POLITECNICO DI TORINO

# (01JEUHT) Formal Languages and Compilers

Laboratory N° 2

Stefano Scanzio

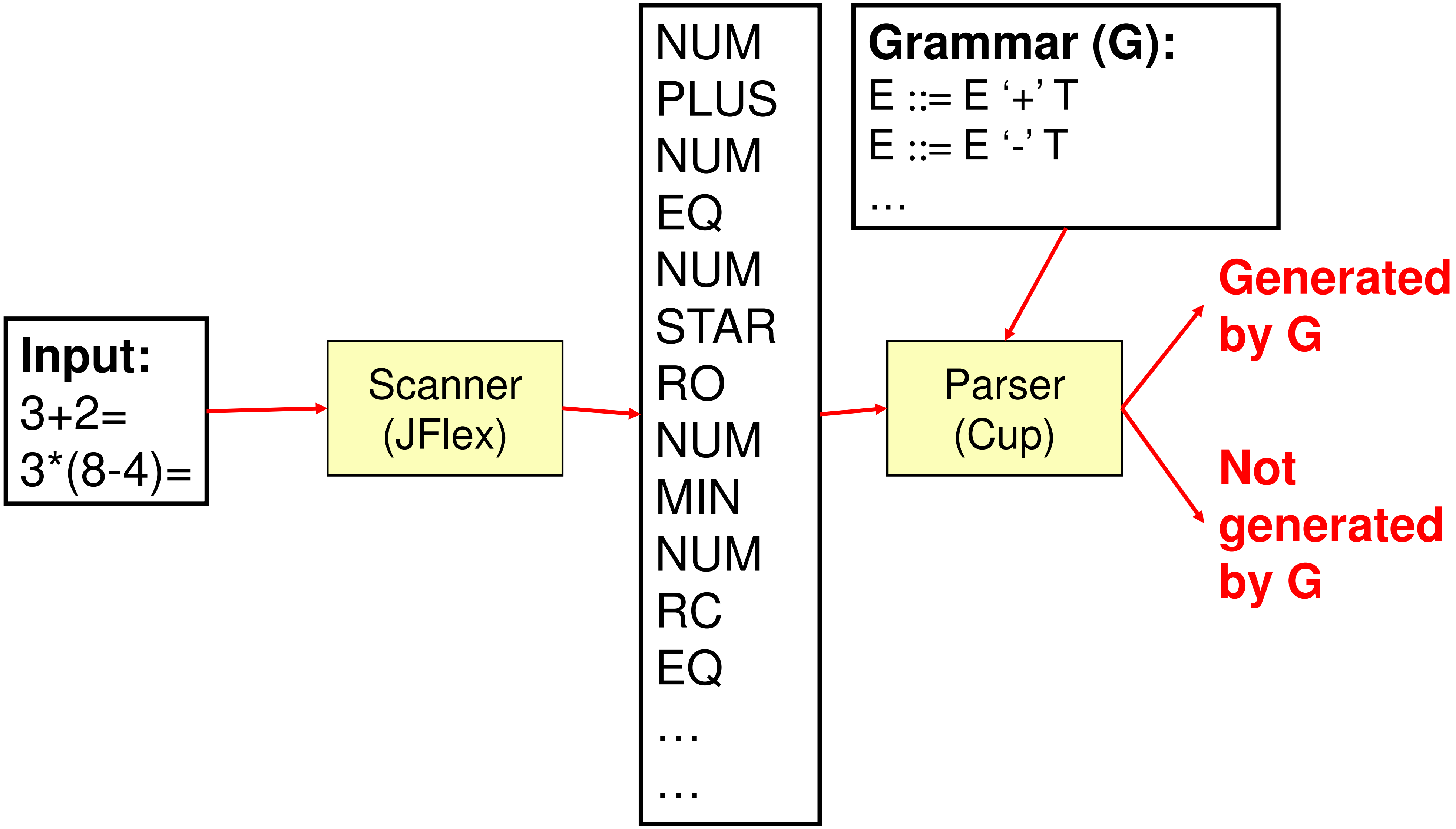
mail: stefano.scanzio@polito.it

Web: http://www.skenz.it/compilers

## Parser and syntax analyzer

- Given a non-ambiguous grammar and a sequence of input symbols, a parser is a program that verifies whether the sequence can be generated by means of a derivation from the grammar.
- A syntax analyzer (parser) is a program capable of associating to the input sequence the correct parse tree.
- Parsers can be classified as
  - top-down (parse tree is built from the root to the leaves )
  - bottom-up (parse tree is built from the leaves to the root ) : CUP

# Scanning and parsing



## Context-Free Grammar Definition

A CF grammar is described by

- T, NT, S, PR
- T: Terminals / tokens of the language
- NT: Non-terminals
  - Denote sets of strings generated by the grammar
- S: Start symbol
  - S ∈ NT
- PR: Production rules
  - Indicate how T and NT are combined to generate valid strings
    - PR: NT ::= T | NT

## Example

- **Derivation**:
  - A sequence of grammar rule applications and substitutions that transform a starting non-terminal into a sequence of terminals (tokens).

```
assign_stmt ::= ID EQ expr S ;
expr ::= expr operator term ;
expr ::= term ;
term ::= ID ;
term ::= FLOAT ;
term ::= INT ;
operator ::= PLUS ;
operator ::= MIN ;
```

## How bottom-up parsing works: Shift/Reduce tecnique

- A stack, initially empty, is used to keep track of symbols already recognized.
- Terminal symbols are pushed in the stack (shift), until the top of the stack contains a handle (right hand side of a production): the handle is then substituted by the corresponding non-terminal (reduce).
- Note that the reduce operation may only be applied to the top of the stack.
- Parsing is successful only when at the end of the input stream the stack contains only the start symbol

# Parse Trees and Shift/Reduce

**Input String:**

a1 , a2 , a3

**Scanner:**

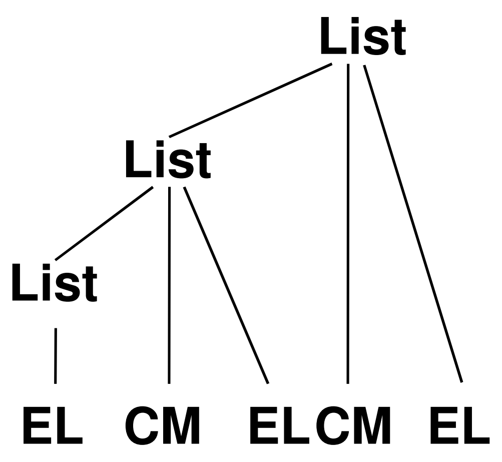
a1 , a2 , a3 → EL CM EL CM EL

**Recursive Left Grammar**

**List** ::= **List CM EL**

**List** ::= **EL**

**Parse Tree**



| **Action:** | **Stack:** |
|---|---|
| | ε |
| Shift: | EL |
| Reduce: | List |
| Shift: | List CM |
| Shift: | List CM EL |
| Reduce: | List |
| Shift: | List CM |
| Shift: | List CM EL |
| Reduce: | List |

## Introduction to CUP

- Cup is a parser generator that transforms the definition of a context-free grammar in a Java program that parses sequences of input symbols according to the grammar itself.
- Besides defining syntax rules, it is possible to specify actions to be executed whenever a production is reduced.
- The parser must be integrated by a scanner: some conventions simplify the integration of Cup-generated parses with JFlex-generated scanners.

- Official manual:

  http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html

## Source file format

- A Cup source file has a syntax very similar to Java programs.
- It can be ideally divided in the following sections:
  - Setup
  - Terminals and non-Terminals
  - Precedences (Next lesson)
  - Rules
- Comments are allowed following Java syntax (included in `/*` and `*/`, or preceded by `//`)

## Setup section

- This section contains all the directives needed for the parser
- Inclusion of Cup library and other libraries:

  import java_cup.runtime.*;

- User code: (Next lesson)
  - Ridefinition of Cup internal methods
  - Integration with scanner other than JFlex

## Terminals / Non-Terminals section

- It contains the definition of
  - Terminals: passed by JFlex
  - Non-Terminals
  - The grammar start symbol

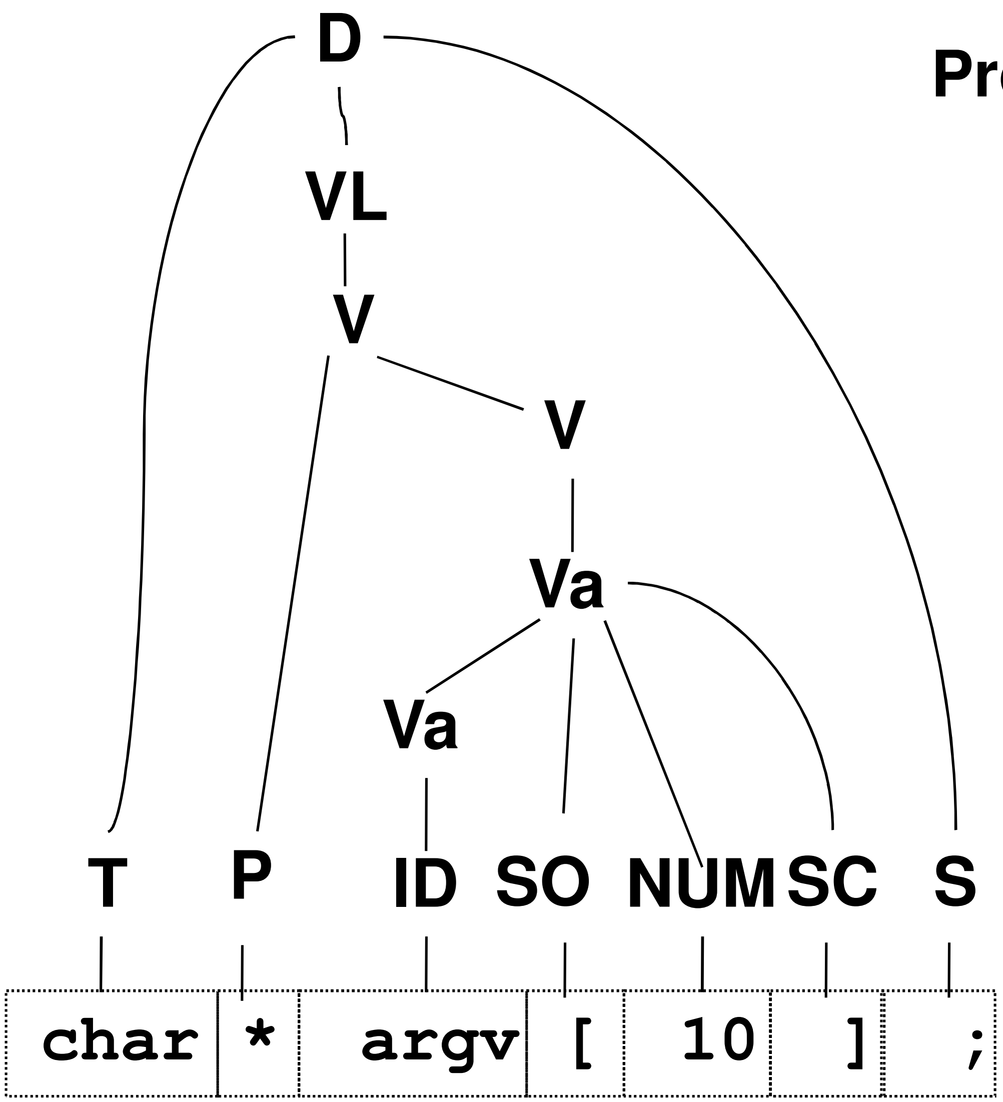
- Start symbol
  - start with <non_terminal_name> ;
  - It is the root of the parse tree
  - Only one occurrence of this keyword is allowed

## Terminals / Non-Terminals section

- Terminals
  - terminal <terminal_1>,…,<terminal_n> ;
    - <terminal>: name containing letters, ‘_’, ‘.’ and digits (the first character must be a letter)
  - Terminals are recognized by Jflex

- Non-Terminals
  - non terminal <non_terminal_1>,…,<non_terminal_n> ;
    - <non_terminal>: name containing letters, ‘_’, ‘.’ and digits (the first character must be a letter).

# Terminals / Non-Terminals section



**Productions (grammal rules):**

$D \longrightarrow T\ VL\ S$

$VL \longrightarrow V$

$VL \longrightarrow VL\ CM\ V$

$V \longrightarrow P\ V$

$V \longrightarrow Va$

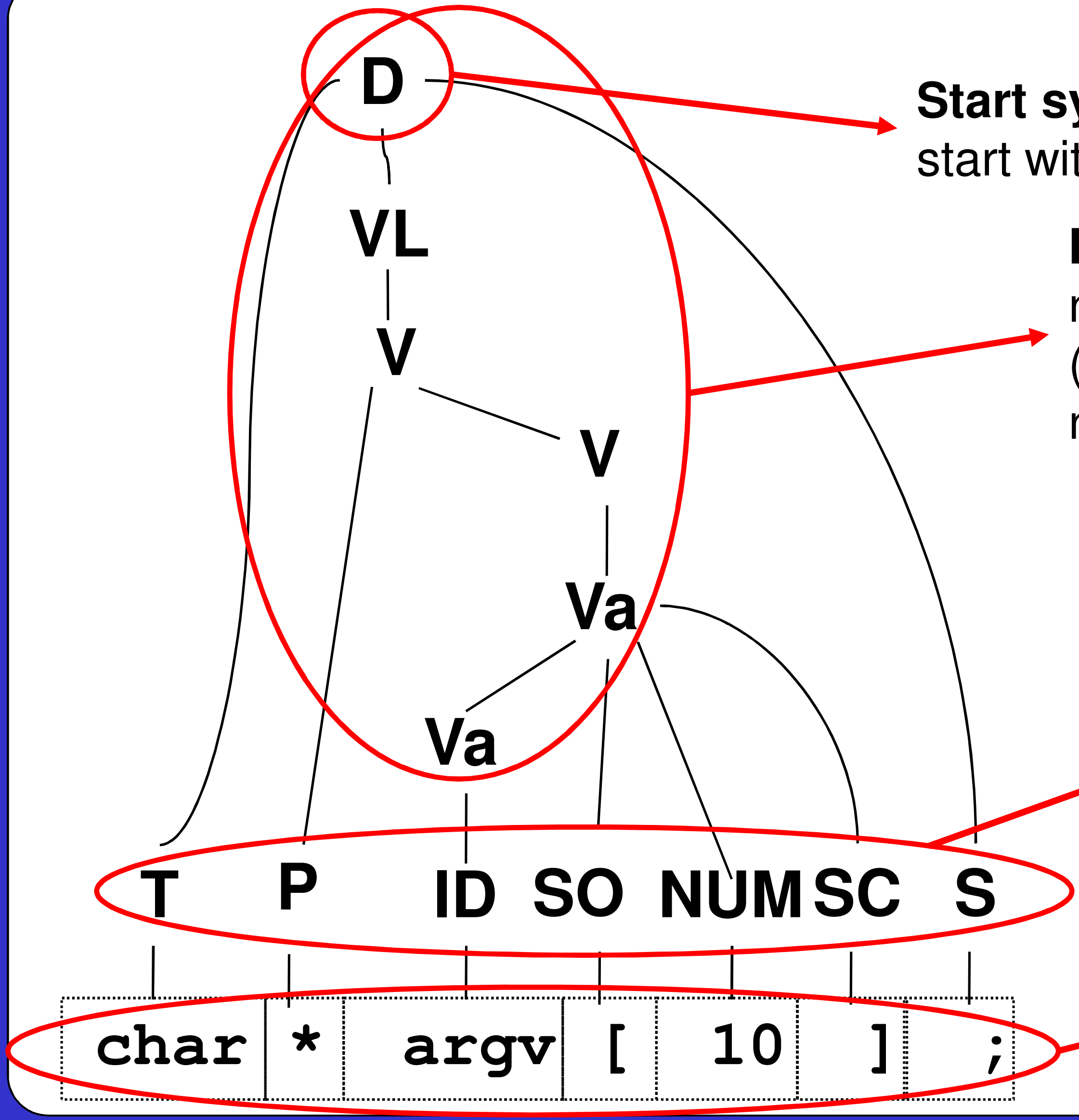
$Va \longrightarrow Va\ SO\ NUM\ SC$

$Va \longrightarrow ID$

**Input string:**

char *argv[10];

# Terminals / Non-Terminals section



**Start symbol**
start with D;

**Non-Terminals**
non terminal D, VL, V, Va;
(Note that the Start symbol is a non-terminal)

**Terminals**
terminal T, P, ID, NUM;
terminal SO, SC, CM, S;
(Recognized by JFlex)

**Input Sequence**

## Rules section

- The Rules section contains one or more productions in the form:

  <non_terminal> **::=** Right_Hand_Side **;**

- where *Right_Hand_Side* is a sequence of 0 or more symbols.
- To each prodution, an action can be associated, which must be enclosed between {: and :}
  - Note: the action is executed right before the reduce operation takes place

- Example:

```
D ::= T VL S
        {: System.out.println("Declaration found"); :}
;
```

## Rules section (2)

- If more than one production exist for a given non-terminal, they must be grouped and separated by ‘|’.

- Es.

```
funz ::=    type ID RO VL RC S
                {: System.out.println("Function prototype"); :}
            | type ID RO VL RC BO stmt_list BC
                {: System.out.println("Function"); :}
 ;
```

- NB: the use of the “|” character generates two separates rules. It is important to remember that the code between {: and :} is executed only when a giver rule is matched.

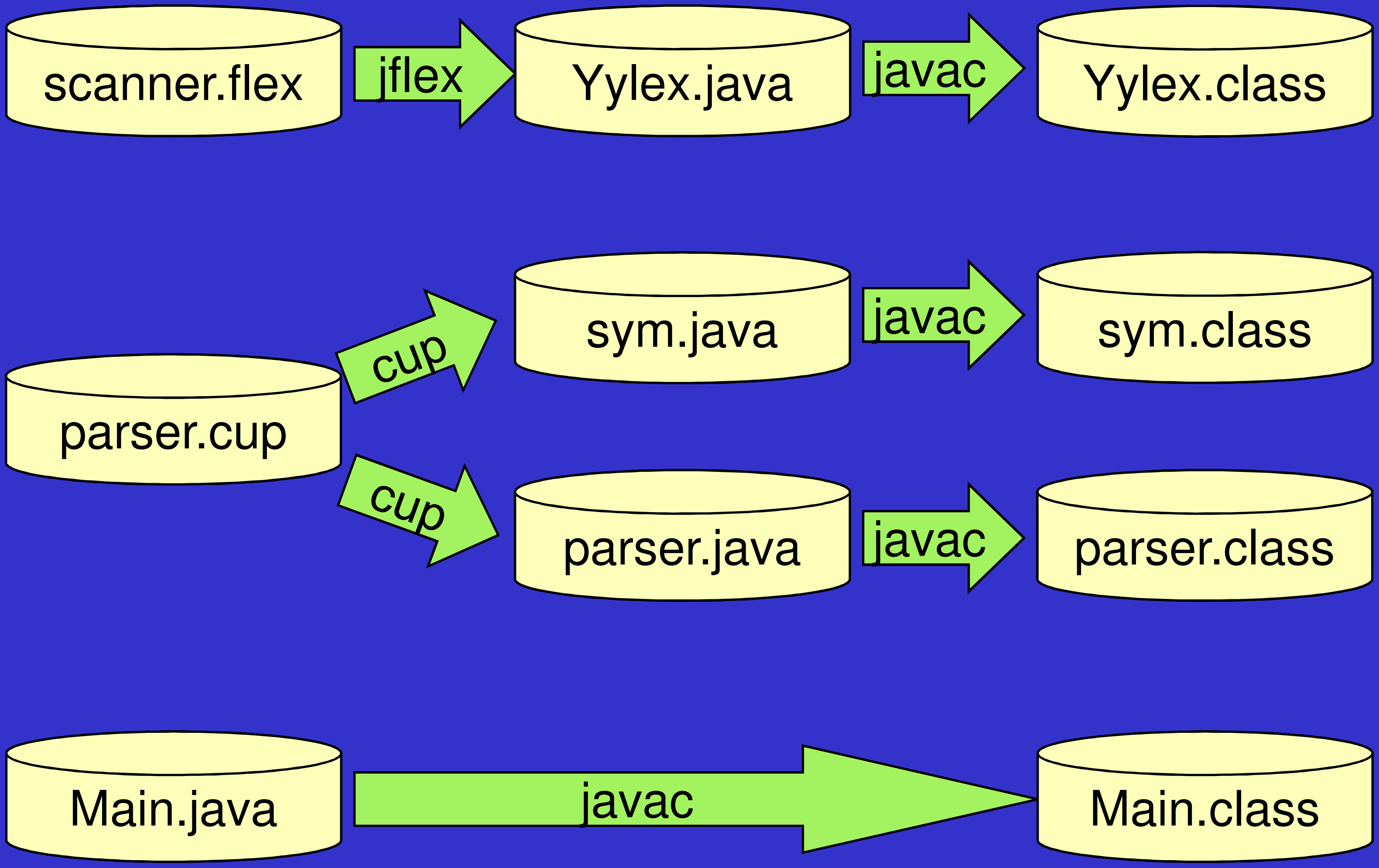
## Rules section : Example

```
import java_cup.runtime.*;

//Terminals / Non-Terminals Section
terminal T, P, ID, NUM, S, CM, SO, SC;
non terminal D, V, VL, Va;
start with D;

//Rule Section
D ::= T VL S ;

VL::= V
    | VL CM V ;

V ::= P V
     | Va ;

Va::= Va SO NUM SC
     | ID ;
```

**Productions:**

$$D \longrightarrow T\ VL\ S$$
$$VL \longrightarrow V$$
$$VL \longrightarrow VL\ CM\ V$$
$$V \longrightarrow P\ V$$
$$V \longrightarrow Va$$
$$Va \longrightarrow Va\ SO\ NUM\ SC$$
$$Va \longrightarrow ID$$

# Integrating JFlex and Cup



# Integrating JFlex and Cup

- Parser and scanner must agree on the values associated to each token (terminal)
- When the scanner recognizes a token, it must pass a suitable value to the parser. This is done by means of the Symbol class, whose constructors are:
  - public Symbol( int sym_id)
  - public Symbol( int sym_id, int left, int right)
  - public Symbol( int sym_id, Object o)
  - public Symbol( int sym_id, int left, int right, Object o)
  - The class Symbol can be found in the cup installation directory:
    - Java_cup/runtime/Symbol.java
- When a terminal is defined by means of the terminal keyword, Cup associated an integer value to that token.
  - This mapping is contained in the file sym.java generated by cup during the compiling process

# Integrating JFlex and Cup (2)

- If in the parser the following list of terminal symbols has been declared:

  ```
  terminal T, P, ID, NUM, PV, CM, SO, SC, S;
  ```

- They can be used inside the scanner and passed to the parser in the following way:

```
…
%%
…
%%
[a-zA-Z_][a-zA-Z0-9_]*   {return new Symbol(sym.ID);}
\[                       {return new Symbol(sym.SO);}
\]                       {return new Symbol(sym.SC);}
…
```

scanner.flex

## Scanner modifications

- Include the Cup library ( java_cup.runtime.* ) in the code section
- Activate Cup compatibility by means of the %cup directive in the Declarations section

```
import java_cup.runtime.*;
…
%%
%cup
…
%%
[a-z]+          { return new Symbol(sym.EL); }
“,”             { return new Symbol(sym.CM); }
```

**List $\rightarrow$ List CM EL**

**List $\rightarrow$ EL**

parser.cup

## The Cup parser

```
import java_cup.runtime.*;

terminal EL, CM;
non terminal List, EList;

start with EList;

EList ::= List      {: System.out.println("List found"); :} |
                    {: System.out.println("Empty list"); :}
;

List ::= List CM EL
;

List ::= EL
;
```

**List $\rightarrow$ List CM EL**

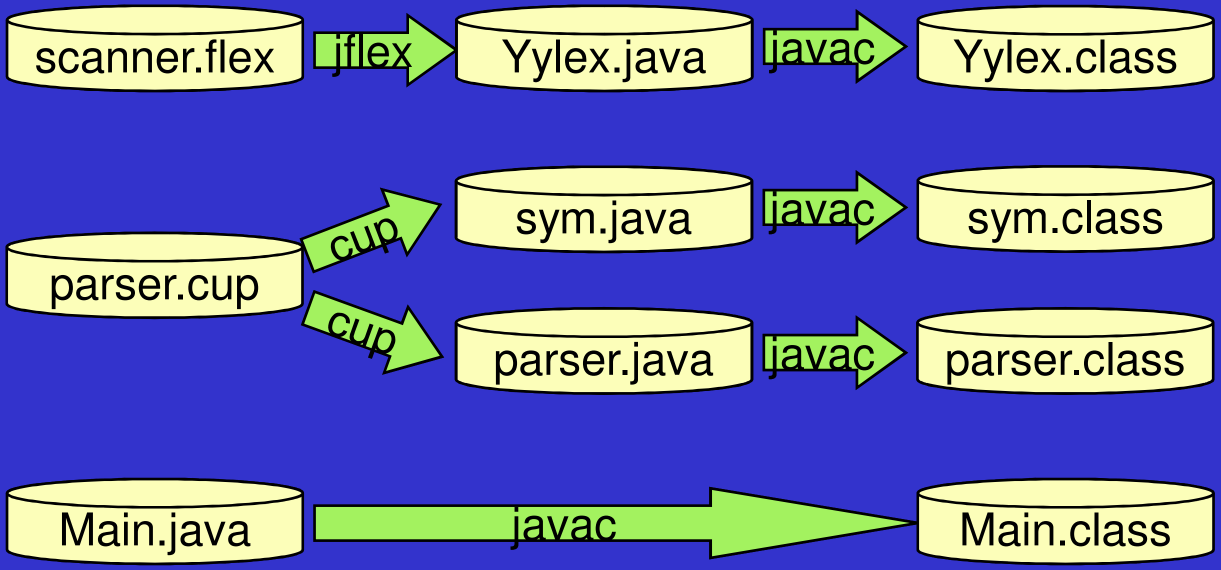
**List $\rightarrow$ EL**

Main.java

## Main

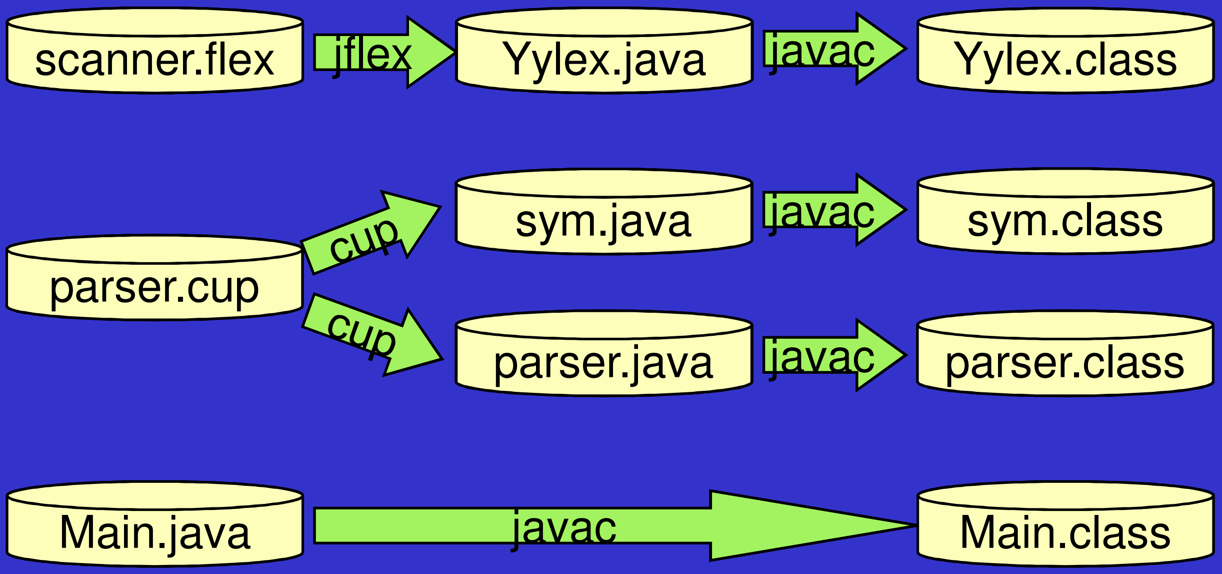
```
import java.io.*;

public class Main {
    static public void main(String argv[]) {
            try {
            /* Instantiate the scanner and open input file argv[0] */
                Yylex l = new Yylex(new FileReader(argv[0]));
                /* Instantiate the parser */
                parser p = new parser(l);
                /* Start the parser */
                Object result = p.parse();
            } catch (Exception e) {
                e.printStackTrace();
            }
     }
}
```

# Compiling



- jflex scanner.jflex
- java java_cup.Main parser.cup
  - In the case of shift/reduce or reduce/reduce conflits:
  - java java_cup.Main –expect <number_of_conflicts> parser.cup
  - java java_cup.MainDrawTree parser.cup
    - Can be used in LABINF or at home installing a modified version of the parser
    - The parse tree is drawn (useful for debugging)

# Compiling



- javac Yylex.java sym.java parser.java Main.java
  - Or javac *.java
  - For the compilation of all the files of the project
- java Main <file>
  - To execute the program using <file> a input