with an input attribute `x` and two output attributes `y` and `z`:

```
S <int x, out int y, ref string z> = ... .
```

An actual attribute looks like an actual parameter. Actual input attributes may be expressions, which are evaluated and assigned to the corresponding formal attributes. In C#, actual output attributes must be preceded by the keywords `out` or `ref`. They are passed by reference like output parameters in C#. Here is an example (`a` and `b` are assumed to be of type `int`, `c` is assumed to be of type `string`):

```
... S <3*a + 1, out b, ref c> ...
```

The production of the nonterminal `S` is translated to the following parsing method:

```
void S(int x, out int y, ref string z) {
   ...
}
```

Attributes in Java. Java methods cannot have output parameters. However, the Java version of Coco/R provides for a single output attribute which is passed to the caller as a return value. This return value can be an object of a class that contains multiple members.

If a nonterminal is to have an output attribute this must be declared as the first attribute. It is denoted by the keyword `out` both in its declaration and in its use. The following example shows a nonterminal `T` with an output attribute `x` and two input attributes `y` and `z` (for compatibility with older versions of Coco/R the symbol `'^'` can be substituted for the keyword `out`):

```
T<out int x, char y, int z> = ... .
```

This nonterminal might be used as follows:

```
... T<out a, 'b', c+3> ...
```

The production of the nonterminal `T` is translated to the following parsing method:

```
int T(char y, int z) {
   int x;
   ...
   return x;
}
```

Note that if expressions that are passed as actual input attributes (both in C# and in Java) are bracketed by '<' and '>' they may not contain the operator '>', which is the closing attribute bracket. Such expressions might be assigned to a temporary variable, which can then be passed as an attribute. The extended version of Coco/R described here also allows for attribute lists to be bracketed by '<.' and '.>'.

Coco/R checks that nonterminals with attributes are always used with attributes and that nonterminals without attributes are always used without attributes. However, it does not check the correspondence between formal and actual attributes, which is left to the compiler of the target language.

Attributes of terminal symbols. Terminal symbols do not have attributes in Cocol/R. For every token, however, the scanner returns the token value (i.e. the token's string representation) as well as the line and column number of the token (see Section 3.3.4). This information can be viewed as output attributes of that token. If users wish to access this data they can wrap a token into a nonterminal with the desired attributes, for example:

```
Ident <out string name> =
  ident                       (. name = token.val; .) .

Number <out int value> =
  number                      (. value = Convert.ToInt32(token.val); .) .
```

The variable token is the most recently recognized token. Its field token.val holds the textual representation of the token (see Section 3.3.4).

2.4.4 The Symbol ANY

In the productions of the grammar the symbol ANY denotes any token that is not an alternative to that ANY symbol in the context where it appears. It can be used to conveniently parse structures that contain arbitrary text. The following production, for example, processes an attribute list in Cocol/R and returns the number of characters between the angle brackets:

```
Attributes < out int len> =
  '<'                         (. int beg = token.pos + 1; .)
    { ANY }
  '>'                         (. len = token.pos - beg; .) .
```

In this example the token '>' is an implicit alternative of the ANY symbol in curly braces. The meaning is that this ANY matches any token except '>'. token.pos is the source text position of the most recently recognized token (see Section 3.3.4).

Here is another example that counts the number of statements in a block:

```
Block <out int stmts> =       (. int n; .)
'{'                           (. stmts = 0; .)
{   ';'                       (. stmts++; .)
  | '{' Block<out n> '}'      (. stmts += n; .)
  | ANY
}
'}'.
```

In this example the ANY matches any token except ';', '{' and '}', which are its alternatives in this context ('{' is a terminal start symbol of Block).

2.4.5 LL(1) Conflicts

Recursive descent parsing requires that the grammar of the parsed language is LL(1) (i.e. parsable from **L**eft to right with **L**eft-canonical derivations and **1** lookahead symbol). This means that at any point in the grammar the parser must be able to decide on the basis of a single lookahead symbol which of several possible alternatives have to be selected. The following production, for example, is not LL(1):

```
Statement =   ident '=' Expression ';'
            | ident '(' [ActualParameters] ')' ';'
            | ... .
```

Both alternatives start with the symbol `ident`. If the parser comes to the beginning of a `Statement` and finds an `ident` as the next input token, it cannot distinguish between the two alternatives. However, this production can easily be transformed to

```
Statement = ident (   '=' Expression ';'
                    | '(' [ ActualParameters ] ')' ';'
                  )
          | ... .
```

where all alternatives start with distinct symbols and the LL(1) conflict has disappeared.

LL(1) conflicts can arise not only from explicit alternatives like those in the example above but also from implicit alternatives that are hidden in optional or iterative EBNF expressions. The following list shows how to check for LL(1) conflicts in these situations (Greek symbols denote arbitrary EBNF expressions such as `a[b]c`; *first*(α) denotes the set of terminal start symbols of the EBNF expression α; *follow*(`A`) denotes the set of terminal symbols that can follow the nonterminal `A` in any other production):

- **Explicit alternatives**

  $\texttt{A} = \alpha|\beta|\gamma.$ check that $\mathit{first}(\alpha) \cap \mathit{first}(\beta) = \{\} \wedge \mathit{first}(\alpha) \cap \mathit{first}(\gamma) = \{\} \wedge \mathit{first}(\beta) \cap \mathit{first}(\gamma) = \{\}$.

  $\texttt{A} = (\alpha|)\beta.$ check that $\mathit{first}(\alpha) \cap \mathit{first}(\beta) = \{\}$

  $\texttt{A} = (\alpha|).$ check that $\mathit{first}(\alpha) \cap \mathit{follow}(\texttt{A}) = \{\}$

- **Options**

  $\texttt{A} = [\alpha]\ \beta.$ check that $\mathit{first}(\alpha) \cap \mathit{first}(\beta) = \{\}$

  $\texttt{A} = [\alpha].$ check that $\mathit{first}(\alpha) \cap \mathit{follow}(\texttt{A}) = \{\}$

- **Iterations**

  $\texttt{A} = \{\alpha\}\ \beta.$ check that $\mathit{first}(\alpha) \cap \mathit{first}(\beta) = \{\}$

  $\texttt{A} = \{\alpha\}.$ check that $\mathit{first}(\alpha) \cap \mathit{follow}(\texttt{A}) = \{\}$

It would be very tedious and error-prone to check all these conditions manually for a grammar of a realistic size. Fortunately, Coco/R does that automatically. For example, processing the grammar

```
A = (a | B C d).
B = [ b ] a.
C = c { d }.
```

will result in the following LL(1) warnings:

```
LL(1) warning in A: a is the start of several alternatives
LL(1) warning in C: d is the start & successor of a deletable structure
```

The first conflict arises because `B` can start with an `a`. The second conflict comes from the fact that `C` may be followed by a `d`, and so the parser does not know whether it should do another iteration of `{d}` in `C` or terminate `C` and continue with the `d` outside.

Another situation that leads to a conflict is when an expression in curly or square brackets is deletable, e.g.:

```
A = [B] a.
B = {b}.
```

If the parser tries to recognize `A` and sees an `a` it cannot decide whether to enter the deletable symbol `B` or to skip `[B]`. Therefore Coco/R prints the warning:

```
LL(1) warning in A: contents of [...] or {...} must not be deletable
```

Note that Coco/R reports LL(1) conflicts as warnings, not as errors. Whenever a parser generated by Coco/R sees two or more alternatives that can start with the same token it always chooses the first one. If this is what the user intends then everything is fine, as in the well-known example of the *dangling else* that occurs in many programming languages:

```
Statement = "if" '(' Expression ')' Statement ["else" Statement]
          | ... .
```

Input for this grammar like

```
if (a > b) if (a > c) max = a; else max = b;
```

is ambiguous: does the `"else"` belongs to the inner or to the outer if statement? The LL(1) conflict arises because

$$\mathit{first}(\texttt{"else" Statement}) \cap \mathit{follow}(\texttt{Statement}) = \{\texttt{"else"}\}$$

However, this is not a big problem, because the parser chooses the first matching alternative, which is the `"else"` of the inner if statement. This is exactly what we want.

Resolving LL(1) conflicts by grammar transformations

If Coco/R reports an LL(1) conflict the user should try to eliminate it by transforming the grammar as is shown in the following examples.

Factorization. Most LL(1) conflicts can be resolved by factorization, i.e. by extracting the common parts of conflicting alternatives and moving them to the front. For example, the production

```
A = a b c | a b d.
```

can be transformed to

```
A = a b (c | d).
```

Left recursion. Left recursion always represents an LL(1) conflict. In the production

```
A = A b | c.
```

both alternatives start with `c` (because *first*(`A`) = {`c`}). However, left recursion can always be transformed into an iteration, e.g. the previous production becomes

```
A = c {b}.
```

Hard conflicts. Some LL(1) conflicts cannot be resolved by grammar transformations. Consider the following (simplified) productions from the C# grammar:

```
Expr   = Factor {'+' Factor}.
Factor =   '(' ident ')' Factor   /* type cast */
         | '(' Expr ')'           /* nested expression */
         | ident | number.
```

The conflict arises, because two alternatives of `Factor` start with `'('`. Even worse, `Expr` can also be derived to an `ident`. There is no way to get rid of this conflict by transforming the grammar. The only way to resolve it is to look at the `ident` following the `'('`: if it denotes a *type* the parser has to select the first alternative otherwise the second one. We shall deal with this kind of conflict resolution in Section 2.4.6.

Readability issues. Some grammar transformations can degrade the readability of the grammar. Consider the following example (again taken from a simplified form of the C# grammar):

```
UsingClause = "using" [ ident '=' ] Qualident ';'.
Qualident   = ident { '.' ident }.
```

The conflict is in `UsingClause` where both `[ ident '=' ]` and `Qualident` start with `ident`. Although this conflict could be eliminated by transforming the production to

```
UsingClause = "using" ident (   { '.' ident }
                              | '=' Qualident
                            )';'.
```

the readability would clearly deteriorate. It is better to resolve this conflict as shown in Section 2.4.6.

Semantic issues. Finally, factorization is sometimes inhibited by the fact that the semantic processing of conflicting alternatives differs, e.g.:

```
A = ident (. x = 1; .) { ',' ident (. x++; .) } ':'
  | ident (. Foo(); .) { ',' ident (. Bar(); .) } ';'.
```

The common parts of these two alternatives cannot be factored out, because each alternative has to be processed semantically in its own way. Again this problem can be solved with the technique explained in Section 2.4.6.

2.4.6 LL(1) Conflict Resolvers

A conflict resolver is a Boolean expression that is inserted into the grammar at the beginning of the first of two conflicting alternatives and decides, using a multi-symbol lookahead or a semantic check, whether this alternative matches the actual input. If the resolver yields `true`, the alternative prefixed by the resolver is selected, otherwise the next alternative will be checked. A conflict resolver is written as

```
Resolver = "IF" '(' ... any expression ... ')' .
```

where `any` Boolean expression can be written between the parentheses. In most cases this will be a function call that returns `true` or `false`.

Thus we can resolve the LL(1) conflict from Section 2.4.5 in the following way:

```
UsingClause = "using" [ IF(IsAlias()) ident '=' ] Qualident ';'.
```

`IsAlias` is a user-defined method that reads two tokens ahead. It returns `true`, if `ident` is followed by `'='`, otherwise it returns `false`.

Conflict resolution by a multi-symbol lookahead

The generated parser remembers the most recently recognized token as well as the current lookahead token in two global variables (see also Section 3.3.4):

```
Token token;     // most recently recognized token
Token la;        // lookahead token
```

The generated scanner offers a method `Peek()` that can be used to read ahead beyond the lookahead token without removing any tokens from the input stream. When normal parsing resumes the scanner will return these tokens again.

With `Peek()` we can implement `IsAlias()` in the following way:

```
static bool IsAlias() {
  Token next = scanner.Peek();
  return la.kind == _ident && next.kind == _eql;
}
```

The conflict mentioned at the end of Section 2.4.5 can be resolved by the production

```
A = IF(FollowedByColon())
      ident (. x = 1; .) { ',' ident (. x++; .) } ':'
    | ident (. Foo(); .) { ',' ident (. Bar(); .) } ';'.
```

and the following implementation of the function `FollowedByColon()`:

```
static bool FollowedByColon() {
  Token x = la;
  while (x.kind == _comma || x.kind == _ident)
    x = scanner.Peek();
  return x.kind == _colon;
}
```

Token names. For peeking, as we have illustrated, it is convenient to be able to refer to the token kinds by names such as `_ident` or `_comma`. Coco/R generates such names for all tokens declared in the `TOKENS` section of the scanner specification. For example, if the `TOKENS` section reads:

```
TOKENS
  ident  = letter {letter | digit}.
  number = digit {digit}.
  eql    = '=';
  comma  = ','.
  colon  = ':'.
```

then Coco/R will add the following constant declarations to the parser:

```
const int _EOF = 0;
const int _ident = 1;
const int _number = 2;
const int _eql = 3;
const int _comma = 4;
const int _colon = 5;
```

These token names are preceded by an underscore in order to avoid conflicts with reserved keywords and other identifiers.

Normally the `TOKENS` section will only contain declarations for token classes like `ident` or `number`, leaving literal tokens to be declared implicitly where they appear in productions. However, if the name of a literal token is needed for peeking, it is convenient to introduce the token explicitly in this section. In the actions associated with productions of the grammar this token can then be referred to by name.

Resetting the peek position. The scanner makes sure that a sequence of `Peek()` calls will return the tokens following the lookahead token `la`. In rare situations, however, the user has to reset the peek position manually. Consider the following grammar:

```
A = (   IF (IsFirstAlternative()) ...
      | IF (IsSecondAlternative()) ...
      | ...
    ).
```

Assume that the function `IsFirstAlternative()` starts peeking and finds out that the input does not match the first alternative. So it returns `false` and the parser checks the second alternative. The function `IsSecondAlternative()` starts peeking again, but before that, it should reset the peek position to the first symbol after the lookahead token `la`. This can be done by calling `Scanner.ResetPeek()`.

```
bool IsSecondAlternative() {
   Scanner.ResetPeek();
   Token x = Scanner.Peek();   // returns the first token after the
   ...                         // lookahead token again
}
```

The peek position is reset automatically every time a regular token is recognized by `scanner.Scan()` (see Section 3.3.1).

Translation of conflict resolvers. Coco/R treats resolvers like semantic actions and simply copies them into the generated parser at the position where they appear in the grammar. For example, the production

```
UsingClause = "using" [ IF(IsAlias()) ident '=' ] Qualident ';'.
```

is translated into the equivalent of the following parsing method:

```
void UsingClause() {
  Expect(_using);
  if (IsAlias()) {
    Expect(_ident);
    Expect(_eql);
  }
  Qualident();
  Expect(_semicolon);
}
```

Conflict resolution by exploiting semantic information

A conflict resolver can base its decision not only on lookahead tokens but also on any other information. For example it could access a symbol table to find out semantic properties about a token. Consider the following LL(1) conflict between type casts and nested expressions, which can be found in many programming languages:

```
Expr   = Factor {'+' Factor}.
Factor =   '(' ident ')' Factor  /* type cast */
         | '(' Expr ')'          /* nested expression */
         | ident | number.
```

Since `Expr` can also start with an `ident`, the conflict can be resolved by checking whether this `ident` denotes a type or some other object:

```
Factor = IF (IsCast())
           '(' ident ')' Factor  /* type cast */
         | '(' Expr ')'          /* nested expression */
         | ident | number.
```

`IsCast()` looks up `ident` in the symbol table, and returns `true` if it is a type name:

```
bool IsCast() {
  Token x = scanner.Peek();
  if (x.kind == _ident) {
    object obj = symTab.Find(x.val);
    return obj != null && obj.kind == Type;
  } else return false;
}
```

Placing resolvers correctly

Coco/R checks that resolvers are placed correctly. The following rules must be obeyed:

1. If two alternatives start with the same token, the resolver must be placed in front of the first one. Otherwise it would never be executed, because the parser would always choose the first matching alternative. More precisely, a resolver must be placed at the earliest possible point where an LL(1) conflict arises.
2. A resolver may only be placed in front of an alternative that is in conflict with some other alternative. Otherwise it would be illegal.

Here is an example where resolvers are incorrectly placed:

```
A =
(   a (IF (...) b) c  // misplaced resolver. No LL(1) conflict.
  | IF (...) a b      // resolver not evaluated. Place it at first alt.
  | IF (...) b        // misplaced resolver. No LL(1) conflict
).
```

Here is how the resolvers should have been placed in this example:

```
A =
(   IF (...) a b     // resolves conflict betw. the first two alternatives
  | a c
  | b
).
```

The following example is also interesting:

```
A =
{   a
  | IF (...) b c      // resolver placed incorrectly.
} b.
```

Although the `b` in the second alternative constitutes an LL(1) conflict with the `b` after the iteration, the resolver is placed incorrectly. The reason is that it should be called only once in the parser - namely in the header of the while loop, and not both in the while header and at the beginning of the second alternative.

If correctly placed at the beginning of the iteration like this:

```
A =
{   IF (AnotherIteration())
    (   a
      | b c
    )
} b.
```

and with function `AnotherIteration()` implemented as follows:

```
bool AnotherIteration() {
  Token next = scanner.Peek();
  return la.kind == _a ||
         la.kind == _b && next.kind == _c;
}
```

Coco/R then generates code like:

```
void A() {
  while (AnotherIteration()) {
    if (la.kind == _a)
      Expect(_a);
    else if (la.kind == _b) {
      Expect(_b); Expect(_c);
    }
  }
  Expect(_b);
}
```

Remember that the resolver must be placed at the earliest possible point where the LL(1) conflict arises.

2.4.7 Syntax Error Handling

If a syntax error is detected during parsing, the generated parser reports the error and should try to recover by synchronizing the erroneous input with the grammar. While error messages are generated automatically, the user has to give certain hints in the grammar in order to enable the parser to perform appropriate synchronization.

Invalid terminal symbols. If a certain terminal symbol was expected but not found in the input the parser just reports that this symbol was expected. For example, if we had a production

```
A = a b c.
```

but the parser was presented with input like

```
a x c
```

the parser would report something like

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

Invalid alternative lists. If the lookahead symbol does not match any alternative from a list of expected alternatives in a nonterminal `A` the parser just reports that `A` was invalid. For example, if we had a production

```
A = a (b | c | d) e.
```

but the parser was presented with input like

```
a x e
```

the parser would report something like

```
file Grammar.atg : (11, 9) invalid A
```

Obviously, this error message can be improved if we turn the alternative list into a separate nonterminal symbol, i.e.:

```
A = a B e.
B = b | c | d.
```

In this case the error message would be

```
file Grammar.atg : (11, 9) invalid B
```

which is more precise.

Synchronization. After an error is reported the parser continues until it gets to a so-called *synchronization point* where it tries to synchronize the input with the grammar again. Synchronization points have to be specified by the keyword `SYNC` . They are points in the grammar where particularly *safe* tokens are expected, i.e. tokens that hardly occur anywhere else and are unlikely to be mistyped. When the parser reaches a synchronization point it skips all input until a token occurs that is expected at this point.

In many languages good candidates for synchronization points are the beginning of a statement (where keywords like `if`, `while` or `for` are expected) or the beginning of a declaration sequence (where keywords like `public`, `private` or `void` are expected). A semicolon is also a good synchronization point in a statement sequence.

The following production, for example, specifies the beginning of a statement as well as the semicolon after an assignment as synchronization points:

```
Statement =
SYNC
(   Designator '=' Expression SYNC ';'
  | "if" '(' Expression ')' Statement [ "else" Statement ]
  | "while" '(' Expression ')' Statement
  | '{' { Statement } '}'
  | ...
).
```

In the generated parser, these synchronization points lead to code as follows (written in pseudo code here):

```
void Statement() {
    while (la.kind ∉ {_EOF, _ident, _if, _while, _lbrace, ...}) {
      Report an error;
      Get next token;
    }
    if (la.kind == _ident) {
      Designator(); Expect(_eql); Expression();
      while (la.kind ∉ {_EOF, _semicolon}) {
        Report an error;
        Get next token;
      }
    } else if (la.kind == _if) { ...
    } ...
}
```

Note that the end-of-file symbol is always included in the set of synchronization symbols. This guarantees that the synchronization loop terminates at least at the end of the input.

In order to avoid a proliferation of error messages during synchronization, an error is only reported if at least two tokens have been recognized correctly since the last error.

Normally there are only a handful of synchronization points in a grammar for a real programming language. This makes error recovery cheap in parsers generated by Coco/R and does not slow down error-free parsing.

Weak tokens. Error recovery can further be improved by specifying tokens that are "weak" in a certain context. A weak token is a symbol that is often mistyped or missing such as a comma in a parameter list, which is often mistyped as a semicolon. A weak token is preceded by the keyword `WEAK`. When the parser expects a weak token, but does not find it in the input stream, it adjusts the input to the next token that is either a legal successor of the weak token or a token expected at any synchronization point (symbols expected at synchronization points are considered to be particularly "strong", so that it makes sense to never skip them).

Weak tokens are often separator symbols that occur at the beginning of an iteration. For example, if we have the productions

```
ParameterList = '(' Parameter { WEAK ',' Parameter } ')'.
Parameter = [ "ref" | "out" ] Type ident.
```

and the parser does not find a `','` or a `')'` after the first parameter, it reports an error and skips the input until it finds either a legal successor of the weak token (i.e., a legal start of `Parameter`), or a successor of the iteration (i.e. `')'`), or any symbol expected at a synchronization point (including the end-of-file symbol). The effect is that the parsing of the parameter list would not be terminated prematurely, but would get a chance to synchronize with the start of the next parameter after a possibly mistyped separator symbol.

In order to get good error recovery the user of Coco/R should perform some experiments with erroneous inputs and place `SYNC` and `WEAK` keywords appropriately to recover from the most likely errors.

2.4.8 Frame Files

The scanner and the parser are generated from template files with the names `Scanner.frame` and `Parser.frame`. Those files contain fixed code parts as well as textual markers that denote positions at which grammar-specific parts are inserted by Coco/R. In rare situations advanced users may want to modify the fixed parts of the frame files by which they can influence the behavior of the scanner and the parser to a certain degree.

The extended version of Coco/R described here can also generate a main driver class from a template file with the name `Grammar.frame` if this exists (where `Grammar` stands for the name of the goal symbol of the grammar). If not, the system can generate such a driver from a file `Driver.frame`. Such a generic driver frame file is provided with the distribution, the idea being that it will act as an example from which a more appropriate `Grammar.frame` can be derived.

3. User Guide

3.1 Installation

Both the Java version and the C# version of the Linz versions of Coco/R can be downloaded from http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/.

Both the Java version and the C# version of the extended versions of Coco/R can be downloaded from http://www.scifac.ru.ac.za/resourcekit/.

C# version. Copy the following files to appropriate directories:

`Coco.exe`	the executable
`Scanner.frame`	the frame file from which the scanner is generated
`Parser.frame`	the frame file from which the parser is generated
`Driver.frame`	the frame file from which the driver is generated, if required

Java version. Copy the following files to appropriate directories:

`Coco.jar`	an archive containing all classes of Coco/R
`Scanner.frame`	the frame file from which the scanner is generated
`Parser frame`	the frame file from which the parser is generated
`Driver.frame`	the frame file from which the driver is generated, if required

3.2 Invocation

Coco/R can be invoked from the command line as follows:

```
Under C#:     Coco fileName [ Options ]
Under Java:   java -jar Coco.jar fileName [ Options ]
```

`fileName` is the name of the file containing the Cocol/R compiler description. As a convention, compiler descriptions have the extension `.ATG` (for attributed grammar).

Options. The following options may be specified:

```
Options =
  {   "-namespace" namespaceName  /* in Java: "-package" packageName */
    | "-frames" framesDirectory
    | ( "-trace" | "-options" ) optionString
  }.
```

The user can specify a namespace (or package) to which the generated scanner and parser should belong (e.g. `at.jku.ssw.Coco`). In the extended version of Coco/R described here, if no namespace is specified the generated classes will belong to a namespace matching the name of the goal symbol of the grammar.

The `-frames` option can be used to specify the directory that contains the frame files `Scanner.frame`, `Parser.frame` and `Driver.frame` (see Section 2.4.8). If this option is missing Coco/R expects the frame files to be in the same directory as the attributed grammar.

The `-trace` (or `-options`) option allows the user to specify a string of switches (e.g. `ASX`) that cause internal data structures of Coco/R to be dumped to the file `trace.txt`. The switches are denoted by the following characters:

`A` print the states of the scanner automaton
`F` print the *first* sets and *follow* sets of all nonterminals
`G` print the syntax graph of all productions
`I` trace the computation of *first* sets
`J` list the ANY and SYNC sets used in error recovery
`P` print statistics about the run of Coco/R
`S` print the symbol table and the list of declared literals
`X` print a cross reference list of all terminals and nonterminals

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

For example, the option `$ASX` will cause the states of the automaton, the symbol table and a cross reference list to be printed to the file `trace.txt`.

Output files. Coco/R translates an attributed grammar into the following files:

- `Scanner.cs` (or `Scanner.java`) containing the classes `Scanner`, `Token` and `Buffer` .
- `Parser.cs` (or `Parser.java`) containing the classes `Parser` and `Errors`.
- `Grammar.cs` (or `Grammar.java`) containing the driver class (if requested).
- `trace.txt` containing trace output (if any).
- `listing.txt` containing a source listing with merged error message (if requested)

All files are generated in the directory that contains the attributed grammar.

3.3 Interfaces of the Generated Classes

3.3.1 Scanner

The generated scanner has the following interface:

```
public class Scanner {

  public Buffer buffer;
  public        Scanner(string sourceFile);
  public        Scanner(Stream s);
  public Token  Scan();
  public Token  Peek();
  public void   ResetPeek();
}
```

The main class of the compiler (see Section 3.4) has to create a scanner object and pass it either an input stream or the name of a file from where the tokens should be read. The scanner's input buffer is exported in the field `buffer`. It can be used to access the input text at random addresses (see Section **Error! Reference source not found.**).

The method `Scan()` is the actual scanner. The parser calls it whenever it needs the next token. Once the input is exhausted `Scan()` returns the end-of-file token, which has the token number `0`. For invalid tokens (caused by illegal token syntax or by invalid characters) `Scan()` returns a special token kind, which normally causes the parser to report an error.

`Peek()` can be used to read one or several tokens ahead without removing them from the input stream. With every call of `Scan()` (i.e. every time a token has been recognized) the peek position is set to the scan position, so that the first `Peek()` after a `Scan()` will return the first yet unscanned token. The method `ResetPeek()` can be used to reset the peek position to the scan position after several calls of `Peek()`.

3.3.2 Token

Every token returned by the scanner is an object of the following class:

```
public class Token {
  public int    kind;   // token code (EOF has the code 0)
  public string val;    // token value
  public int    pos;    // position in the source stream (starting at 0)
  public int    line;   // line number (starting at 1)
  public int    col;    // column number (starting at 0)
}
```

3.3.3 Buffer

This is an auxiliary class that is used by the scanner (and possibly by other classes) to read the source stream into a buffer and retrieve portions of it:

```
public class Buffer {
  public const char EOF = (char)256;
  public        Buffer(Stream s);
  public int    Read();
  public int    Peek();
  public int    Pos {get; set;}
  public string GetString(int beg, int end);
}
```

A buffer is initialized with the source stream s. `s.Read()` returns the next character, or `256` if the input is exhausted. `Peek()` allows the scanner to read characters ahead without consuming them. `Pos` allows the scanner to get or set the reading position, which is initially `0`. `GetString(a,b)` can be used to retrieve the text interval `[a..b[` from the input stream.

3.3.4 Parser

The generated parser has the following interface:

```
public class Parser {
  public Scanner scanner;  // the scanner of this parser
  public Errors  errors;   // the error message stream
  public Token   token;    // most recently recognized token
  public Token   la;       // lookahead token
  public         Parser(Scanner scanner, Errors errors);
  public void    Parse();
  public void    SemErr(string msg);

  public  bool   Successful()          // no errors occurred
  public void    SemError(string msg)  // same as SemErr
  public void    Warning(string msg)   // Warning message only
  public string  LexString()           // Returns token.val
  public string  LookAheadString()     // Returns la.val
}
```

The field `token` holds the most recently recognized token. It can be used in semantic actions to access the token value or the token position. The field `la` holds the lookahead token, i.e. the first token after `token`, which has not yet been recognized by the parser.

After creating a scanner and error handler, the main class of the compiler (see Section 3.4) has to create a parser object and call its method `Parse` in order to start parsing.

The method `Parse` is the actual parser. It has to be called by the main class of the compiler (see Section 3.4) after initializing the scanner.

The method SemErr(msg) can be used to report semantic errors during parsing. It calls errors.Error (see Section 3.3.5) and suppresses error messages that are too close to the position of the previous error, thus avoiding spurious error messages (see Section 2.4.7).

The remaining methods – found only in the extended version of Coco/R – provide compatibility with earlier versions of Coco/R, which were not object oriented.

3.3.5 Errors

This class is used to print error messages. Coco/R distinguishes four kinds of errors: syntax errors, semantic errors, warnings and runtime exceptions. The interface of Errors, given below, is of little direct interest, as calls to the methods involved with syntactic errors are incorporated automatically by Coco/R, while semantic errors are best dealt with by using by using the method parser.SemErr (or parser.SemError – the methods are equivalent and both are provided simply for compatibility with other versions of Coco/R

```
public class Errors {
  public int     count = 0;
  public string errMsgFormat = "file {0} : ({1}, {2}) {3}";
  public void    SynErr(int line, int col, int n);
  public void    SemErr(int line, int col, int n);
  public void    Error(int line, int col, string msg);
  public void    Exception(string msg);
  public         Errors(Scanner scanner, string fn, string dir, bool merge)
  public void    Summarize();
  public void    StoreError(int line, int col, string msg);
  public void    Warn(int line, int col, string msg);
}
```

The variable count holds the number of errors reported by SynErr, SemErr, SemError and Error.

Errors can either be reported with an error number or with an error message. Syntax errors are automatically reported by the generated parser, which calls the method SynErr. Semantic errors should be reported by the user by calling either SemErr, SemError or Error from the semantic actions of the attributed grammar.

Unless a source 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 source listing is required with merged error messages, 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.

The method Exception can be called for errors from which the compiler cannot recover. In Coco/R it is called, for example, if the frame files cannot be found or are corrupt. It prints an error message to the console and terminates the compiler.

3.4 Main Class of the Compiler

The main class of a compiler that is generated with Coco/R has also to be provided in some manner. It has to initialize the scanner, call the parser and possibly print a message about the success of the compilation. In its simplest form it might look like:

```
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");
  }
}
```

As mentioned previously, in the extended version of Coco/R a compiler driver class can be generated from an appropriate driver frame file.. 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 *Compiling with C# and Java* [Terry05].

3.5 Grammar Tests

Coco/R checks if the grammar in the compiler specification is well-formed. This includes the following tests:

- **Completeness**
  For every nonterminal symbol there must be a production. If a nonterminal `X` does not have a production Coco/R prints the message

  ```
  No production for X
  ```

- **Lack of redundancy**
  If the grammar contains productions for a nonterminal `X` that does not occur in any other productions derived from the start symbol Coco/R prints the message

  ```
  X cannot be reached
  ```

- **Derivability**
  If the grammar contains nonterminals that cannot be derived into a sequence of terminals, such as in

  ```
  X = Y ';'.
  Y = '(' X ')'.
  ```

  Coco/R prints the messages

  ```
  X cannot be derived to terminals
  Y cannot be derived to terminals
  ```

- **Lack of circularity**
  If the grammar contains circular productions, i.e. if nonterminals can be derived into themselves (directly or indirectly) such as in

  ```
  A = [ a ] B.
  B = ( C| b ).
  C = A { c }.
  ```

  Coco/R prints the messages

```
A --> B
B --> C
C --> A
```

- **Lack of ambiguity**
  If two or more tokens are declared so that they can have the same structure and thus cannot be distinguished by the scanner, as in the following example, where the input `123` could either be recognized as an `integer` or as a `float`:

  ```
  TOKENS
    integer = digit { digit }.
    float   = digit { digit } [ '.' { digit } ].
  ```

  Coco/R prints the message

  ```
  Tokens integer and float cannot be distinguished
  ```

In all these cases the compiler specification is erroneous and no scanner and parser is generated.

Warnings

There are also situations in grammars that - although legal - might lead to problems. In such cases Coco/R prints a warning, but still generates a scanner and a parser. The user should carefully check if these situations are acceptable and, if not, repair the grammar.

- **Deletable symbols**
  Sometimes, nonterminals can be derived into the empty string, such as in the following grammar:

  ```
  A = B [ a ].
  B = { b }.
  ```

  In such cases Coco/R prints the warnings

  ```
  A deletable
  B deletable
  ```

- **LL(1) conflicts**
  If two or more alternatives start with the same token such as in

  ```
  Statement = ident '=' Expression ';'
            | ident '(' Parameters ')' ';'.
  ```

  Coco/R prints the warning

  ```
  LL(1) warning in Statement: ident is the start of several alternatives
  ```

  If the start symbols and the successors of a deletable EBNF expression `{...}` or `[...]` are not disjoint, as as in

  ```
  QualId = [ id '.' ] id.
  IdList = id { ',' id } [ ',' ].
  ```

  Coco/R prints the warnings

  ```
  LL(1) warning in QualId: id is the start & successor of deletable structure
  LL(1) warning in IdList: ',' is the start & successor of deletable structure
  ```

  The resolution of LL(1) conflicts is discussed in Section 2.4.5.

4. A Sample Compiler

This section shows how to use Coco/R for building a compiler for a tiny programming language called *Taste*. Taste bears some similarities with C# or Java. It has variables of type `int` and `bool` as well as functions without parameters. It allows assignments, procedure calls, `if` and `while` statements. Integers may be read from a file and written to the console, each of them in a single line. It has arithmetic expressions (+,-,*,/) and relational expressions (==,<,>). Here is an example of a Taste program:

```
int i;  // global variable

// compute the sum of 1..i
void SumUp() {
   int sum;
   sum = 0;
   while (i > 0) { sum = sum + i; i = i - 1; }
   write sum;
}

// the program starts here
void Main() {
   read i;
   while (i > 0) {
      SumUp();
      read i;
   }
}
```

Of course Taste is too restrictive to be used as a real programming language. Its purpose is just to give you a taste of how to write a compiler with Coco/R.

The Taste compiler is a compile-and-go compiler, which means that it reads a source program and translates it into a target program which is executed (i.e. interpreted) immediately after the compilation. In order to run it type

```
Taste Test.TAS
```

The file `Test.TAS` holds the sample program shown above. This file is then compiled and immediately executed, prompting the user for the names of appropriate input (data) and output (results) files.

Classes

Figure 2 shows the classes of the compiler.

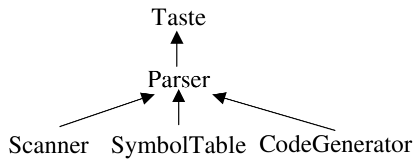

Figure 2 Classes of the Taste compiler

`Taste` is the main class. It creates the scanner and the parser and then calls the parser and the interpreter. The *symbol table* has methods to handle scopes and to store and retrieve object information. The *code generator* has methods to emit instructions. It also contains the interpreter and its data structures. The source code of all classes as well as the attributed grammar `Taste.ATG` can be found in Appendix B.

Target Code

We define an abstract stack machine for the interpretation of Taste programs. The compiler translates a source program into instructions of that machine, which are then interpreted. The machine uses the following data structures:

```
char[] code;    // object code (filled by the compiler)
int[]  globals; // data area for global variables
int[]  stack;   // stack with frames for local variables
int    top;     // stack pointer (points to next free stack slot)
int    pc;      // program counter
int    bp;      // base pointer of current frame
```

The architecture of the Taste VM is shown in Figure 3.

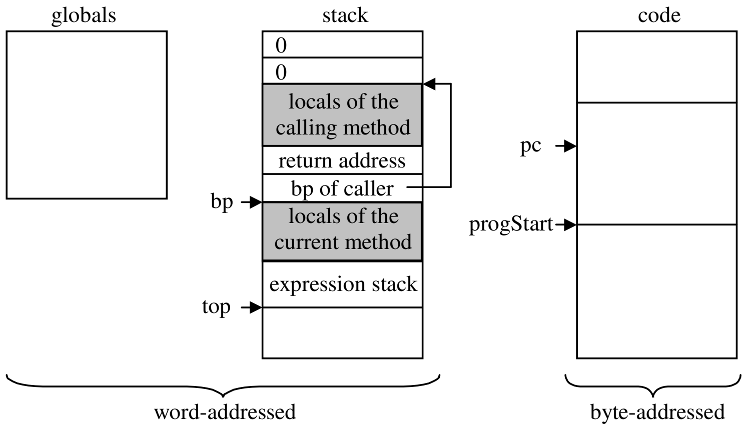

Figure 3: Data structures of the Taste VM

Global variables are stored in the word-addressed array `globals` at fixed addresses. *Local variables* are stored in stack frames that are linked with the stack frame of their caller. They are addressed with a word offset relative to the base pointer (`bp`) of the frame. At the end of the topmost stack frame there is the *expression stack* that is used for expression evaluation. After every statement the expression stack is empty.

The *machine code* is stored in the byte-addressed array `code`. The program counter `pc` points to the currently executed instruction. `progStart` is the address of the `Main` method. This is the point where the execution of the program starts.

The machine instructions are described by the following table (the initial values are: `stack[0] = 0; top = 1; bp = 0;`):

`CONST`	`n`	Load constant	`Push(n);`
`LOAD`	`a`	Load local variable	`Push(stack[bp+a]);`
`LOADG`	`a`	Load global variable	`Push(globals[a]);`
`STO`	`a`	Store local variable	`stack[bp+a]=Pop();`
`STOG`	`a`	Store global variable	`globals[a]=Pop();`
`ADD`		Add	`Push(Pop()+Pop());`
`SUB`		Subtract	`Push(-Pop()+Pop());`
`DIV`		Divide	`x=Pop(); Push(Pop()/x);`
`MUL`		Multiply	`Push(Pop()*Pop());`
`NEG`		Negate	`Push(-Pop());`
`EQU`		Compare if equal	`if (Pop()==Pop()) Push(1); else Push(0);`
`LSS`		Compare if less	`if (Pop()>Pop()) Push(1); else Push(0);`
`GTR`		Compare if greater	`if (Pop()<Pop()) Push(1); else Push(0);`
`JMP`	`a`	Jump	`pc = a;`
`FJMP`	`a`	Jump if false	`if (Pop()==0) pc=adr;`
`READ`		Read integer	`Push(ReadInt());`

`WRITE`	Write integer	`WriteLine(Pop());`
`CALL a`	Call method	`Push(pc+2); pc=a;`
`RET`	Return from method	`pc = Pop(); if (pc==0) return;`
`ENTER n`	Enter method	`Push(bp); bp=top; top+=n;`
`LEAVE`	Leave method	`top=bp; bp=Pop();`

For example, the method

```
void Foo() {
   int a, b, max;
   read a; read b;
   if (a > b) max = a; else max = b;
   write max;
}
```

is translated into the following code

```
 1: ENTER 3

 4: READ
 5: STO   0
 8: READ
 9: STO   1

12: LOAD  0
15: LOAD  1
18: GTR
19: FJMP  31

22: LOAD  0
25: STO   2
28: JMP   37

31: LOAD  1
34: STO   2

37: LOAD  2
40: WRITE

41: LEAVE
42: RET
```

Appendix B contains the source code of the following files, which can also be downloaded from http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/ (for the original Linz version) or from http://www.scifac.ru.ac.za/resourcekit / (for a version compatible with the extended version of Coco/R, but further modified to use some identifiers more like those in other case studies in the book *Compiling with C# and Java* [Terry05]).

`Taste.ATG`	the attributed grammar
`Taste.cs`	the main program
`SymTab.cs`	the symbol table
`CodeGen.cs`	the code generator and interpreter

5. Applications of Coco/R

Coco/R can be used not only to write proper compilers, but also to build many kinds of tools that process structured input data. Various people have used Coco/R for the following applications:

- An analyzer for the static complexity of programs. The analyzer evaluates the kind of operators and statements, the nesting of statements and expressions as well as the use of local and global variables to obtain a measure of the program complexity and an indication if the program is well structured.

- A cross reference generator which lists all occurrences of the objects in a program according to their scope together with information where the objects have been assigned a value and where they have been referenced.
- A pretty printer that uses the structure and the length of statements for proper indentation.
- A program that generates an index for books and reports. The index is generated from a little language that describes page numbers and the keywords occurring on those pages.
- The front end of a syntax-oriented editor. A program is translated into a tree representation which is the internal data structure of the editor.
- A program that builds a repository of symbols and their relations in a program. The repository is accessed by a case tool.
- A profiler that inserts counters and timers into the source code of a program and evaluates them after the program has been run.
- A white-box test tool that inserts counters into the source code of a program to find out which paths of the programs have been executed.
- Various compilers for special-purpose scripting languages.
- A log file analyzer that reads machine-generated information and evaluates it.

6. Acknowledgements

The author gratefully acknowledges the help of the following people, who contributed ideas and improvements to Coco/R or ported it to other programming languages:

Frankie Arzu, John Gough, Markus Löberbauer, Peter Rechenberg, Josef Templ, Pat Terry, and Albrecht Wöß.

References

[Möss90] Mössenböck, H.: A Generator for Production Quality Compilers. 3rd Intl. Workshop on Compiler Compilers (CC'90), Schwerin, LNCS 477, Springer-Verlag 1990

[Terry97] Terry, P.D.: Compilers and Compiler Generators – An Introduction Using C++. International Thomson Computer Press, 1997.

[Terry05] Terry, P.D.: Compiling with C# and Java. Pearson Addison Wesley, 2005.

[Wirth77] Wirth, N.: What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions? Communications of the ACM, November 1977

[WLM03] Wöß A., Löberbauer M., Mössenböck H.: LL(1) Conflict Resolution in a Recursive Descent Compiler Generator, Joint Modular Languages Conference (JMLC'03), Klagenfurt, 2003

A. Syntax of Cocol/R

```
Cocol =
  { ANY }               /* using clauses in C# and import clauses in Java */
  "COMPILER" ident
  { ANY }               /* global fields and methods */
  ScannerSpecification
  ParserSpecification
  "END" ident '.'.
```

```
ScannerSpecification =
                 ["IGNORECASE"]
                 ["CHARACTERS" {SetDecl}]
                 ["TOKENS"  {TokenDecl}]
                 ["NAMES"   {NameDecl}]
                 ["PRAGMAS" {PragmaDecl}]
                 {CommentDecl}
                 {WhiteSpaceDecl}.
SetDecl        = ident '=' Set.
Set            = BasicSet { ('+'|'-') BasicSet }.
BasicSet       = string | ident | Char [".." Char] | "ANY".
Char           = char | "CHR" '(' number ')'.
TokenDecl      = Symbol ['=' TokenExpr '.'].
TokenExpr      = TokenTerm {'|' TokenTerm}.
TokenTerm      = TokenFactor {TokenFactor} ["CONTEXT" '(' TokenExpr ')'].
TokenFactor    =   Symbol
                 | '(' TokenExpr ')'
                 | '[' TokenExpr ']'
                 | '{' TokenExpr '}'.
Symbol         = ident | string | char.
NameDecl       = ident "=" ( ident | string ) "." .
PragmaDecl     = TokenDecl [SemAction].
CommentDecl    = "COMMENTS" "FROM" TokenExpr "TO" TokenExpr [ "NESTED" ].
WhiteSpaceDecl = "IGNORE" Set.
```

```
ParserSpecification = "PRODUCTIONS" {Production}.
Production  = ident [Attributes] [SemAction] '=' Expression '.'.
Expression  = Term {'|' Term}.
Term        = [[Resolver] Factor {Factor}].
Factor      =   ["WEAK"] Symbol [Attributes]
              | '(' Expression ')'
              | '[' Expression ']'
              | '{' Expression '}'
              | "ANY"
              | "SYNC"
              | SemAction.
Attributes  = '<' {ANY} '>' | '<.' {ANY} '.>'.
SemAction   = "(." {ANY} ".)".
Resolver    = "IF" '(' {ANY} ')'.
```

B. Sources of the Sample Compiler

These have been modified from those in the Linz distribution, so as to correspond more closely with other case studies in *Compiling with C# and Java* [Terry05].

B.1 Taste.ATG

```
COMPILER Taste
/* Taste Compiler - based on the original by Hanspeter Moessenboeck
   This version for Coco/R for C# (Terry version)
   Modifications by Pat Terry, Rhodes University */

  public SymbolTable   tab;
  public CodeGenerator gen;

/*--------------------------------------------------------------------------*/

CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".
  digit = "0123456789".
  cr  = '\r'.
  lf  = '\n'.
  tab = '\t'.

TOKENS
  ident  = letter {letter | digit}.
  number = digit {digit}.

COMMENTS FROM "/*" TO "*/" NESTED
COMMENTS FROM "//" TO cr lf

IGNORE cr + lf + tab

PRODUCTIONS

/*------------------------------------------------------------------------*/

AddOp<out int op>
=                        (. op = CodeGenerator.BAD; .)
  ( '+'                  (. op = CodeGenerator.ADD; .)
  | '-'                  (. op = CodeGenerator.SUB; .)
  ).

/*------------------------------------------------------------------------*/

Expr<out int type>       (. int type1, op; .)
= SimExpr<out type>
  [ RelOp<out op>
    SimExpr<out type1>   (. if (type != type1) SemErr("incompatible types");
                            gen.Emit(op); type = SymbolTable.BOOL; .)
  ].

/*------------------------------------------------------------------------*/

Factor<out int type>     (. int n; Entry entry; string name; .)
=                        (. type = SymbolTable.NONE; .)
  ( Ident<out name>      (. entry = tab.Find(name); type = entry.type;
                            if (entry.kind == SymbolTable.vars) {
                              if (entry.level == 0)
                                gen.Emit(CodeGenerator.LOADG, entry.adr);
                              else
                                gen.Emit(CodeGenerator.LOAD, entry.adr);
                            } else SemErr("variable name expected"); .)
  | number               (. n = Convert.ToInt32(token.val);
                            gen.Emit(CodeGenerator.CONST, n);
                            type = SymbolTable.INT; .)
  | '-'
    Factor<out type>     (. if (type != SymbolTable.INT) {
                              SemErr("integer type expected");
                              type = SymbolTable.INT;
                            }
                            gen.Emit(CodeGenerator.NEG); .)
  | "true"               (. gen.Emit(CodeGenerator.CONST, 1);
                            type = SymbolTable.BOOL; .)
  | "false"              (. gen.Emit(CodeGenerator.CONST, 0);
                            type = SymbolTable.BOOL; .)
  ).

/*------------------------------------------------------------------------*/
```

```
Ident<out string name>
= ident                  (. name = token.val; .).

/*------------------------------------------------------------------------*/

MulOp<out int op>
=                        (. op = CodeGenerator.BAD; .)
  ( '*'                  (. op = CodeGenerator.MUL; .)
  | '/'                  (. op = CodeGenerator.DIV; .)
  ).

/*------------------------------------------------------------------------*/

ProcDecl                 (. string name; Entry entry; int adr; .)
= "void"
  Ident<out name>        (. entry = tab.NewEntry(name, SymbolTable.funcs,
SymbolTable.NONE);
                            entry.adr = gen.pc;
                            if (name.Equals("Main")) gen.progStart = gen.pc;
                            tab.OpenScope(); .)
  '(' ')'
  '{'                    (. gen.Emit(CodeGenerator.ENTER, 0);
                            adr = gen.pc - 2; .)
  { VarDecl | Stat }
  '}'                    (. gen.Emit(CodeGenerator.LEAVE);
                            gen.Emit(CodeGenerator.RET);
                            gen.Patch(adr, tab.topScope.nextAdr);
                            tab.CloseScope(); .).

/*------------------------------------------------------------------------*/

RelOp<out int op>
=                        (. op = CodeGenerator.BAD; .)
  ( "=="                 (. op = CodeGenerator.EQU; .)
  | '<'                  (. op = CodeGenerator.LSS; .)
  | '>'                  (. op = CodeGenerator.GTR; .)
  ).

/*------------------------------------------------------------------------*/

SimExpr<out int type>    (. int type1, op; .)
= Term<out type>
  { AddOp<out op>
    Term<out type1>      (. if (type != SymbolTable.INT || type1 != SymbolTable.INT)
                              SemErr("integer type expected");
                            gen.Emit(op); .)
  }.

/*------------------------------------------------------------------------*/

Stat                     (. int type; string name; Entry entry;
                            int adr, adr2, loopstart; .)
= Ident<out name>        (. entry = tab.Find(name); .)
  ( '='                  (. if (entry.kind != SymbolTable.vars)
                              SemErr("cannot assign to function"); .)
    Expr<out type> ';'
                         (. if (type != entry.type) SemErr("incompatible types");
                            if (entry.level == 0)
                              gen.Emit(CodeGenerator.STOG, entry.adr);
                            else
                              gen.Emit(CodeGenerator.STO, entry.adr); .)
  | '(' ')' ';'          (. if (entry.kind != SymbolTable.funcs)
                              SemErr(name + " is not a function");
                            gen.Emit(CodeGenerator.CALL, entry.adr); .)
  )

| "if"
  '(' Expr<out type> ')' (. if (type != SymbolTable.BOOL)
                              SemErr("boolean type expected");
                            gen.Emit(CodeGenerator.FJMP, 0); adr = gen.pc - 2; .)
  Stat
  [ "else"               (. gen.Emit(CodeGenerator.JMP, 0); adr2 = gen.pc - 2;
                            gen.Patch(adr, gen.pc);
                            adr = adr2; .)
    Stat
  ]                      (. gen.Patch(adr, gen.pc); .)
```

```
| "while"                 (. loopstart = gen.pc; .)
  '(' Expr<out type> ')' (. if (type != SymbolTable.BOOL)
                               SemErr("boolean type expected");
                             gen.Emit(CodeGenerator.FJMP, 0); adr = gen.pc - 2; .)
  Stat                    (. gen.Emit(CodeGenerator.JMP, loopstart); gen.Patch(adr,
gen.pc); .)

| "read"
  Ident<out name> ';'    (. entry = tab.Find(name);
                             if (entry.type != SymbolTable.INT)
                               SemErr("integer type expected");
                             gen.Emit(CodeGenerator.READ);
                             if (entry.level == 0)
                               gen.Emit(CodeGenerator.STOG, entry.adr);
                             else
                               gen.Emit(CodeGenerator.STO, entry.adr); .)

| "write"
  Expr<out type> ';'     (. if (type != SymbolTable.INT)
                               SemErr("integer type expected");
                             gen.Emit(CodeGenerator.WRITE); .)

| '{' { Stat | VarDecl } '}' .

/*------------------------------------------------------------------------*/

Taste
=                         (. tab.OpenScope(); .)
  { VarDecl |
    ProcDecl }            (. tab.CloseScope();
                             if (gen.progStart == -1)
                               SemErr("Main function never defined"); .).

/*------------------------------------------------------------------------*/

Term<out int type>        (. int type1, op; .)
= Factor<out type>
  { MulOp<out op>
    Factor<out type1>     (. if (type != SymbolTable.INT || type1 != SymbolTable.INT)
                               SemErr("integer type expected");
                             gen.Emit(op); .)
  }.

/*------------------------------------------------------------------------*/

Type<out int type>
=                         (. type = SymbolTable.NONE; .)
 ( "int"                  (. type = SymbolTable.INT; .)
 | "bool"                 (. type = SymbolTable.BOOL; .)
 ).

/*------------------------------------------------------------------------*/

VarDecl                   (. string name; int type; .)
= Type<out type>
  Ident<out name>         (. tab.NewEntry(name, SymbolTable.vars, type); .)
  { ',' Ident<out name>   (. tab.NewEntry(name, SymbolTable.vars, type); .)
  } ';'.

END Taste.
```

B.2 SymTab.cs (symbol table)

```
using System;

namespace Taste {

public class Entry {  // symbol table entries
  public string name;      // name of the entry
  public int    type;      // type of the entry (NONE for funcs)
  public Entry  next;      // to next entry in same scope
  public int    kind;      // vars, funcs, scope
  public int    adr;       // address in memory or start of function
  public int    level;     // nesting level; 0=global, 1=local
  public Entry  locals;    // scopes: to locally declared entries
  public int    nextAdr;   // scopes: next free address in this scope
}

public class SymbolTable {

  public const int // types
    NONE = 0, INT = 1, BOOL = 2;

  public const int // entry kinds
    vars = 0, funcs = 1, scopes = 2;

  public int curLevel;      // nesting level of current scope
  public Entry undefEntry;  // entry node for erroneous symbols
  public Entry topScope;    // topmost procedure scope

  Parser parser;

  public SymbolTable(Parser parser) {
    this.parser = parser;
    topScope = null;
    curLevel = -1;
    undefEntry = new Entry();
    undefEntry.name = "NONE"; undefEntry.type = NONE; undefEntry.kind = vars;
    undefEntry.adr = 0; undefEntry.level = 0; undefEntry.next = null;
  }

  public void OpenScope () {
  // Open a new scope and make it the current scope (topScope)
    Entry scope = new Entry();
    scope.name = ""; scope.kind = scopes;
    scope.locals = null; scope.nextAdr = 0;
    scope.next = topScope; topScope = scope;
    curLevel++;
  }

  public void CloseScope () {
  // Close the current scope
    topScope = topScope.next; curLevel--;
  }

  public Entry NewEntry (string name, int kind, int type) {
  // Create a new entry node in the current scope
    Entry p, last, entry = new Entry();
    entry.name = name; entry.kind = kind; entry.type = type;
    entry.level = curLevel;
    p = topScope.locals; last = null;
    while (p != null) {
      if (p.name.Equals(name)) parser.SemErr("name declared twice");
      last = p; p = p.next;
    }
    if (last == null) topScope.locals = entry; else last.next = entry;
    if (kind == vars) entry.adr = topScope.nextAdr++;
    return entry;
  }

  public Entry Find (string name) {
  // Search the name in all open scopes and return its entry node
    Entry entry, scope;
    scope = topScope;
    while (scope != null) {   // for all open scopes
      entry = scope.locals;
      while (entry != null) { // for all entries in this scope
        if (entry.name.Equals(name)) return entry;
        entry = entry.next;
      }
      scope = scope.next;
    }
    parser.SemErr(name + " is undeclared");
    return undefEntry;
  }

} // end SymbolTable

} // end namespace
```

B.3 CodeGen.cs (code generator)

```
using Library;
using System;
using System.IO;

namespace Taste {

public class CodeGenerator {

  // opcodes
  public const int
    ADD = 0, SUB = 1, MUL = 2, DIV = 3, EQU = 4, LSS = 5, GTR = 6, NEG = 7,
    LOAD = 8, LOADG = 9, STO = 10, STOG = 11, CONST = 12, CALL = 13, RET = 14,
    ENTER = 15, LEAVE = 16, JMP = 17, FJMP = 18, READ = 19, WRITE = 20, BAD = 99;

  string[] opcode =
    {"ADD  ", "SUB  ", "MUL  ", "DIV  ", "EQU  ", "LSS  ", "GTR  ", "NEG  ",
     "LOAD ", "LOADG", "STO  ", "STOG ", "CONST", "CALL ", "RET  ", "ENTER",
     "LEAVE", "JMP  ", "FJMP ", "READ ", "WRITE"};

  Parser parser;

  public int progStart;  // address of first instruction of main program
  public int pc;         // program counter
  bool generatingCode;
  const int memSize = 15000;
  byte[] code = new byte[memSize];

  public CodeGenerator(Parser parser) {
    this.parser = parser;
    pc = 1;
    generatingCode = true;
    progStart = -1;
  }

  //----- code generation methods -----

  public void Emit (int op) {
    if (generatingCode)
      if (pc > memSize - 4) {
        parser.SemErr("program too long");
        generatingCode = false;
      }
      else code[pc++] = (byte)op;
  }

  public void Emit (int op, int val) {
    Emit(op); Emit(val >> 8); Emit(val);
  }

  public void Patch (int adr, int val) {
    if (generatingCode) {
      code[adr]     = (byte)(val >> 8);
      code[adr + 1] = (byte)val;
    }
  }

  public void Decode() {
    int maxPc = pc;
    pc = 1;
    while (pc < maxPc) {
      int code = Next();
      Console.Write("{0,3}: {1} ", pc - 1, opcode[code]);
      switch(code) {
        case LOAD: case LOADG: case CONST: case STO: case STOG:
        case CALL: case ENTER: case JMP: case FJMP:
          Console.WriteLine(Next2()); break;
        case ADD: case SUB: case MUL: case DIV: case NEG:
        case EQU: case LSS: case GTR: case RET: case LEAVE:
        case READ: case WRITE:
          Console.WriteLine(); break;
      }
    }
  }
  // data for Interpreter
  int[] globals = new int[100];
  int[] stack = new int[100];
  int top;   // top of stack
  int bp;    // base pointer

  //----- interpreter methods -----

  int Next () {
    return code[pc++];
  }

  int Next2 () {
    int x,y;
    x = (sbyte)code[pc++]; y = code[pc++];
    return (x << 8) + y;
  }

  int Int (bool b) {
    if (b) return 1; else return 0;
  }

  void Push (int val) {
    stack[top++] = val;
  }

  int Pop() {
    return stack[--top];
  }

  public void Interpret () {
    int val;
    Console.Write("\nData file [STDIN] ? ");
    InFile data = new InFile(Console.ReadLine());
    Console.Write("\nResults file [STDOUT] ? ");
    OutFile results = new OutFile(Console.ReadLine());
    pc = progStart; stack[0] = 0; top = 1; bp = 0;
    for (;;)
      switch (Next()) {
        case CONST: Push(Next2()); break;
        case LOAD:  Push(stack[bp + Next2()]); break;
        case LOADG: Push(globals[Next2()]); break;
        case STO:   stack[bp + Next2()] = Pop(); break;
        case STOG:  globals[Next2()] = Pop(); break;
        case ADD:   Push(Pop() + Pop()); break;
        case SUB:   Push(-Pop() + Pop()); break;
        case DIV:   val = Pop();
                    if (val != 0) { Push(Pop() / val); break; }
                    else {
                      Console.WriteLine("Division by zero");
                      results.Close(); System.Environment.Exit(1); break;
                    }
        case MUL:   Push(Pop() * Pop()); break;
        case NEG:   Push(-Pop()); break;
        case EQU:   Push(Int(Pop() == Pop())); break;
        case LSS:   Push(Int(Pop() > Pop())); break;
        case GTR:   Push(Int(Pop() < Pop())); break;
        case JMP:   pc = Next2(); break;
        case FJMP:  val = Next2(); if (Pop() == 0) pc = val; break;
        case READ:  val = data.ReadInt(); Push(val); break;
        case WRITE: results.WriteLine(Pop()); break;
        case CALL:  Push(pc+2); pc = Next2(); break;
        case RET:   pc = Pop();
                    if (pc == 0) {
                      results.Close(); return;
                    }
                    break;
        case ENTER: Push(bp); bp = top; top = top + Next2(); break;
        case LEAVE: top = bp; bp = Pop(); break;
        default:    Console.WriteLine("Illegal Instruction");
                    results.Close(); System.Environment.Exit(1); break;
      }
  }

} // end CodeGenerator

} // end namespace
```

B.4 Taste.cs (main program)

```
using System;

namespace Taste {

class Taste {

  public static void Main (string[] args) {
    bool mergeErrors = false;
    string inputName = null;

    for (int i = 0; i < args.Length; i++) {
      if (args[i].ToLower().Equals("-l")) mergeErrors = true;
      else inputName = args[i];
    }
    if (inputName == null) {
      Console.WriteLine("No input file specified");
      System.Environment.Exit(1);
    }

    int pos = inputName.LastIndexOf('/');
    if (pos < 0) pos = inputName.LastIndexOf('\\');
    string dir = inputName.Substring(0, pos+1);
    Scanner scanner = new Scanner(inputName);
    Errors errors = new Errors(scanner, inputName, dir, mergeErrors);
    Parser parser = new Parser(scanner, errors);
    parser.tab = new SymbolTable(parser);
    parser.gen = new CodeGenerator(parser);
    parser.Parse();
    errors.Summarize();
    if (errors.count == 0) {
      parser.gen.Decode();
      parser.gen.Interpret();
    }
  }

}

} // end namespace
```