Using Coco/R for Compiler Construction

CSE 530 Design of Compilers

Summer 2025

Dar-jen Chang

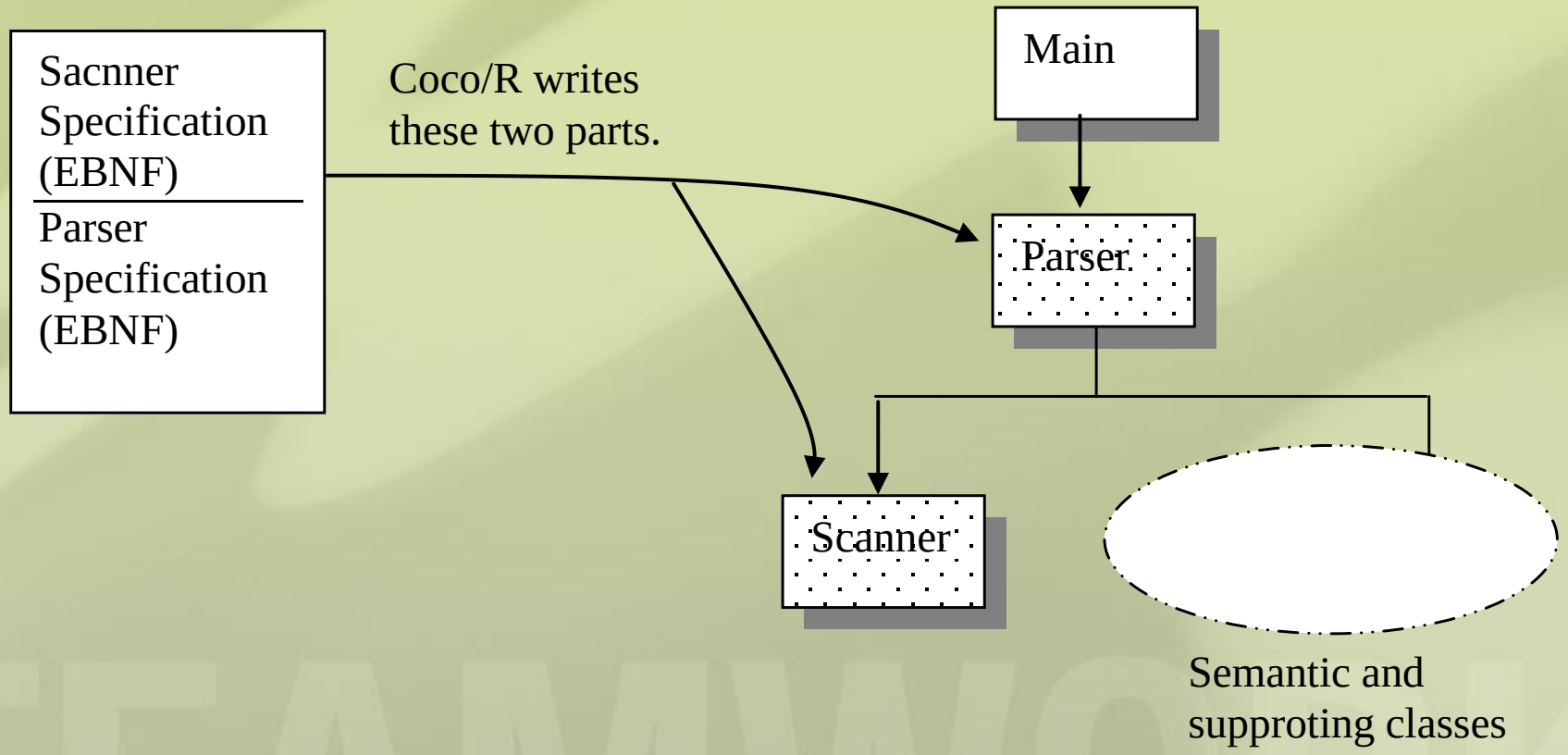
Coco/R - Compiler Compiler / Recursive Descent

Coco/R User Manual

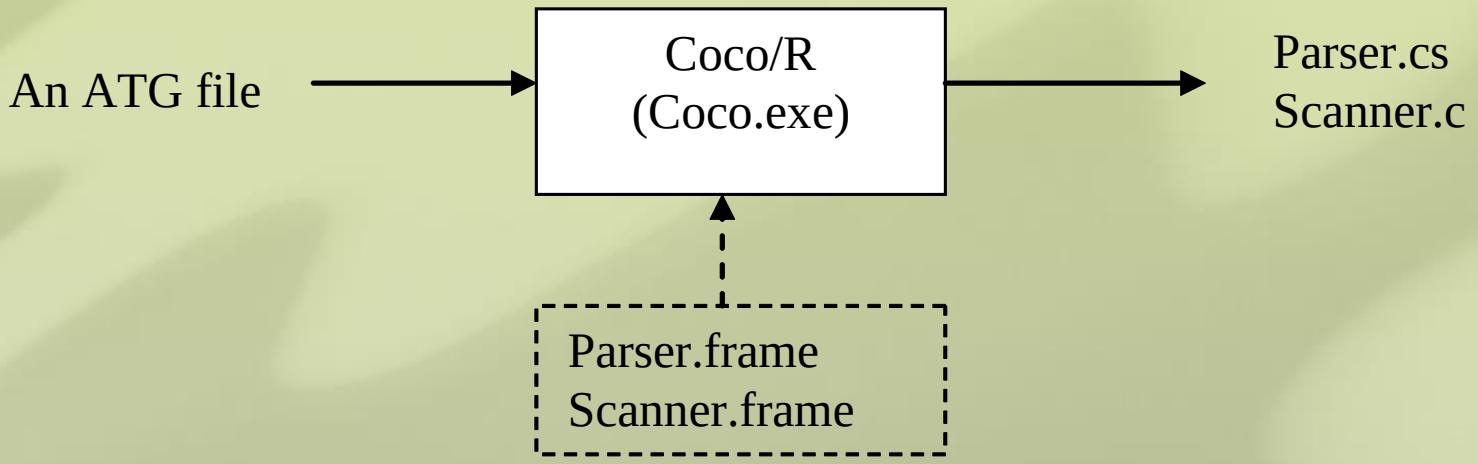
https://ssw.jku.at/Research/Projects/Coco/Doc/UserManual.pdf

Table of Contents

Compiler Construction Tools Coco/R



Coco/R Input and Output

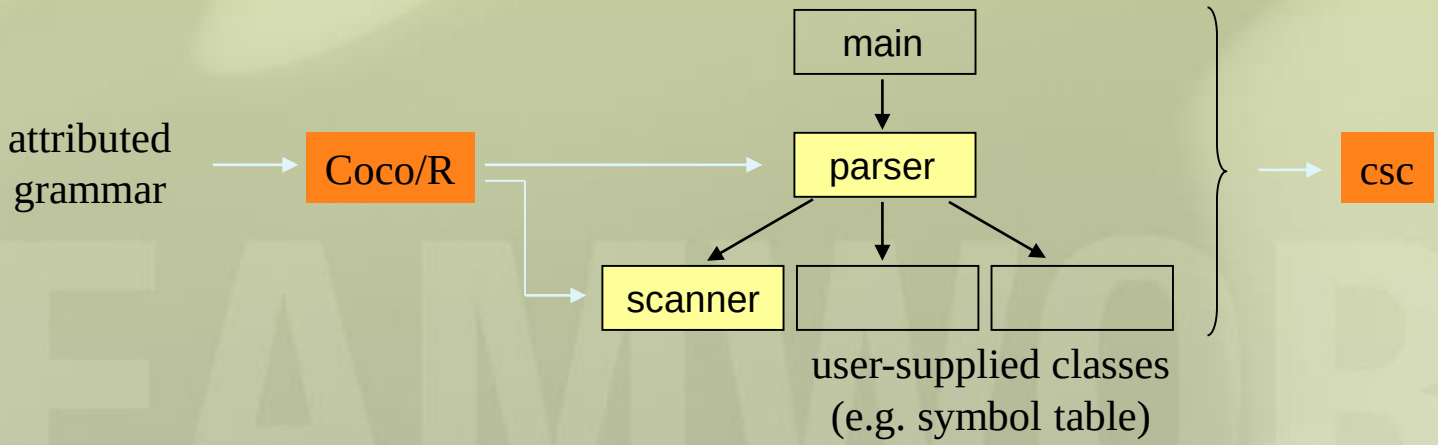


Coco/R - Compiler Compiler / Recursive Descent

Facts

- Generates a scanner and a parser from an attributed grammar
  - scanner as a deterministic finite automaton (DFA)
  - recursive descent parser
- Developed at the University of Linz (Austria)
- There are versions for C#, Java, C/C++, VB.NET, Delphi, Modula-2, Oberon, ...
- Gnu GPL open source: http://ssw.jku.at/Coco/

How it works



Coco/R Command Format

Coco/R Command format: Coco Grammar.ATG {Option}

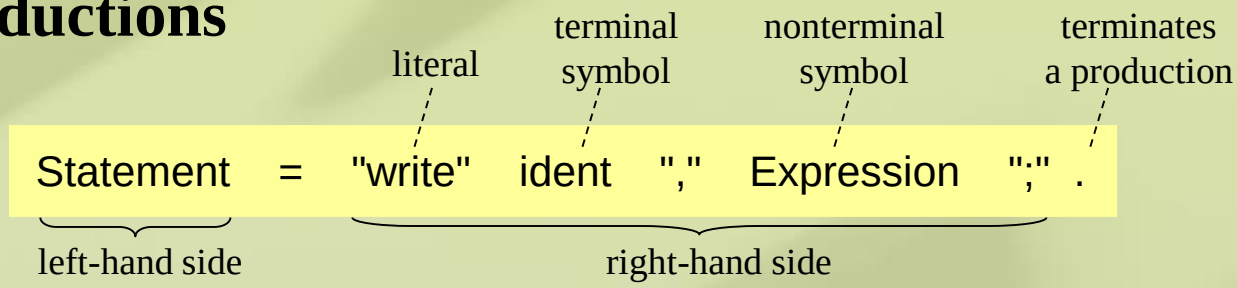
Options:

-namespace <namespaceName>
-frames <frameFilesDirectory>
-trace <traceString>
-o <outputDirectory>

Note: Scanner.frame and Parser.frame files must be in the ATG file directory or in the directory specified in the -frames option.

Coco EBNF Notation

Productions



by convention

- terminal symbols start with lower-case letters
- nonterminal symbols start with upper-case letters

Metasymbols

|	separates alternatives	a | b | c	≡ a or b or c
(...)	groups alternatives	a (b | c)	≡ ab | ac
[...]	optional part	[a] b	≡ ab | b
{...}	iterative part	{a}b	≡ b | ab | aab | aaab | ...

Coco/R Command Format Trace Option

Coco/R Command format: Coco Grammar.ATG {Option}

Trace Option: -trace <traceString>

Valid characters in the trace string:

A trace automaton
F list first/follow sets
G print syntax graph
I trace computation of first sets
J list ANY and SYNC sets
P print statistics
S list symbol table
X list cross reference table

Note: The trace information is written to the file trace.txt in the project folder.

Coco/R Command Format Trace Option

Coco/R Command format: Coco Grammar.ATG {Option}

Trace Option: -trace <traceString>

Valid characters in the trace string:

A trace automaton
F list first/follow sets
G print syntax graph
I trace computation of first sets
J list ANY and SYNC sets
P print statistics
S list symbol table
X list cross reference table

Note: The trace information is written to the file trace.txt in the project folder.

Structure of a Compiler Description (ATG Files)

[UsingClauses]
"COMPILER" ident
[GlobalFieldsAndMethods]
ScannerSpecification
ParserSpecification
"END" ident "."

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

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

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

ATG File Example

```
using System.IO;
```

To be included in the beginning of Parser.cs and Scanner.cs (optional)

```
COMPILER List
```

```
    int IdentCount = 0;
```

To be included in the beginning of the Parser class in Parser.cs (optional)

```
CHARACTERS
    UpperLetter = 'A'..'Z'.
    LowerLetter = 'a'..'z'.
    letter = UpperLetter + LowerLetter.
    digit = '0'..'9'.
TOKENS
    ident = letter { letter | digit }.
IGNORE '\n'+'\t'+' '
```

Scanner specification used to generate Scanner,cs

```
PRODUCTIONS

List =
    ident          (. IdentCount++; .)
    { ',' ident    (. IdentCount++; .)
    }
    .
```

Parser specification used to generate Parser,cs

```
END List.
```

Table of Contents

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      = '\r'.
   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

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

Literal Tokens

Literal tokens can be used without declaration

```
TOKENS
   ...
PRODUCTIONS
   ...
   Statement = "while" ... .
```

... but one can also declare them

```
TOKENS
   while = "while".
   ...
PRODUCTIONS
   ...
   Statement = while ... .
```

Sometimes useful because Coco/R generates constant names (in the Parser class) for the token numbers of all declared tokens

```
const int _while = 17;
```

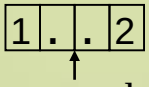
Context-dependent Tokens

Problem

floating point number	1.23
integer range	1..2

Scanner tries to recognize the longest possible token

decides to scan a float



got stuck; no way to continue in float

CONTEXT clause

```
TOKENS
   intCon    =  digit {digit}
             |  digit {digit} CONTEXT ("..").
   floatCon  =  digit {digit} "." digit {digit}.
```

Recognize a digit sequence as an *intCon* if its right-hand context is ".."

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}.  (.  foreach (char ch in t.val)
                                 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 expressions
- must be ignored by the parser

Example

```
COMMENTS FROM "/*" TO "*/" NESTED
COMMENTS FROM "//" TO "\r\n"
```

If comments are not nested they can also be described as pragmas
Advantage: can be semantically processed

White Space and Case Sensitivity

White space

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

blanks are ignored by default

character set

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 value returned to the parser retains original casing

Interface of the Generated Scanner

```
public class Scanner {
   public Buffer buffer;
   public Scanner (string fileName);
   public Scanner (Stream 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
   public Token next;
}
```

Table of Contents

Structure of a Parser Specification

```
ParserSpecification = "PRODUCTIONS" {Production}.
Production  =  ident [FormalAttributes] '=' EbnfExpr '.'.
```

```
EbnfExpr    =  Alternative { '|' Alternative}.
Alternative =  [Resolver] {Element}.
Element     =  Symbol [ActualAttributes]
            |  '(' EbnfExpr ')'
            |  '[' EbnfExpr ']'
            |  '{' EbnfExpr '}'
            |  "ANY"
            |  "SYNC"
            |  SemAction.
Symbol      =  ident
            |  string | char.
```

```
SemAction  =  "(." ArbitraryCSharpStatements ".)".
Resolver   =  "IF" '(' ArbitraryCSharpPredicate ')'.
```

```
FormalAttributes   =  '<' ArbitraryText '>'.
ActualAttributes   =  '<' ArbitraryText '>'.
```

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 C# code between (. and .)

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

Semantic action (local variable declaration)

semantic action (statement)

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

Global semantic declarations

```
using System.IO;
COMPILER Sample
   Stream s;
   void OpenStream(string path) {
      s = File.OpenRead(path);
      ...
   }
...
PRODUCTIONS
   Sample = ...        (. OpenStream("in.txt"); .)
   ...
END Sample.
```

using namespaces

Semantic action (global declarations, i.e. becoming fields and methods of the Parser class)

semantic actions can access global declarations as well as classes included in the project

Attributes

For nonterminal symbols

	actual attributes (RHS)	*formal attributes (LHS)*
input attributes pass values from the "caller" to a production	... = ... IdentLIst<type> ...	IdentList<Type t> = ...
output attributes pass results of a production to the "caller"	... = ... Expr<out n> ... ... = ... List<ref b> ...	Expr<out int val> = ... List<ref StringBuilder buf> = ...

For terminal symbols

no explicit attributes;
values are returned
by the scanner

adapter nonterminals necessary

```
Number<out int n> =
   number    (. n = Convert.ToInt32(t.val); .) .
```

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

Parser has two global token variables

```
Token t;     // most recently recognized token
Token la;    // lookahead token (not yet recognized)
```

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.
```

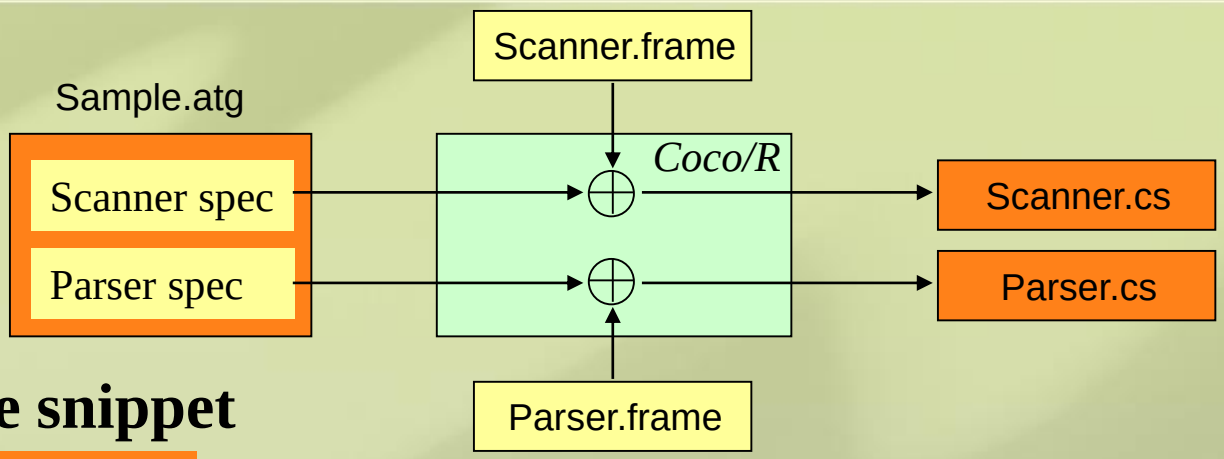
any token except "int"

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 {
   const char EOL = '\n';
   const int eofSym = 0;
-->declarations
   ...
   public Scanner (Stream s) {
      buffer = new Buffer(s, true);
      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");
    }
}
```

Table of Contents

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} '}'
  |   ...
  ).
```

synchronization points

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?

Locations 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 TextWriter errorStream = Console.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) {
      errorStream.WriteLine(errMsgFormat, 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;
          ...
       }
       errorStream.WriteLine(errMsgFormat, line, col, msg);
       count++;
    }

}
```

syntax error messages generated by Coco/R

Table of Contents

Terminal Start Symbols of Nonterminals (First Sets)

Those terminal symbols with which a nonterminal symbol can start

```
Expr    = ["+" | "-"] Term {("+" | "-") Term}.
Term    = Factor {("*" | "/") Factor}.
Factor  = ident | number | "(" Expr ")".
```

First(Factor) = ident, number, "("

First(Term) = First(Factor)
= ident, number, "("

First(Expr) = "+", "-", First(Term)
= "+", "-", ident, number, "("

Terminal Successors of Nonterminals (Follow Sets)

Those terminal symbols that can follow a nonterminal in the grammar

```
Expr    = ["+" | "-"] Term {("+" | "-") Term}.
Term    = Factor {("*" | "/") Factor}.
Factor  = ident | number | "(" Expr ")".
```

Where does *Expr* occur on the right-hand side of a production? What terminal symbols can follow there?

Follow(Expr) =")", eof

Follow(Term) = "+", "-", Follow(Expr)
= "+", "-", ")", eof

Follow(Factor) ="*", "/", Follow(Term)
= "*", "/", "+", "-", ")", eof

LL(1) Condition

For recursive descent parsing a grammar must be LL(1)

(parseable from **L**eft to right with **L**eft-most canonical derivations and **1** lookahead symbol)

Definition

1. A grammar is LL(1) if all its productions are LL(1).
2. A production is LL(1) if all its alternatives have disjoint select sets.

```
S = a b | c.
```

LL(1)

First(a b) = {a}
First(c) = {c}

```
S = a b | T.
T = [a] c.
```

not LL(1)

First(a b) = {a}
First(T) = {a, c}

In other words

The parser must always be able to select one of the alternatives by looking at the lookahead token.

```
S = (a b | T).
```

if the parser sees an "a" here it cannot decide which alternative to select

How to Remove LL(1) Conflicts

Factorization

```
IfStatement =  "if" "(" Expr ")" Statement
             |  "if" "(" Expr ")" Statement "else" Statement.
```

Extract common start sequences

```
IfStatement =  "if" "(" Expr ")" Statement (
                                            | "else" Statement
                                            ).
```

... or in EBNF

```
IfStatement =  "if" "(" Expr ")" Statement ["else" Statement].
```

Sometimes nonterminal symbols must be inlined before factorization

```
Statement   =  Designator "=" Expr ";"
             |  ident "(" [ActualParameters] ")" ";".
Designator  =  ident {"." ident}.
```

Inline *Designator* in *Statement*

```
Statement   =  ident {"." ident} "=" Expr ";"
             |  ident "(" [ActualParameters] ")" ";".
```

then factorize

```
Statement   =  ident  (   {"." ident} "=" Expr ";"
                       |  "(" [ActualParameters] ")" ";"
                       ).
```

How to Remove Left Recursion

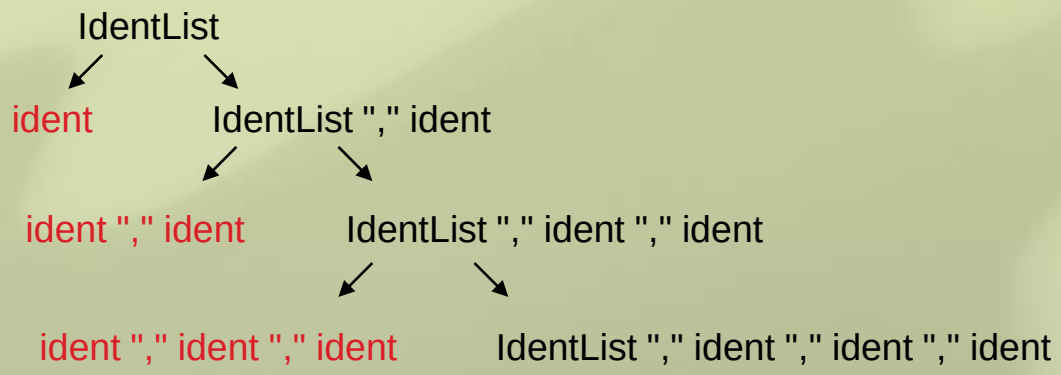
Left recursion is always an LL(1) conflict and must be eliminated

For example

```
IdentList = ident | IdentList "," ident.
```

(both alternatives start with *ident*)

generates the following phrases



can always be replaced by iteration

```
IdentList = ident {"," ident}.
```

Hidden LL(1) Conflicts

EBNF options and iterations are hidden alternatives

S = [α] β.	⇔	S = α β | β.	α and β are arbitrary EBNF expressions
S = {α} β.	⇔	S = β | α β | α α β |	

Rules

S = [α] β.	First(α) ∩ First(β) must be {}
S = {α} β.	First(α) ∩ First(β) must be {}

S = α [β].	First(β) ∩ Follow(S) must be {}
S = α {β}.	First(β) ∩ Follow(S) must be {}

Removing Hidden LL(1) Conflicts

```
Name = [ident "."] ident.
```

Where is the conflict and how can it be removed?

```
Name = ident ["." ident].
```

Is this production LL(1) now?

We have to check if First("." ident) ∩ Follow(Name) = {}

```
Prog = Declarations ";" Statements.
Declarations = D {";" D}.
```

Where is the conflict and how can it be removed?

Inline *Declarations* in *Prog*

```
Prog = D {";" D} ";" Statements.
```

First(";" D) ∩ First(";" Statements) ≠ {}

```
Prog = D ";" {D ";"} Statements.
```

We still have to check if First(D ";") ∩ First(Statements) = {}

Dangling Else

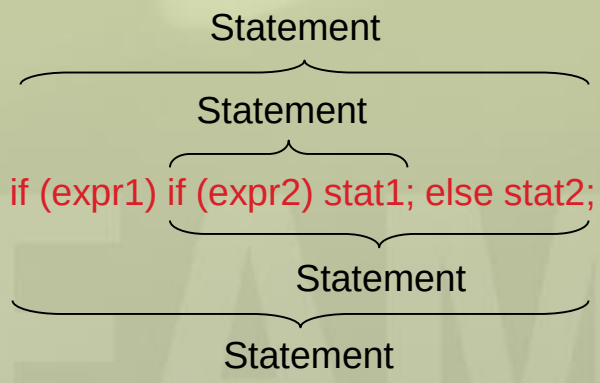
If statement in C# or Java

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

This is an LL(1) conflict!

First("else" Statement) ∩ Follow(Statement) = {"else"}

It is even an ambiguity which cannot be removed



We can build 2 different syntax trees!

Can We Ignore LL(1) Conflicts?

An LL(1) conflict is only a warning

The parser selects the first matching alternative

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

← if the lookahead token is *a* the parser selects this alternative

Example: Dangling Else

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

If the lookahead token is "else" here
the parser starts parsing the option;
i.e. the "else" belongs to the innermost "if"

if (expr1) if (expr2) stat1; else stat2;

Statement

Statement

Luckily this is what we want here.

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 (Aug 22, 2006)
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
```

Problems with LL(1) Conflicts

Some conflicts are hard to remove by grammar transformations

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

} both alternatives can start with '(' ident ')'

Transformations can corrupt readability

```
Using  = "using" [ident '='] Qualid ';'.
Qualid = ident {'.' ident}.
```

⇒

```
Using  = "using" ident (   {'.' ident} ';'
                       |   '=' Qualid ';'.
                       ).
```

Semantic actions may prevent factorization

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

=> Coco/R offers a special mechanism to resolve LL(1) conflicts

LL(1) Conflict Resolvers

Syntax

```
EBNFexpr   =  Alternative { '|' Alternative}.
Alternative  =  [Resolver] Element {Element}.
Resolver     =  "IF" '(' ArbitraryCSharpPredicate ')'.
```

Example

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

We have to write the following method (in the global semantic declarations)

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

returns *true* if the input is
   ident = ...
and *false* if the input is
   ident . ident ...

Token names

```
TOKENS
   ident     = letter {letter | digit}.
   number  = digit {digit}.
   assign    = '='.
   ...
```

Coco/R generates the following declarations for tokens names

```
const int  _EOF      = 0;
const int  _ident    = 1;
const int  _number = 2;
const int  _assign   = 3;
...
```

Example

Conflict resolution by a 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

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

Summary

- Coco/R generates a scanner and a recursive descent parser from an attributed grammar
- LL(1) conflicts can be handled with resolvers
  Grammars for C# and Java are available in Coco/R format
- Coco/R is open source software (Gnu GPL)
  http://ssw.jku.at/Coco/
- Coco/R has been used by us to build
  - a white-box test tool for C#
  - a profiler for C#
  - a static program analyzer for C#
  - a metrics tool for Java
  - compilers for domain-specific languages
  - a log file analyzer
  - ...
- Many companies and projects use Coco/R
  - SharpDevelop: a C# IDE — www.icsharpcode.net
  - Software Tomography: Static Analysis Tool — www.software-tomography.com
  - CSharp2Html: HTML viewer for C# sources — www.charp2html.net
  - currently 39000 hits for Coco/R in Google