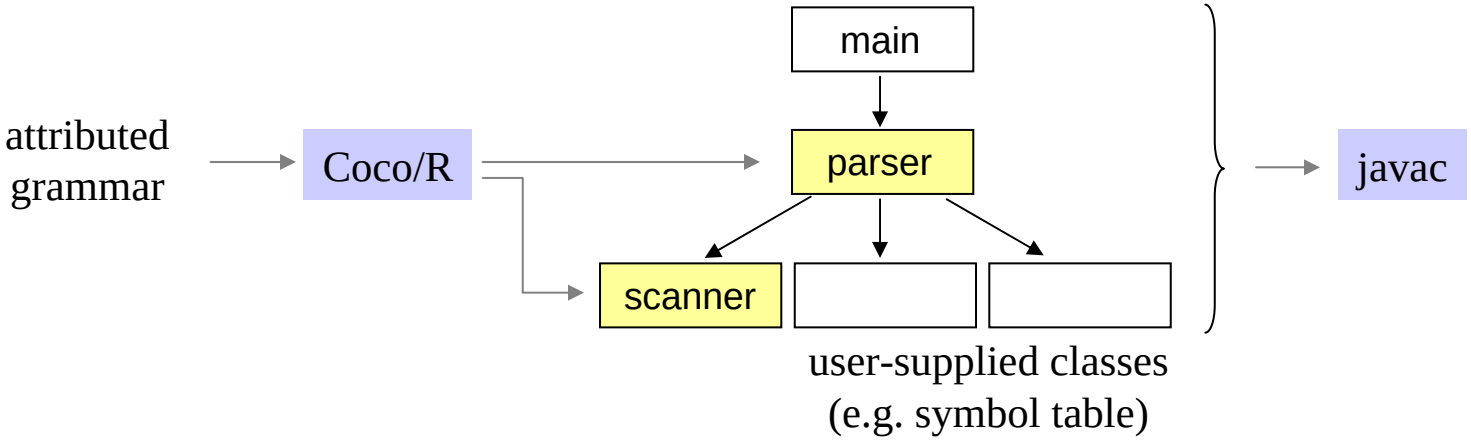
7. Building Generators with Coco/R

Coco/R - Compiler Compiler / Recursive Descent

Generates a scanner and a parser from an ATG



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, ...

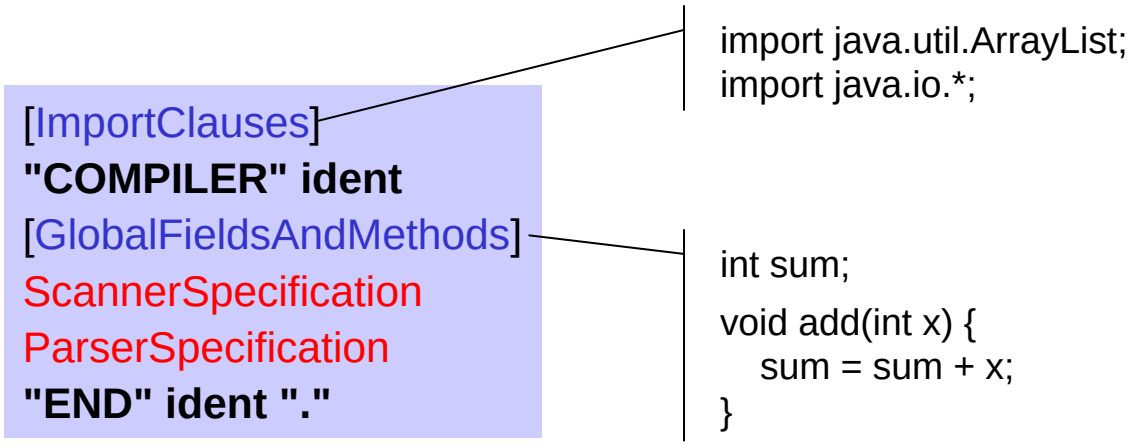
Example: Compiler for Arithmetic Expressions


```
COMPILER Calc
CHARACTERS
   digit = '0' .. '9'.
TOKENS
   number = digit {digit}.
COMMENTS FROM "//" TO cr lf
COMMENTS FROM "/*" TO "*/" NESTED
IGNORE '\t' + '\r' + '\n'
PRODUCTIONS
   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>
   =  number                     (. x = Integer.parseInt(t.val); .)
   |  '(' Expr<out x> ')'.
END Calc.
```

Scanner specification

Parser specification

Structure of a Compiler Description



ident denotes the start symbol of the grammar (i.e. the topmost nonterminal symbol)

7. Building Generators with Coco/R

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

```
CHARACTERS
   digit    = "0123456789".
   hexDigit = digit + "ABCDEF".
   letter   = 'A' .. 'Z'.
   eol      = '\n'.
   noDigit  = ANY - digit.
```

the set of all digits
the set of all hexadecimal digits
the set of all upper-case letters
the end-of-line character
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

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

no problem if alternatives start
with the same character

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

Pragmas

Special tokens (e.g. compiler options)

- can occur anywhere in the input
- are not part of the grammar
- must be semantically processed

Example

```
PRAGMAS
   option = '$' {letter}.  (.  for (int i = 1; i < la.val.length(); i++) {
                                  char ch = la.val.charAt(i);
                                  if (ch == 'A') ...
                                  else if (ch == 'B') ...
                                  ...
                               } .)
```

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

Comments

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 "\r\n"
```

White Space and Case Sensitivity

White space

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

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`

```
COMPILER Sample
IGNORECASE
CHARACTERS
   hexDigit = digit + 'a'..'f'.
   ...
TOKENS
   number = "0x" 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


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

main method: returns a token upon every call

reads ahead from the current scanner position without removing tokens from the input stream

resets peeking to the current scanner position

```
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
}
```

7. Building Generators with Coco/R

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 .)

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

local semantic declaration

semantic action

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

Global semantic declarations

```
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); .)
```

```
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

  formal attr.: `A <int x, char c> = ... .` actual attr.: `... A <y, 'a'> ...`

- NTS can have at most one output attribute (must be the first in the attribute list)

  `B <out int x, int y> = ... .` `... 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.
```

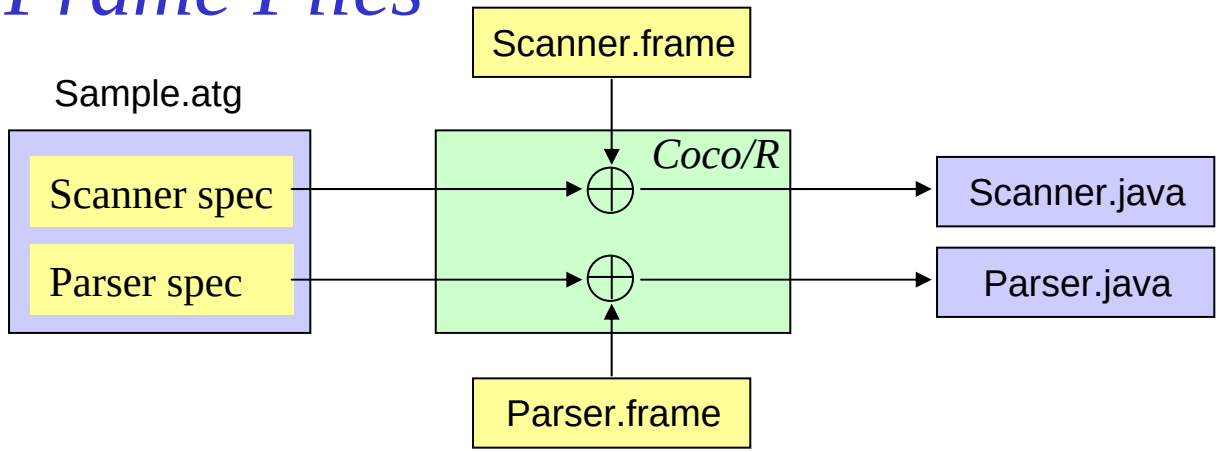
any token except "int"

Example: computing the length of a semantic action

```
SemAction<out int len>
=  "(."          (. int beg = t.pos + 2; .)
   { ANY }
   ".)"          (. len = t.pos - beg; .) .
```

any token except ".)"

Frame Files



Scanner.frame snippet

```
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();
      Console.WriteLine(parser.errors.count + " errors detected");
   }
}
```

7. Building Generators with Coco/R

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

synchronization points

```
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

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

***SemErr* method in the parser**

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

Errors Class

Coco/R generates a class for error message reporting

```
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++;
   }
   ...
}
```

syntax error messages generated by Coco/R

7. Building Generators with Coco/R

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, 2008)
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 = ident (. x = 1; .) {',' ident (. x++; .) } ':'
  | ident (. Foo(); .) {',' ident (. Bar(); .) } ';'.
```

LL(1) conflict

Resolution

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

Resolution method

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

```
TOKENS
   ident = letter {letter | digit} .
   comma = ','.
   ...
```

```
static final int
   _ident = 17,
   _comma = 18,
   ...
```

Conflict Resolution by Semantic Information


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

LL(1) conflict

Resolution

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

Resolution method

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

returns true if '(' is followed by a type name

7. Building Generators with Coco/R

Example: Query Form Generator

Input: A domain-specific language for describing query forms

```
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?"
...
```

Output: HTML query form

How did you like this course?
- ○ very much
- ○ much
- ○ somewhat
- ○ not so much
- ○ not at all

What is the field of your study?
- ☐ Computer Science
- ☐ Mathematics
- ☐ Physics

What should be improved?

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.
```

```
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?"
```

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

Scanner Specification


```
COMPILER QueryForm

CHARACTERS
   noQuote = ANY - "".
   tab = '\t'.
   cr = '\r'.
   lf = '\n'.

TOKENS
   string = "" {noQuote} "".

COMMENTS
   FROM "//" TO cr lf

IGNORE tab + cr + lf

...

END QueryForm.
```

Parser Specification

```
import java.util.ArrayList;
COMPILER QueryForm
   HtmlGenerator html;
   ...
PRODUCTIONS
QueryForm =                          (. html.printHeader(); .)
   { Query }                         (. html.printFooter(); .) .
//-----------------------------------------------------------------------------------
Query                                (. String caption; ArrayList values; .)
=  "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>         (. String s; .)
=  '(' StringVal<out s>              (. values = new ArrayList(); values.add(s); .)
   { ',' StringVal<out s>            (. values.add(s); .)
   }
   ')'.
//-----------------------------------------------------------------------------------
StringVal<out String s>
=  string                            (. s = t.val.substring(1, t.val.length()-1); .) .
END QueryFormGenerator.
```

Class HtmlGenerator


```
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>");
      s.println("<head><title>Query Form</title></head>");
      s.println("<body>");
      s.println("  <form>");
   }

   public void printFooter() {
      s.println("  </form>");
      s.println("</body>");
      s.println("</html>");
      s.close();
   }
   ...
```

Class HtmlGenerator (continued)


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

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

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

```
<input type='radio' name='Q0'
    value='very much'>very much<br>
```

```
<input type='checkbox' name='Q1'
    value='Mathematics'>Mathematics<br>
```

```
<textarea name='Q2' cols='50' rows='3'>
</textarea><br>
```

Main Program

Tasks

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

```
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
- metric tools for source code
- source code instrumentation
- domain-specific languages
- log file analyzers
- data stream processing
- ...