7. Building Compilers with Coco/R

7.1 Overview

7.2 Scanner Specification

7.3 Parser Specification

7.4 Error Handling

7.5 LL(1) Conflicts

7.6 Example
Coco/R - Compiler Compiler / Recursive Descent

Generates a scanner and a parser from an ATG

- attributed grammar → Coco/R → parser
- scanner
- user-supplied classes (e.g. symbol table)

Scanner: DFA
Parser: Recursive Descent

Origin: 1980, built at the University of Linz
Current versions: for Java, C#, C++, VB.NET, Delphi, Modula-2, Visual Basic, Oberon, ...

Open source: http://ssw.jku.at/Coco/
Similar tools: Lex/Yacc, JavaCC, ANTLR, ...

### Example: Compiler for Arithmetic Expressions

#### Compiler Calc

**Characters**
- digit = '0' .. '9'.

**Tokens**
- number = digit {digit}.

**Comments**
- FROM "//" TO cr lf
- FROM "/*" TO "*/" NESTED
- IGNORE 't' + 'r' + 'n'

**Productions**

```plaintext
Calc (. int x; .) = "CALC" Expr<out x> (. System.out.println(x); .) .

Expr <out int x> (. int y; .) = Term<out x>
  { '+' Term<out y> (. x = x + y; .) }

Term <out int x> (. int y; .) = Factor<out x>
  { '*' Factor<out y> (. x = x * y; .) }

Factor <out int x> (. int y; .) = number (. x = Integer.parseInt(t.val); .)
| (' Expr<out x> ').
```

**END Calc.**

---

**Scanner specification**

**Parser specification**
Structure of a Compiler Description

- "COMPILER" ident
- ScannerSpecification
- ParserSpecification
- "END" ident "."

ident denotes the start symbol of the grammar (i.e. the topmost nonterminal symbol)
7. Building Generators with Coco/R

7.1 Overview

7.2 Scanner Specification

7.3 Parser Specification

7.4 Error Handling

7.5 LL(1) Conflicts

7.6 Example
Structure of a Scanner Specification

ScannerSpecification =
"IGNORECASE"
"CHARACTERS" {SetDecl}
"TOKENS" {TokenDecl}
"PRAGMAS" {PragmaDecl}
{CommentDecl}
{WhiteSpaceDecl}.

Should the generated compiler be case-sensitive?

Which character sets are used in the token declarations?

Here one has to declare all structured tokens (i.e. terminal symbols) of the grammar

Pragmas are tokens which are not part of the grammar

Here one can declare one or several kinds of comments for the language to be compiled

Which characters should be ignored (e.g. \t, \n, \r)?
Character Sets

Example

<table>
<thead>
<tr>
<th>CHARACTERS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>digit</td>
<td>= &quot;0123456789&quot;. the set of all digits</td>
</tr>
<tr>
<td>hexDigit</td>
<td>= digit + &quot;ABCDEF&quot;. the set of all hexadecimal digits</td>
</tr>
<tr>
<td>letter</td>
<td>= 'A' .. 'Z'. the set of all upper-case letters</td>
</tr>
</tbody>
</table>
| eol                | = '
'. the end-of-line character                      |
| noDigit            | = ANY - digit. any character that is not a digit      |

Valid escape sequences in character constants and strings

\ 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 character value

Coco/R allows Unicode (UTF-8)
Token Declarations

Define the structure of token classes (e.g. ident, number, ...)
Literals such as "while" or ">=" don't have to be declared

Example

<table>
<thead>
<tr>
<th>TOKEs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ident</td>
<td>= letter {letter</td>
</tr>
<tr>
<td>number</td>
<td>= digit {digit} |</td>
</tr>
<tr>
<td></td>
<td>&quot;0x&quot; hexDigit hexDigit hexDigit hexDigit.</td>
</tr>
<tr>
<td>float</td>
<td>= /digit {digit} '.' digit {digit} ['E' ['+'</td>
</tr>
</tbody>
</table>

• Right-hand side must be a regular EBNF expression
• Names on the right-hand side denote character sets

no problem if alternatives start with the same character
Pragmas

Special tokens (e.g. compiler options)
- can occur anywhere in the input
- are not part of the grammar
- must be semantically processed

Example

Compiler options (e.g., $AB) that can occur anywhere in the code

Whenever an option (e.g. $ABC) occurs in the input, this semantic action is executed.

Typical applications
- compiler options
- preprocessor commands
- comment processing
- end-of-line processing
Comment

Described in a special section because

- nested comments cannot be described with regular grammars
- must be ignored by the parser

Example

COMMENTS FROM "/*" TO "*/" NESTED
COMMENTS FROM "//" TO "\n"
White Space and Case Sensitivity

White space

```plaintext
<table>
<thead>
<tr>
<th>IGNORE</th>
<th>\t + \r + \n</th>
</tr>
</thead>
</table>
| character set                               blanks are ignored by default
```

Case sensitivity

Compilers generated by Coco/R are case-sensitive by default.

Can be made case-insensitive by the keyword `IGNORECASE`.

```plaintext
COMPIlER Sample
IGNORECASE
CHARACTERS
  hexDigit = digit + 'a'..'f'.
...
TOKEnS
  number = "0x" hexDigit hexDigit hexDigit hexDigit hexDigit.
...
PRODUCTIONS
  WhileStat = "while" '(' Expr ')' Stat.
...
END Sample.
```

Will recognize
- 0x00ff, 0X00ff, 0X00FF as a number
- while, While, WHILE as a keyword

Token values returned to the parser retain their original casing.
**Interface of the Generated Scanner**

```java
public class Scanner {
    public Buffer buffer;
    public Scanner (String fileName);
    public Scanner (InputStream s);
    public Token Scan();
    public Token Peek();
    public void ResetPeek();
}
```

- **Scanner**
  - `public class Scanner {` (declaration of class)
  - `public Buffer buffer;` (field declaration)
  - `public Scanner (String fileName);` (constructor)
  - `public Scanner (InputStream s);` (constructor)
  - `public Token Scan();` (main method: returns a token upon every call)
  - `public Token Peek();` (reads ahead from the current scanner position without removing tokens from the input stream)
  - `public void ResetPeek();` (resets peeking to the current scanner position)

```java
public class Token {
    public int kind; // token kind (i.e. token number)
    public int pos;  // token position in the source text (starting at 0)
    public int col;  // token column (starting at 1)
    public int line; // token line (starting at 1)
    public String val; // token value
}
```

- **Token**
  - `public class Token {` (declaration of class)
  - `public int kind;` (token kind)
  - `public int pos;` (token position)
  - `public int col;` (token column)
  - `public int line;` (token line)
  - `public String val;` (token value)
7. Building Generators with Coco/R

7.1 Overview
7.2 Scanner Specification
7.3 **Parser Specification**
7.4 Error Handling
7.5 LL(1) Conflicts
7.6 Example
Productions

- Can occur in any order
- There must be exactly 1 production for every nonterminal
- There must be a production for the start symbol (the grammar name)

Example

COMPILER Expr
...

PRODUCTIONS

Expr = SimExpr [RelOp SimExpr].
SimExpr = Term {AddOp Term}.
Term = Factor {Mulop Factor}.
Factor = ident | number | "-" Factor | "true" | "false".
RelOp = "==" | "<" | ">".
AddOp = "+" | "-".
MulOp = "*" | "/".
END Expr.

Arbitrary context-free grammar in EBNF
**Semantic Actions**

**Arbitrary Java code between (. and .)**

```java
IdentList = ident (. n = 1; .)
{ ',' ident (. n++; .)
}
(. System.out.println(n); .)
```

Semantic actions are copied to the generated parser without being checked by Coco/R

**Global semantic declarations**

```java
import java.io.*;
COMPILER Sample
    FileWriter w;
    void Open(string path) {
        w = new FileWriter(path);
        ...
    }
...
PRODUCTIONS
    Sample = ...
    (. Open("in.txt"); .)
END Sample.
```

import of classes from other packages

global semantic declarations
(become fields and methods of the parser)

semantic actions can access global declarations as well as imported classes
**Attributes**

**For terminal symbols**
- terminal symbols do not have explicit attributes
- their values can be accessed in sem. actions using the following variables declared in the parser
  - `Token t;` the most recently recognized token
  - `Token la;` the lookahead token (not yet recognized)

**Example**
```
Factor <out int x> = number    (. x = Integer.parseInt(t.val); .)
```

```java
class Token {
    int kind;     // token code
    String val;   // token value
    int pos;      // token position in the source text (starting at 0)
    int line;     // token line (starting at 1)
    int col;      // token column (starting at 1)
}
```

**For nonterminal symbols**
- NTS can have any number of input attributes
- NTS can have at most one output attribute (must be the first in the attribute list)

**formal attr.:**
```
A <int x, char c> = ... .
```

**actual attr.:**
```
... A <y, 'a'> ...
```

**formal attr.:**
```
B <out int x, int y> = ... .
```

**actual attr.:**
```
... B <out z, 3> ...
```
Productions are Translated to Parsing Methods

Production

Expr<out int n> (. int n1; .) = Term<out n>
{ '+'
   Term<out n1> (. n = n + n1; .)
}.

Resulting parsing method

int Expr() {
    int n;
    int n1;
    n = Term();
    while (la.kind == 3) {
        Get();
        n1 = Term();
        n = n + n1;
    }
    return n;
}

Attributes => parameters or return values
Semantic actions => embedded in parser code
The symbol ANY

Denotes any token that is not an alternative of this ANY symbol

Example: counting the number of occurrences of int

```
Type
= "int" (. intCounter++; .)
| ANY. ← any token except "int"
```

Example: computing the length of a block

```
Block<out int len>
= "{" (. int beg = t.pos + 1; .
| { ANY } ← any token except "}""                                 |
| "}" (. len = t.pos - beg; .).                                    |
```

Example: counting statements in a block

```
Block<out int stmts>
= "{" (. int n; .)
| { ".;" (. stmts = 0; .)
| Block<out n> (. stmts++; .)
| ANY } ← any token except "{" , "}" or ";;"
| "}" .
```
Frame Files

Scanner.frame snippet

```java
public class Scanner {
    static final char EOL = '\n';
    static final int eofSym = 0;

    -->declarations
    ...
    public Scanner (InputStream s) {
        buffer = new Buffer(s);
        Init();
    }
    void Init () {
        pos = -1; line = 1; ...
    }
    --->initialization
    ...
}
```

- Coco/R inserts generated parts at positions marked by "-->..."
- Users can edit the frame files for adapting the generated scanner and parser to their needs
- Frame files are expected to be in the same directory as the compiler specification (e.g. Sample.atg)
Interface of the Generated Parser

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

Parser invocation in the main program

public class MyCompiler {

    public static void main(String[] arg) {
        Scanner scanner = new Scanner(arg[0]);
        Parser parser = new Parser(scanner);
        parser.Parse();
        System.out.println(parser.errors.count + " errors detected");
    }
}
7. Building Generators with Coco/R

7.1 Overview
7.2 Scanner Specification
7.3 Parser Specification
7.4 Error Handling
7.5 LL(1) Conflicts
7.6 Example
**Syntax Error Handling**

Syntax error messages are generated automatically

For invalid terminal symbols

*production*  
\[ S = a \ b \ c. \]

*input*  
\[ a \ x \ c \]

*error message*  
`-- line ... col ...: b expected`

For invalid alternative lists

*production*  
\[ S = a (b \ | \ c \ | \ d) \ e. \]

*input*  
\[ a \ x \ e \]

*error message*  
`-- line ... col ...: invalid S`

Error message can be improved by rewriting the production

*productions*  
\[ S = a \ T \ e. \]
\[ T = b \ | \ c \ | \ d. \]

*input*  
\[ a \ x \ e \]

*error message*  
`-- line ... col ...: invalid T`
Syntax Error Recovery

The user must specify synchronization points where the parser should recover

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

What happens if an error is detected?

- parser reports the error
- parser continues to the next synchronization point
- parser skips input symbols until it finds one that is expected at the synchronization point

```
while (la.kind is not accepted here) {
    la = scanner.Scan();
}
```

What are good synchronization points?

Points in the grammar where particularly "safe" tokens are expected

- start of a statement: if, while, do, ...
- start of a declaration: public, static, void, ...
- in front of a semicolon
**Semantic Error Handling**

Must be done in semantic actions

```java
Expr<out Type type> = Term<out type>
{ '+' Term<out type1> ( if (type != type1) SemErr("incompatible types"); )
}
```

*SemErr method in the parser*

```java
void SemErr (String msg) {
    ...
    errors.SemErr(t.line, t.col, msg);
    ...
}
```
Errors Class

Coco/R generates a class for error message reporting

```java
public class Errors {

    public int count = 0;                // number of errors detected
    public PrintStream errorStream = System.out; // error message stream
    public String errMsgFormat = "-- line {0} col {1}: {2}"; // 0=line, 1=column, 2=text

    // called by the programmer (via Parser.SemErr) to report semantic errors
    public void SemErr (int line, int col, String msg) {
        printMsg(line, col, msg);
        count++;
    }

    // called automatically by the parser to report syntax errors
    public void SynErr (int line, int col, int n) {
        String msg;
        switch (n) {
            case 0: msg = "..."; break;
            case 1: msg = "..."; break;
            ...
        }
        printMsg(line, col, msg);
        count++;
    }

    ...

    ...
}
```
7. Building Generators with Coco/R

7.1 Overview
7.2 Scanner Specification
7.3 Parser Specification
7.4 Error Handling
7.5 LL(1) Conflicts
7.6 Example
Coco/R finds LL(1) Conflicts automatically

Example

... PRODUCTIONS
  Sample = \{Statement\}.
  Statement = Qualident '=' number ';'
                 | Call
                 | "if" '(' ident ')' Statement ["else" Statement].
  Call = ident '(' ')' ';'.
  Qualident = [ident '.'] ident.
...

Coco/R produces the following warnings

>coco Sample.atg
Coco/R (Sep 19, 2015)
checking
  Sample deletable
  LL1 warning in Statement: ident is start of several alternatives
  LL1 warning in Statement: "else" is start & successor of deletable structure
  LL1 warning in Qualident: ident is start & successor of deletable structure
parser + scanner generated
0 errors detected
Conflict Resolution by Multi-symbol Lookahead

\[
A = \text{ident} (. \ x = 1; .) \ {','} \ \text{ident} (. \ x++; .) \ } ':' \\
| \ \text{ident} (. \ \text{Foo}(); .) \ {','} \ \text{ident} (. \ \text{Bar}(); .) \ } ':'
\]

\( LL(1) \) conflict

Resolution

\[
A = \text{IF} (\text{FollowedByColon}()) \\
\quad \text{ident} (. \ x = 1; .) \ {','} \ \text{ident} (. \ x++; .) \ } ':' \\
| \ \text{ident} (. \ \text{Foo}(); .) \ {','} \ \text{ident} (. \ \text{Bar}(); .) \ } ':'
\]

Resolution method

boolean \text{FollowedByColon}() \{
    \text{Token} \ x = \text{la}; \\
    \text{while} (x.\text{kind} == _\text{ident} || x.\text{kind} == _\text{comma}) \{
        x = \text{scanner.Peek}(); \\
    \} \\
    \text{return} x.\text{kind} == _\text{colon}; \\
\}

TOKENS
ident = \text{letter \{letter | digit\} .} \\
comma = ",".

... static final int
_ident = 17, \\
_comma = 18,
...
Conflict Resolution by Semantic Information

\[
\text{Factor} = \left(\begin{array}{l}
'(\text{ident} ')\text{Factor} \\
'(\text{Expr} ')\text{Factor} \\
\text{ident} \\
\text{number}
\end{array}\right)
\]

LL(1) conflict

Resolution

\[
\text{Resolution method}
\]

```java
boolean IsCast() {
    Token next = scanner.Peek();
    if (la.kind == _lpar && next.kind == _ident) {
        Obj obj = Tab.find(next.val);
        return obj.kind == Obj.Type;
    } else return false;
}
```

returns true if '(' is followed by a type name
7. Building Generators with Coco/R

7.1 Overview
7.2 Scanner Specification
7.3 Parser Specification
7.4 Error Handling
7.5 LL(1) Conflicts
7.6 Example
Example: Query Form Generator

**Input:** A domain-specific language for describing query forms

<table>
<thead>
<tr>
<th>RADIO</th>
<th>&quot;How did you like this course?&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(&quot;very much&quot;, &quot;much&quot;, &quot;somewhat&quot;, &quot;not so much&quot;, &quot;not at all&quot;)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHECKBOX</th>
<th>&quot;What is the field of your study?&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(&quot;Computer Science&quot;, &quot;Mathematics&quot;, &quot;Physics&quot;)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TEXTBOX</th>
<th>&quot;What should be improved?&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

**Output:** HTML query form

To do

1. Describe the input by a grammar
2. Define attributes for the symbols
3. Define semantic routines to be called
4. Write an ATG
Input Grammar

QueryForm = {Query}.
Query = "RADIO" Caption Values
| "CHECKBOX" Caption Values
| "TEXTBOX" Caption.
Values = '(' string {',' string} ')'.
Caption = string.

Attributes

• Caption returns a string Caption<out String s>
• Values returns a list of strings Values<out ArrayList list>

Semantic routines

• printHeader()
• printFooter()
• printRadio(caption, values)
• printCheckbox(caption, values)
• printTextbox(caption)

implemented in a class HtmlGenerator

RADIO "How did you like this course?"
("very much", "much", "somewhat",
"not so much", "not at all")

CHECKBOX "What is the field of your study?"
("Computer Science", "Mathematics", "Physics")

TEXTBOX "What should be improved?"
COMPILER QueryForm
CHARACTERS
  noQuote = ANY - "".
TOKENS
  string = "" {noQuote} "".
COMMENTS
  FROM "//" TO "\n"
IGNORE \t + \r + \n
...
import java.util.ArrayList;
COMPILER QueryForm
    HtmlGenerator html;
...

PRODUCTIONS
QueryForm =
    { Query }

//----------------------------------------------------------------------
Query = 
    "RADIO" Caption<out caption> Values<out values>
    (. html.printRadio(caption, values); .)
| "CHECKBOX" Caption<out caption> Values<out values>
    (. html.printCheckbox(caption, values); .)
| "TEXTBOX" Caption<out caption>
    (. html.printTextbox(caption); .)

//----------------------------------------------------------------------
Caption<out String s> = StringVal<out s>.

//----------------------------------------------------------------------
Values<out ArrayList values> = 
    '(' StringVal<out s> 
    { ', ' StringVal<out s> 
      } 
    ')'.

//----------------------------------------------------------------------
StringVal<out String s> = string 
    (. s = t.val.substring(1, t.val.length()-1); .)

END QueryFormGenerator.
import java.io.*;
import java.util.ArrayList;

class HtmlGenerator {
    PrintStream s;
    int itemNo = 0;

    public HtmlGenerator(String fileName) throws FileNotFoundException {
        s = new PrintStream(fileName);
    }

    public void printHeader() {
        s.println("<html>\n        <head><title>Query Form</title></head>\n        <body>();
        s.println("  <form>\n        s.println("    <html>\n        s.println("      </form>\n        s.close();
    }

    public void printFooter() {
        s.println("  </form>\n        s.println("</body>\n        s.println("</html>\n        s.close();
    }
}
public void **printRadio** (String caption, ArrayList values) {
    s.println(caption + "<br>");
    for (Object val: values) {
        s.println("<input type='radio' name='Q" + itemNo + '" value='" + val + "'>" + val + "<br>");
    }
    itemNo++; s.println("<br>");
}

public void **printCheckbox** (String caption, ArrayList values) {
    s.println(caption + "<br>");
    for (Object val: values) {
        s.println("<input type='checkbox' name='Q" + itemNo + '" value='" + val + "'>" + val + "<br>");
    }
    itemNo++; s.println("<br>");
}

public void **printTextbox** (String caption) {
    s.println(caption + "<br>");
    s.println("<textarea name='Q" + itemNo + '" cols='50' rows='3'>" + itemNo + "</textarea><br>");
    itemNo++; s.println("<br>");
}
Main Program

Tasks
• Read command-line arguments
• Create and initialize scanner and parser
• Start the parser

```java
import java.io.*;
class MakeQueryForm {
    public static void main(String[] args) {
        String inFileName = args[0];
        String outFileName = args[1];
        Scanner scanner = new Scanner(inFileName);
        Parser parser = new Parser(scanner);
        try {
            parser.html = new HtmlGenerator(outFileName);
            parser.Parse();
            System.out.println(parser.errors.count + " errors detected");
        } catch (FileNotFoundException e) {
            System.out.println("-- cannot create file " + outFileName);
        }
    }
}
```
Putting it All Together

Run Coco/R

```
java -jar Coco.jar QueryForm.ATG
```

Scanner.java, Parser.java

Compile everything

```
javac Scanner.java Parser.java HtmlGenerator.java MakeQueryForm.java
```

Run the Query Form Generator

```
java MakeQueryForm input.txt output.html
```
Summary

Compiler-generating tools like Coco/R can always be applied if

- some input is to be transformed into some output
- the input is syntactically structured

Typical applications

- static program analyzers
- metrics tools for source code
- source code instrumentation
- domain-specific languages
- log file analyzers
- data stream processing
- ...