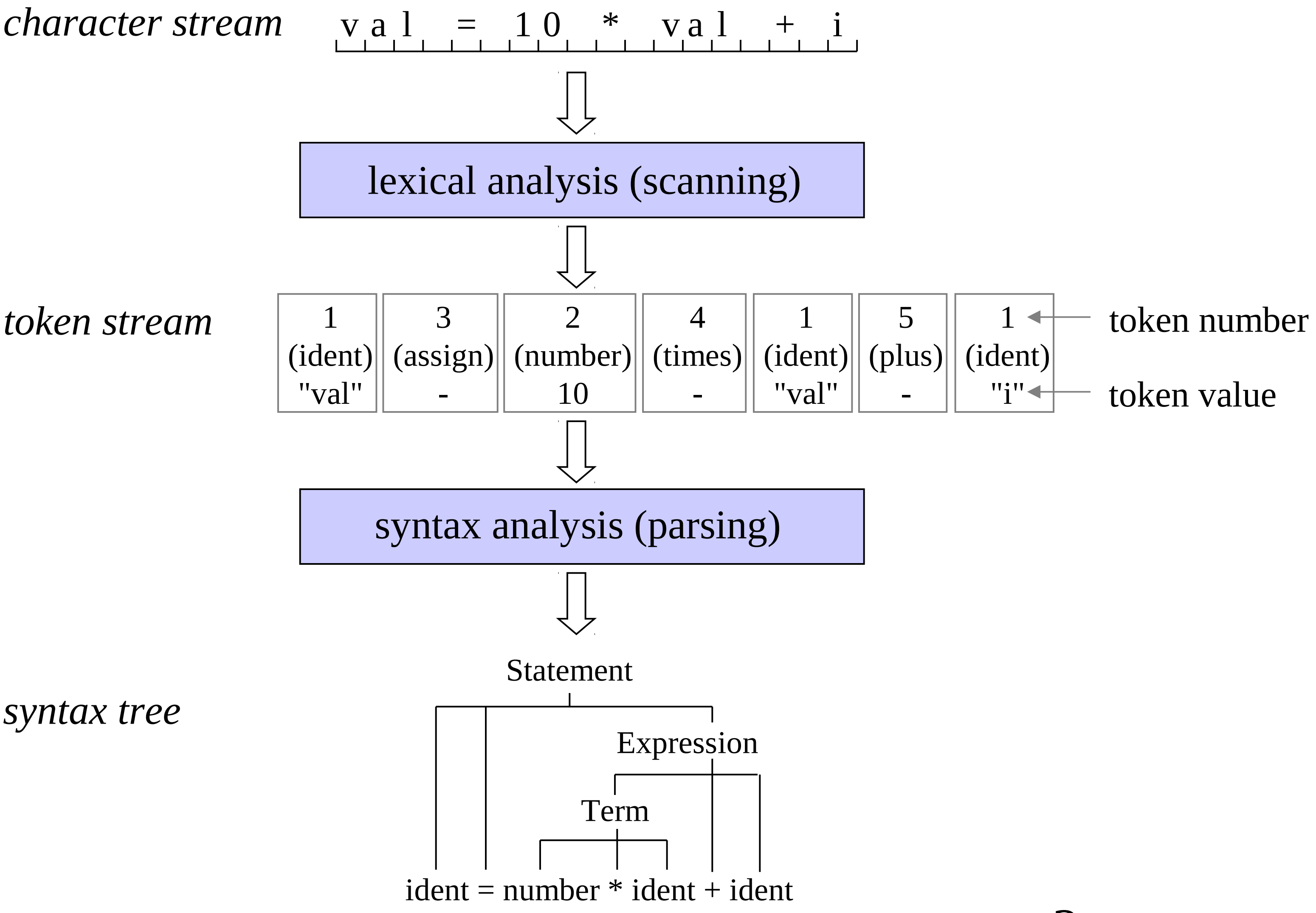
# Generating Compilers with Coco/R

Hanspeter Mössenböck
University of Linz
http://ssw.jku.at/Coco/

1. Compilers
2. Grammars
3. Coco/R Overview
4. Scanner Specification
5. Parser Specification
6. Error Handling
7. LL(1) Conflicts
8. Case Study

# Compilation Phases



*character stream*

v a l = 1 0 * v a l + i

lexical analysis (scanning)

*token stream*

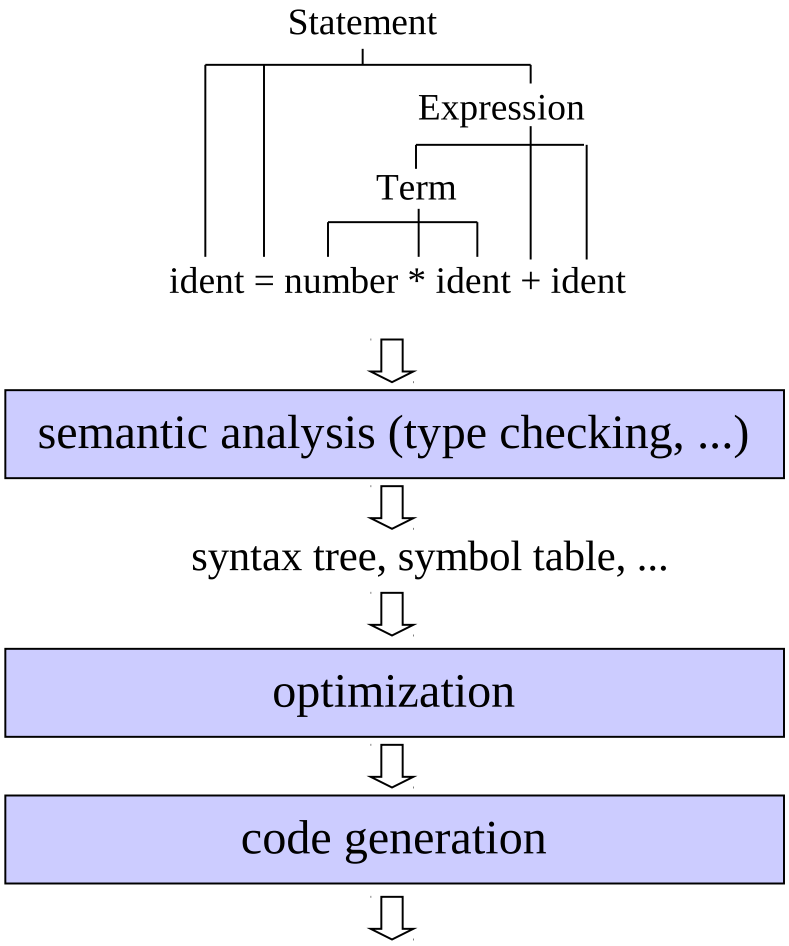
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 3 | 2 | 4 | 1 | 5 | 1 | token number |
| (ident) | (assign) | (number) | (times) | (ident) | (plus) | (ident) | |
| "val" | - | 10 | - | "val" | - | "i" | token value |

syntax analysis (parsing)

*syntax tree*

# Compilation Phases

*syntax tree*



semantic analysis (type checking, ...)

*intermediate representation*

syntax tree, symbol table, ...

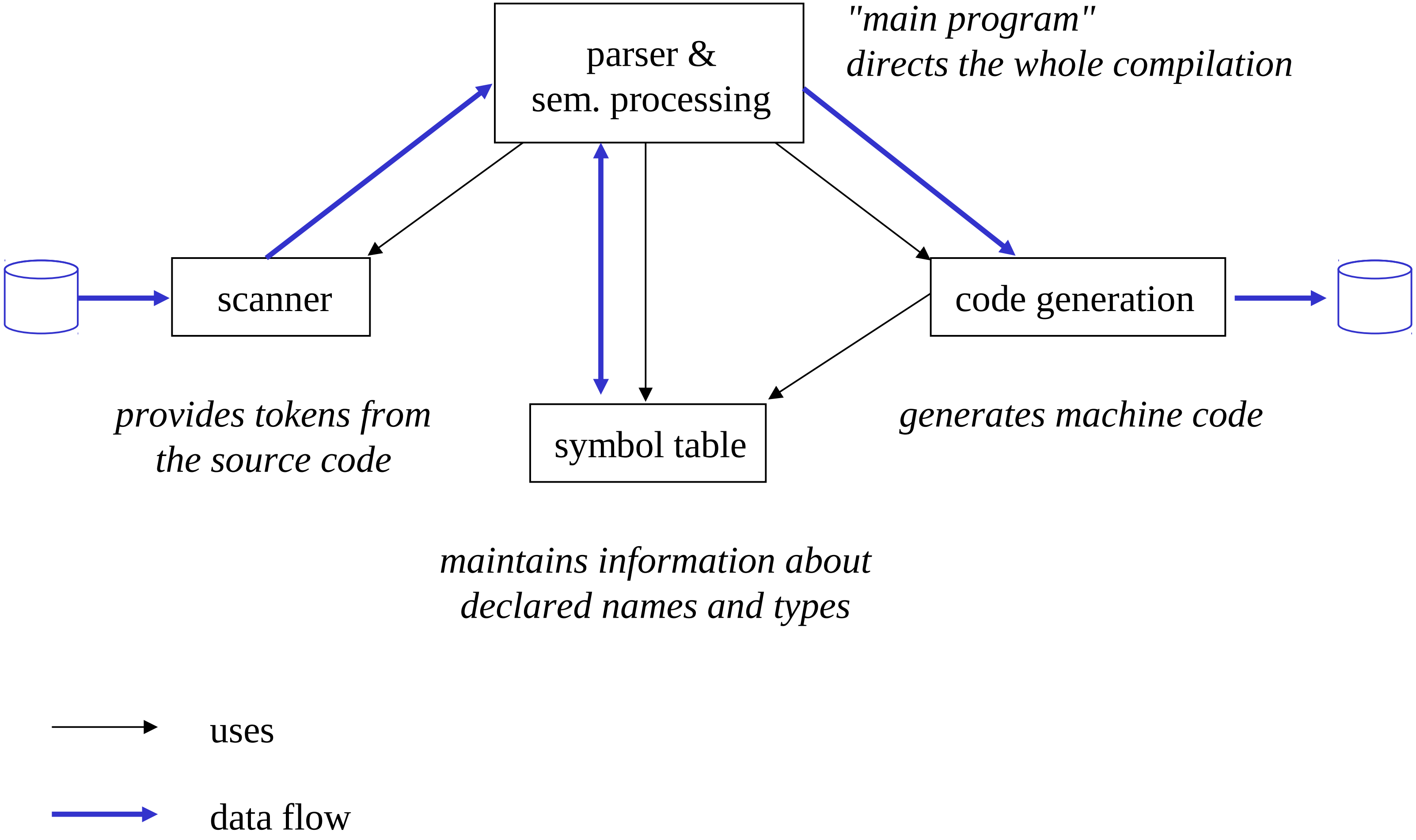
optimization

code generation

*machine code*

```
const 10
load  1
mul
...
```

# *Structure of a Compiler*



parser & sem. processing

*"main program" directs the whole compilation*

scanner

*provides tokens from the source code*

code generation

*generates machine code*

symbol table

*maintains information about declared names and types*

uses

data flow

# Generating Compilers with Coco/R

1. Compilers
2. Grammars
3. Coco/R Overview
4. Scanner Specification
5. Parser Specification
6. Error Handling
7. LL(1) Conflicts
8. Case Study

# *What is a grammar?*

**Example**

```
Statement = "if" "(" Condition ")" Statement ["else" Statement].
```

**Four components**

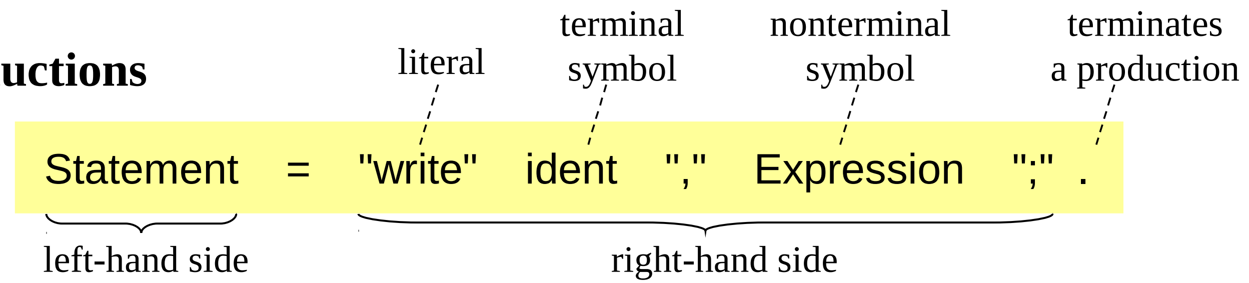
| | | |
|---|---|---|
| **terminal symbols** | are atomic | "if", ">=", ident, number, ... |
| **nonterminal symbols** | are decomposed into smaller units | Statement, Condition, Type, ... |
| **productions** | rules how to decompose nonterminals | Statement = Designator "=" Expr ";".<br>Designator = ident ["." ident].<br>... |
| **start symbol** | topmost nonterminal | CSharp |

# *EBNF Notation*

**Extended Backus-Naur form for writing grammars**

*John Backus*: developed the first Fortran compiler
*Peter Naur*: edited the Algol60 report

## Productions



```
Statement = "write" ident "," Expression ";" .
```

by convention

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

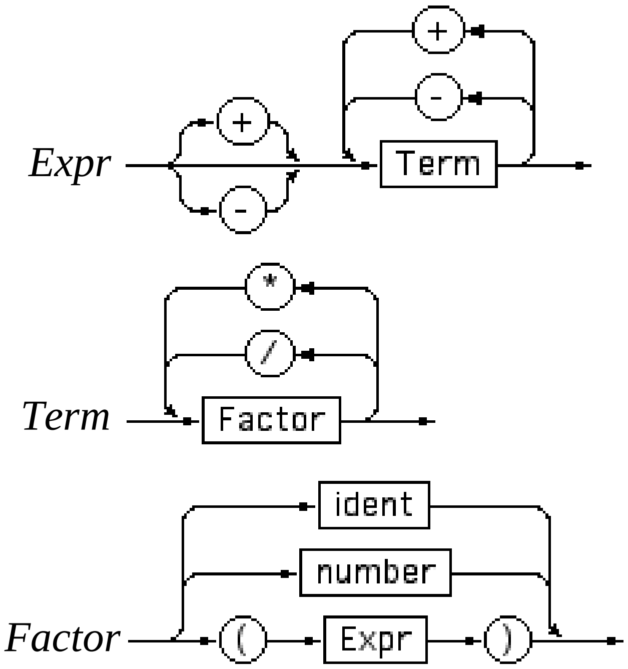
## Metasymbols

| | | | |
|---|---|---|---|
| \| | 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 \| ... |

# *Example: Grammar for Arithmetic Expressions*

## Productions

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



## Terminal symbols

| | |
|---|---|
| simple TS: | "+", "-", "*", "/", "(", ")"<br>(just 1 instance) |
| terminal classes: | ident, number<br>(multiple instances) |

## Nonterminal symbols

Expr, Term, Factor

## Start symbol

Expr

# Generating Compilers with Coco/R

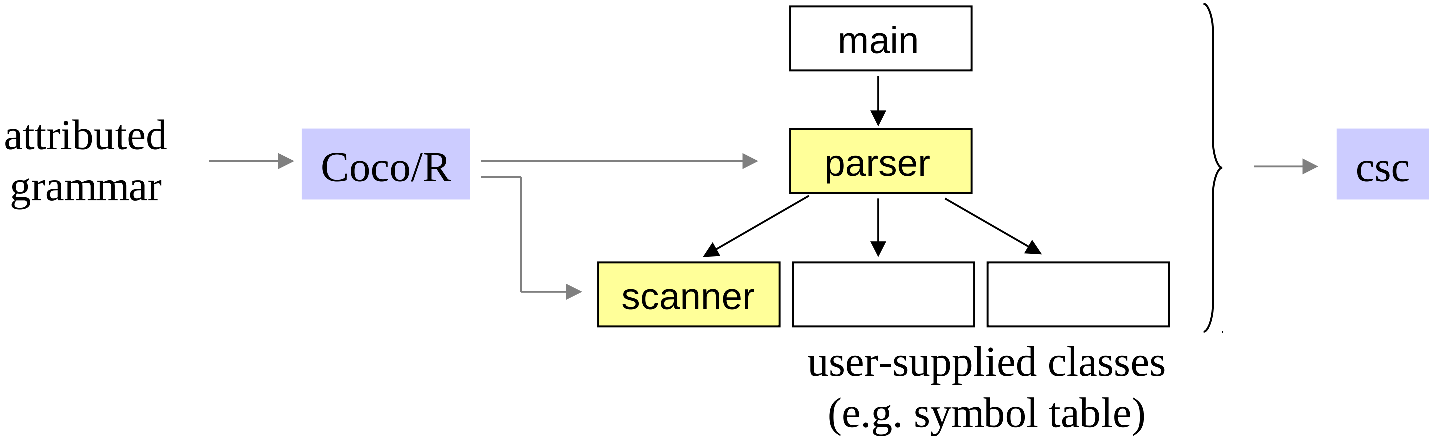
1. Compilers
2. Grammars
3. Coco/R Overview
4. Scanner Specification
5. Parser Specification
6. Error Handling
7. LL(1) Conflicts
8. Case Study

# *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



# A Very Simple Example

**Assume that we want to parse one of the following two alternatives**

```
red apple
```

```
orange
```

**We write a grammar ... and embed it into a Coco/R compiler description**

```
COMPILER Sample
PRODUCTIONS
    Sample = "red" "apple" | "orange".
END Sample.
```

file *Sample.atg*

**We invoke Coco/R to generate a scanner and a parser**

```
>coco Sample.atg
Coco/R (Aug 22, 2006)
checking
parser + scanner generated
0 errors detected
```

# A Very Simple Example

We write a main program

```
using System;
class Compile {
    static void Main(string[] arg)
        Scanner scanner = new Scanner(arg[0]);
        Parser parser = new Parser(scanner);
        parser.Parse();
        Console.Write(parser.errors.count + " errors detected");
    }
}
```

must
- create the scanner
- create the parser
- start the parser
- report number of errors

We compile everything ...

```
>csc Compile.cs Scanner.cs Parser.cs
```

... and run it

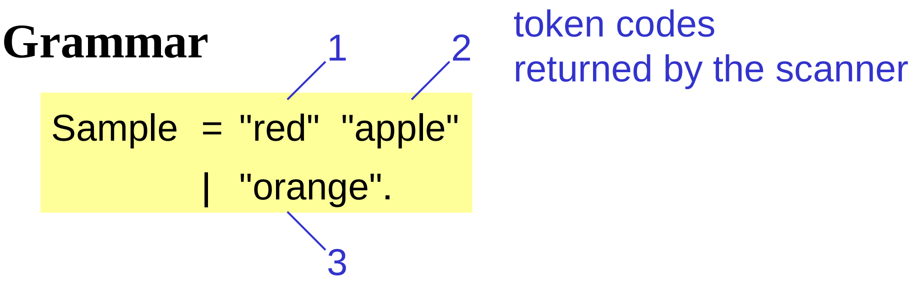
```
>Compile Input.txt
0 errors detected
```

file *Input.txt*

```
red apple
```

# *Generated Parser*

```
class Parser {
    ...
    void Sample() {
        if (la.kind == 1) {
            Get();
            Expect(2);
        } else if (la.kind == 3) {
            Get();
        } else SynErr(5);
    }
    ...
    Token la;  // lookahead token

    void Get () {
        la = Scanner.Scan(); ...
    }

    void Expect (int n) {
        if (la.kind == n) Get(); else SynErr(n);
    }

    public void Parse() {
        Get();
        Sample();
    }
    ...
}
```



**Grammar**

```
Sample  =  "red"  "apple"
        |  "orange".
```

token codes
returned by the scanner

# *A Slightly Larger Example*

**Parse simple arithmetic expressions**

```
calc  34 + 2 + 5
calc  2 + 10 + 123 + 3
```

**Coco/R compiler description**

```
COMPILER Sample
CHARACTERS
   digit = '0'..'9'.
TOKENS
   number = digit {digit}.
IGNORE '\r' + '\n'
PRODUCTIONS
   Sample = {"calc" Expr}.
   Expr = Term {'+' Term}.
   Term = number.
END Sample.
```

file *Sample.atg*

```
>coco Sample.atg
>csc Compile.cs Scanner.cs Parser.cs
>Compile Input.txt
```

The generated scanner and parser will check the syntactic correctness of the input

# *Now we add Semantic Processing*

```
COMPILER Sample
...
PRODUCTIONS
  Sample                  (. int n; .)
  = {  "calc"
       Expr<out n>        (. Console.WriteLine(n); .)
    }.
  /*---------------------------------------------------------*/
  Expr<out int n>         (. int n1; .)
  =  Term<out n>
     {  '+'
        Term<out n1>      (. n = n + n1; .)
     }.
  /*---------------------------------------------------------*/
  Term<out int n>
  =  number               (. n = Convert.Int32(t.val); .)
     .
END Sample.
```

This is called an "attributed grammar"

*Attributes*
similar to parameters
of the symbols

*Semantic Actions*
ordinary C# code
executed during parsing

# Generated Parser

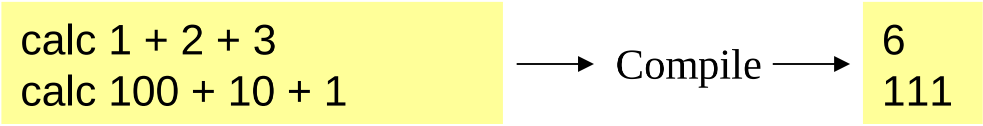
```
class Parser {
    ...
    void Sample() {
        int n;
        while (la.kind == 2) {
            Get();
            Expr(out n);
            Console.WriteLine(n);
        }
    }
    void Expr(out int n) {
        int n1;
        Term(out n);
        while (la.kind == 3) {
            Get();
            Term(out n1);
            n = n + n1;
        }
    }
    void Term(out int n) {
        Expect(1);
        n = Convert.ToInt32(t.val);
    }
    ...
}
```

```
Sample              (. int n; .)
= {   "calc"
      Expr<out n>   (. Console.WriteLine(n); .)
  }.
...
```

## Token codes

1 ... number
2 ... "calc"
3 ... '+'

```
>coco Sample.atg
>csc Compile.cs Scanner.cs Parser.cs
>Compile Input.txt
```

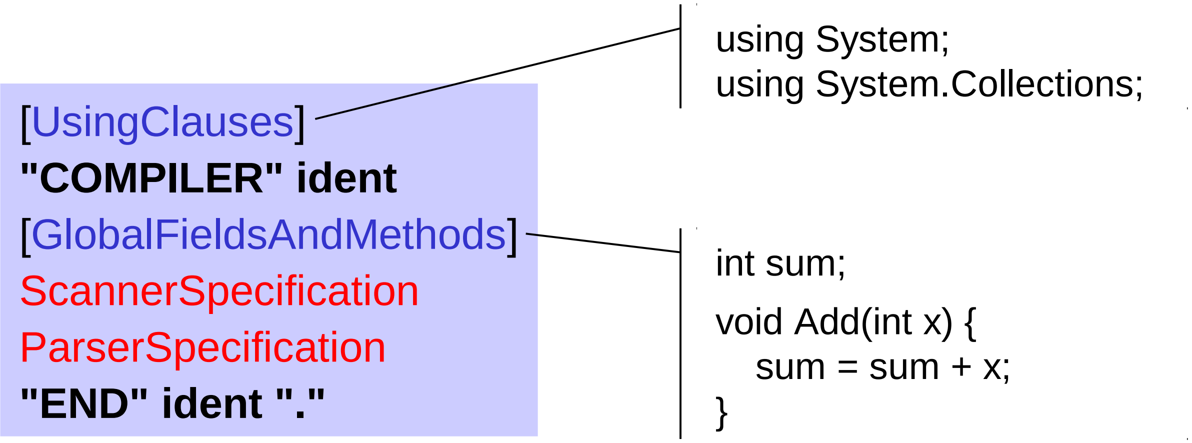


calc 1 + 2 + 3
calc 100 + 10 + 1

Compile

6
111

# *Structure of a Compiler Description*



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

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

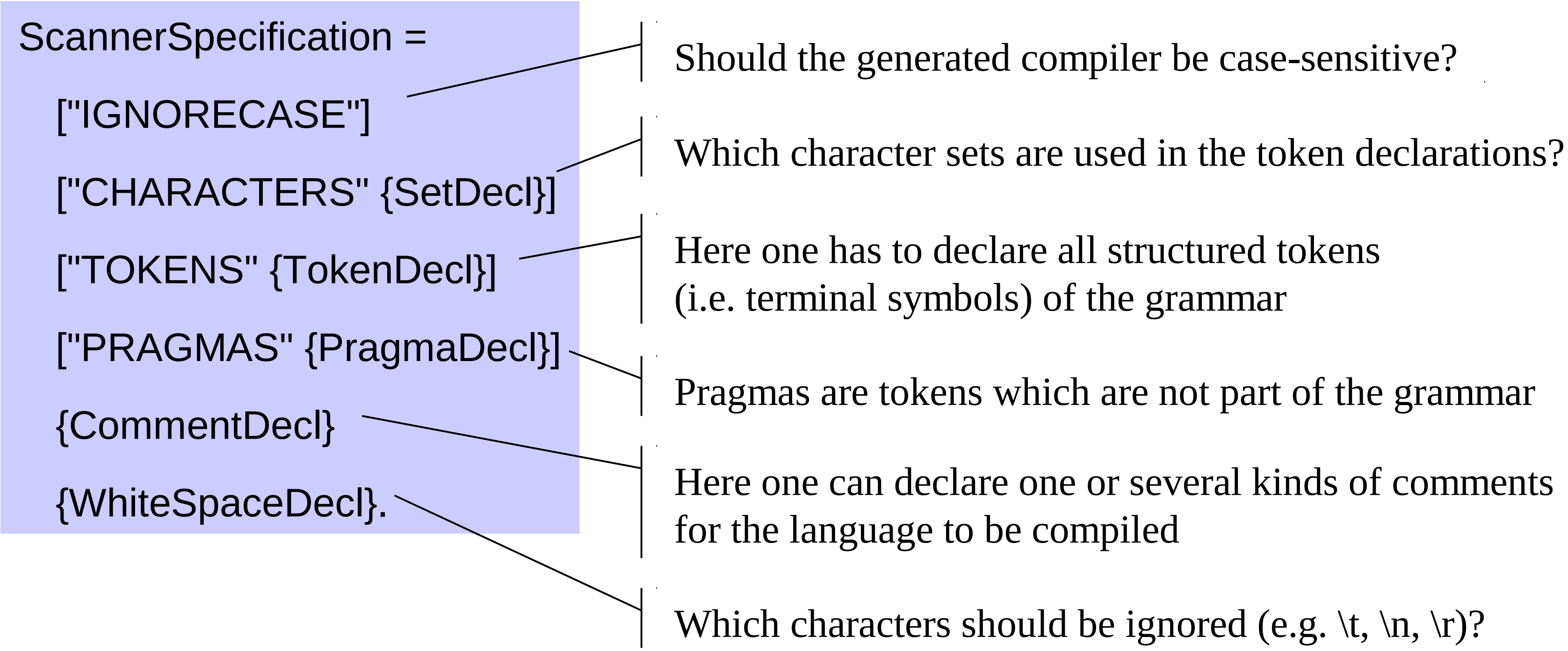
```
int sum;

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

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

# Generating Compilers with Coco/R

1. Compilers
2. Grammars
3. Coco/R Overview
4. Scanner Specification
5. Parser Specification
6. Error Handling
7. LL(1) Conflicts
8. Case Study

# *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".           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
```

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
for the token numbers of all declared tokens

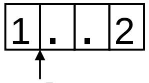
```
const int _while = 17;
```

# *Context-dependent Tokens*

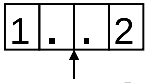
**Problem**

| | |
|---|---|
| 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'
```

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 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();
}
```

Scan(): main method: returns a token upon every call

Peek(): reads ahead from the current scanner position without removing tokens from the input stream

ResetPeek(): 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
}
```

# Generating Compilers with Coco/R

1. Compilers
2. Grammars
3. Coco/R Overview
4. Scanner Specification
5. Parser Specification
6. Error Handling
7. LL(1) Conflicts
8. Case Study

# Structure of a Parser Specification

```
ParserSpecification = "PRODUCTION" {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); .)
  .
```

local semantic declaration

semantic action

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.
```

import of namespaces

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

semantic actions can access global declarations
as well as imported classes

# *Attributes*

## For nonterminal symbols

| | *actual attributes* | *formal attributes* |
|---|---|---|
| ***input attributes***<br>pass values from the "caller" to a production | `... = ... IdentLIst<type> ...` | `IdentList<Type t> = ...` |
| ***output attributes***<br>pass results of a production to the "caller" | `... = ... Expr<out n> ...`<br>`... = ... List<ref b> ...` | `Expr<out int val> = ...`<br>`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.
```

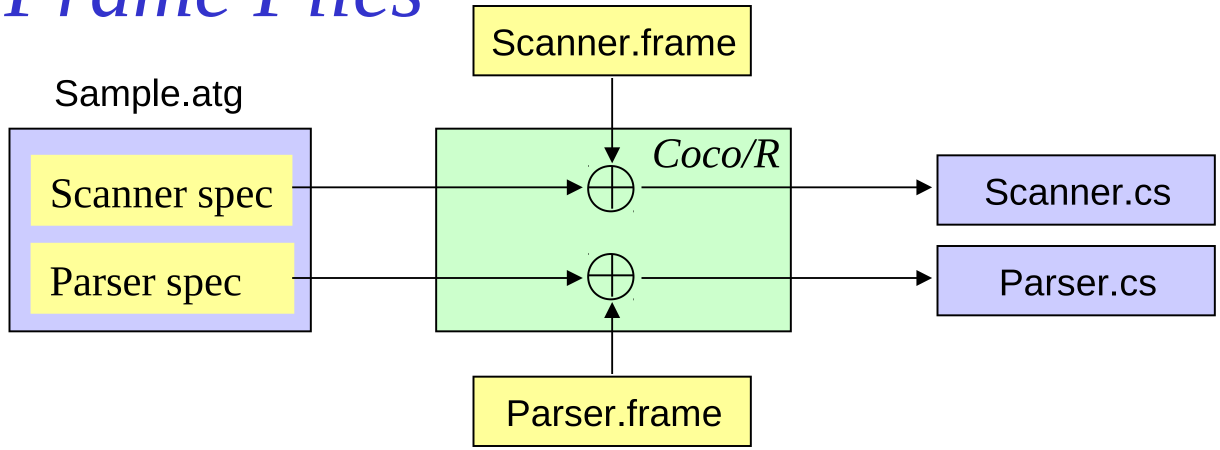
any token except "int"

**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");
    }
}
```

# Generating Compilers with Coco/R

1. Compilers
2. Grammars
3. Coco/R Overview
4. Scanner Specification
5. Parser Specification
6. Error Handling
7. LL(1) Conflicts
8. Case Study

# *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**

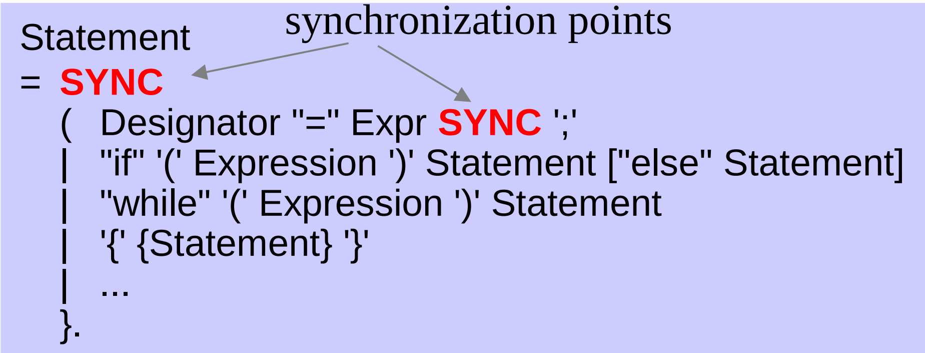
| | |
|---|---|
| *production* | S = a (b \| c \| d) e. |
| *input* | a x e |
| *error message* | -- line ... col ...: invalid S |

**Error message can be improved by rewriting the production**

| | |
|---|---|
| *productions* | S = a T e.<br>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?

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

# Generating Compilers with Coco/R

1. Compilers
2. Grammars
3. Coco/R Overview
4. Scanner Specification
5. Parser Specification
6. Error Handling
7. LL(1) Conflicts
8. Case Study

# *Terminal Start Symbols of Nonterminals*

**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*

**Those terminal symbols that can follow a nonterminal in the grammar**

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

Follow(Expr) = ")", eof

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

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

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

# *LL(1) Condition*

**For recursive descent parsing a grammar must be LL(1)**
(parseable from **L**eft to right with **L**eftcanonical 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 start with different terminal symbols

```
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*

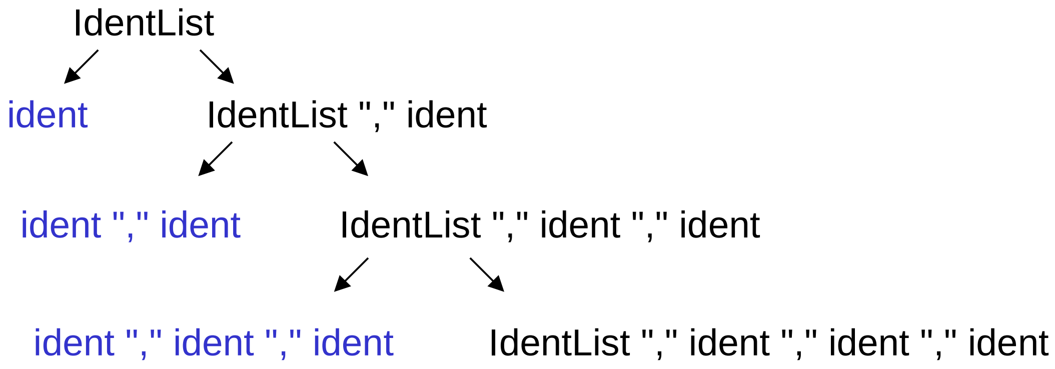
**Left recursion is always an LL(1) conflict and must be eliminated**

For example

IdentList = ident | IdentList "," ident.

(both alternatives start with *ident*)

generates the following phrases



IdentList

ident

IdentList "," ident

ident "," ident

IdentList "," ident "," ident

ident "," ident "," ident

IdentList "," ident "," ident "," ident

can always be replaced by iteration

IdentList = ident {"," ident}.

# *Hidden LL(1) Conflicts*

**EBNF options and iterations are hidden alternatives**

S = [$\alpha$] $\beta$. ⇔ S = $\alpha$ $\beta$ | $\beta$. $\alpha$ and $\beta$ are arbitrary EBNF expressions

S = {$\alpha$} $\beta$. ⇔ S = $\beta$ | $\alpha$ $\beta$ | $\alpha$ $\alpha$ $\beta$ | ... .

**Rules**

| | |
|---|---|
| S = [$\alpha$] $\beta$. | First($\alpha$) $\cap$ First($\beta$) must be {} |
| S = {$\alpha$} $\beta$. | First($\alpha$) $\cap$ First($\beta$) must be {} |

| | |
|---|---|
| S = $\alpha$ [$\beta$]. | First($\beta$) $\cap$ Follow(S) must be {} |
| S = $\alpha$ {$\beta$}. | First($\beta$) $\cap$ Follow(S) must be {} |

# *Removing Hidden LL(1) Conflicts*

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

Where is the conflict and how can it be removed?

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

Where is the conflict and how can it be removed?

# *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

$$\underbrace{\overbrace{\text{if (expr1) }\underbrace{\overbrace{\text{if (expr2) stat1;}}^{\text{Statement}}\text{ else stat2;}}_{\text{Statement}}}^{\text{Statement}}}_{\text{Statement}}$$

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"

$$\underbrace{\text{if (expr1) }\underbrace{\text{if (expr2) stat1; else stat2;}}_{\text{Statement}}}_{\text{Statement}}$$

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;
}
```

# *Example*

## Conflict resolution by exploiting 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

```
bool IsCast() {
    Token next = scanner.Peek();
    if (la.kind == _lpar && next.kind == _ident) {
        Obj obj = SymTab.Find(next.val);
        return obj != null && obj.kind == TYPE;
    } else return false;
}
```

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

# Generating Compilers with Coco/R

1. Compilers
2. Grammars
3. Coco/R Overview
4. Scanner Specification
5. Parser Specification
6. Error Handling
7. LL(1) Conflicts
8. Case Study -- The Programming Language *Taste*

# A Simple Taste Program

```
program Test {

   int i;

   // 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;
      }
   }
}
```

a single main program

global variables

methods without parameters

Main method

# *Syntax of Taste*

## Programs and Declarations

```
Taste     =  "program" ident "{"  {VarDecl} {ProcDecl} "}".
ProcDecl  =  "void" ident "(" ")" "{" { VarDecl | Stat} "}".
VarDecl   =  Type ident {"," ident} ";".
Type      =  "int" | "bool".
```

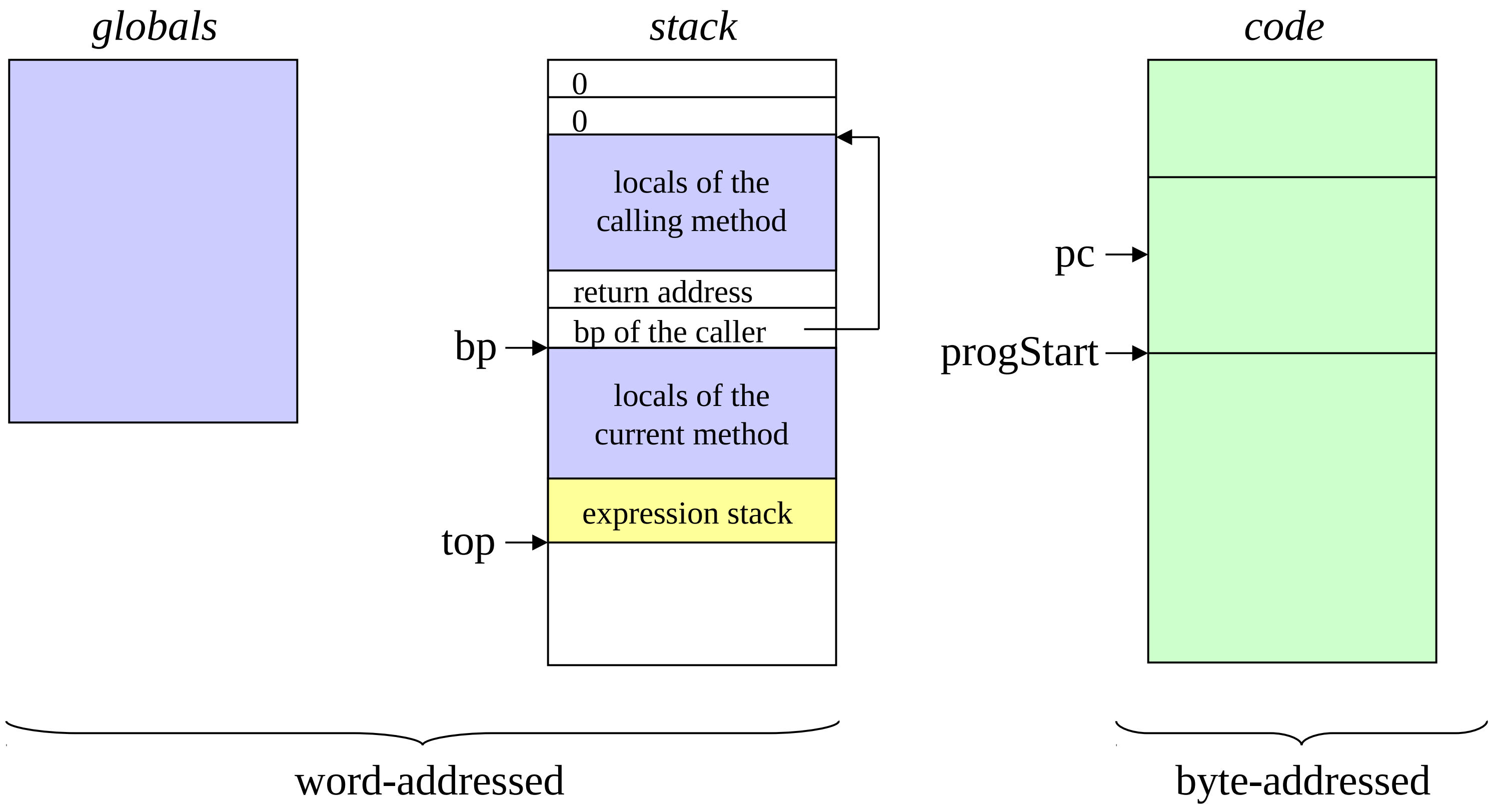
## Statements

```
Stat      =  ident "=" Expr ";"
          |  ident "(" ")" ";"
          |  "if" "(" Expr ")" Stat ["else" Stat].
          |  "while" "(" Expr ")" Stat
          |  "read" ident ";"
          |  "write" Expr ";"
          |  "{" { Stat | VarDecl } "}".
```

## Expressions

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

# *Architecture or the Taste VM*



# *Instructions of the Taste VM*

| | | | |
|---|---|---|---|
| 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()); |
| MUL | | Multiply | Push(Pop() * Pop()); |
| DIV | | Divide | x = Pop(); Push(Pop() / x); |
| NEG | | Negate | Push(-Pop()); |
| EQL | | Check if equal | if (Pop()==Pop()) Push(1); else Push(0); |
| LSS | | Check if less | if (Pop()>Pop()) Push(1); else Push(0); |
| GTR | | Check if greater | if (Pop()<Pop()) Push(1); else Push(0); |
| JMP | a | Jump | pc = a; |
| FJMP | a | Jump if false | if (Pop() == 0) pc = a; |
| READ | | Read integer | x = ReadInt(); Push(x); |
| WRITE | | Write integer | WriteInt(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(); |

# *Sample Translation*

## Source code

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

## Object 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
```

# Scanner Specification

```
COMPILER Taste

CHARACTERS
    letter = 'A'..'Z' + 'a'..'z'.
    digit = '0'..'9'.

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

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

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

PRODUCTIONS
    ...
END Taste.
```

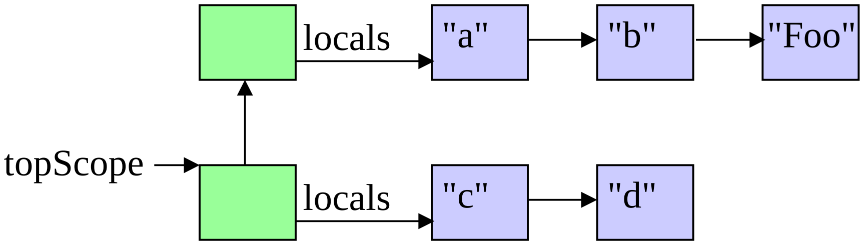
# *Symbol Table Class*

```
public class SymbolTable {
    public Obj   topScope;

    public       SymbolTable(Parser parser) {...}

    public Obj   Insert(string name, int kind, int type) {...}
    public Obj   Find(string name) {...}

    public void  OpenScope() {...}
    public void  CloseScope() {...}
}
```

```
public class Obj {
    public string  name;
    public int     kind;
    public int     type;
    public int     adr;
    public int     level;
    public Obj     locals;
    public Obj     next;
}
```

**Sample symbol table**

```
program P {
    int a;
    bool b;

    void Foo() {
        int c, d;
        ...
    }
    ...
}
```



# *Code Generator Class*

```
public class CodeGenerator {
    public int     pc;
    public int     progStart;

    public         CodeGenerator() {...}

    public void  Emit(int op) {...}
    public void  Emit(int op, int val) {...}
    public void  Patch(int adr, int val) {...}

    ...
}
```

# *Parser Specification -- Declarations*

```
PRODUCTIONS
   Taste                   (. string name; .)
   = "program"
      Ident<out name>      (. tab.OpenScope(); .)
      '{'
      { VarDecl }
      { ProcDecl }
      '}'                  (. tab.CloseScope(); .).
```

```
public SymbolTable    tab;
public CodeGenerator  gen;
```

```
   VarDecl                 (. string name; int type; .)
   = Type<out type>
      Ident<out name>      (. tab.Insert(name, VAR, type); .)
      { ',' Ident<out name> (. tab.Insert(name, VAR, type); .)
      } ';'.
```

```
   Type<out int type>
   =               (. type = UNDEF; .)
      ( "int"      (. type = INT; .)
      | "bool"     (. type = BOOL; .)
      ).
```

```
   ProcDecl                (.  string name; Obj obj; int adr; .)
   = "void"
      Ident<out name>      (.  obj = tab.Insert(name, PROC, UNDEF); obj.adr = gen.pc;
                               if (name == "Main") gen.progStart = gen.pc;
                               tab.OpenScope(); .)
      '(' ')'
      '{'                  (.  gen.Emit(ENTER, 0); adr = gen.pc - 2; .)
      { VarDecl | Stat }
      '}'                  (.  gen.Emit(LEAVE); gen.Emit(RET);
                               gen.Patch(adr, tab.topScope.adr);
                               tab.CloseScope(); .).
```

# *Parser Specification -- Expressions*

```
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 = BOOL; .)
  ].
```

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

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

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

```
AddOp<out int op>
=           (. op = UNDEF; .)
  (  '+'    (. op = PLUS; .)
  |  '-'    (. op = MINUS; .)
  ).
```

```
MulOp<out int op>
=           (. op = UNDEF; .)
  (  '*'    (. op = TIMES; .)
  |  '/'    (. op = SLASH; .)
  ).
```

# *Parser Specification -- Factor*

```
Factor<out int type>      (. int n; Obj obj; string name; .)
=                         (. type = UNDEF; .)
  (  Ident<out name>      (. obj = tab.Find(name); type = obj.type;
                             if (obj.kind == VAR) {
                               if (obj.level == 0) gen.Emit(LOADG, obj.adr);
                               else gen.Emit(LOAD, obj.adr);
                             } else SemErr("variable expected"); .)

  |  number               (. n = Convert.ToInt32(t.val);
                             gen.Emit(CONST, n); type = INT; .)

  |  '-'
     Factor<out type>     (. if (type != INT) {
                               SemErr("integer type expected");
                               type = INT;
                             }
                             gen.Emit(NEG); .)

  |  "true"               (. gen.Emit(CONST, 1); type = BOOL; .)
  |  "false"              (. gen.Emit(CONST, 0); type = BOOL; .)
  ).
```

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

# *Parser Specification -- Statements*

```
Stat                    (. int type; string name; Obj obj; int adr, adr2, loopstart; .)
= Ident<out name>       (. obj = tab.Find(name); .)
   ( '='                (. if (obj.kind != VAR) SemErr("can only assign to variables"); .)
     Expr<out type> ';' (. if (type != obj.type) SemErr("incompatible types");
                           if (obj.level == 0) gen.Emit(STOG, obj.adr);
                           else gen.Emit(STO, obj.adr); .)
   | '(' ')' ';'        (. if (obj.kind != PROC) SemErr("object is not a procedure");
                           gen.Emit(CALL, obj.adr); .)
   )

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

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

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

|  ... .
```

# *Parser Specification -- Statements*

```
Stat                         (. int type; string name; Obj obj; int adr, adr2, loopstart; .)
=  ...
|  "if"
   '(' Expr<out type> ')'    (. if (type != BOOL) SemErr("boolean type expected");
                                gen.Emit(FJMP, 0); adr = gen.pc - 2; .)
   Stat
   [  "else"                 (. gen.Emit(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 != BOOL) SemErr("boolean type expected");
                                gen.Emit(FJMP, 0); adr = gen.pc - 2; .)
   Stat                      (. gen.Emit(JMP, loopstart);
                                gen.Patch(adr, gen.pc); .) .
```

# *Main Program of Taste*

```
using System;

public class Taste {

    public static void Main (string[] arg) {
        if (arg.Length > 0) {
            Scanner scanner = new Scanner(arg[0]);
            Parser parser = new Parser(scanner);
            parser.tab = new SymbolTable(parser);
            parser.gen = new CodeGenerator();
            parser.Parse();
            if (parser.errors.count == 0) parser.gen.Interpret("Taste.IN");
        } else
            Console.WriteLine("-- No source file specified");
    }

}
```

## Building the whole thing

```
c:> coco Taste.atg
c:> csc Taste.cs Scanner.cs Parser.cs SymbolTable.cs CodeGenerator.cs
c:> Taste Sample.tas
```

# 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