COMPILERS

1) JFlex (SCANNER [Lexical analyzer])

LEXICON (order of letters; using **JFlex** [lexical analyzer]) + **SYNTAX** (order of words; using **Cup** [syntax & semantic analyzer]) + **SEMANTIC** (meaning). JFlex transforms regexp in Java Program (scanner):

Summary of **REGEXP** (Regular Expressions) **in JFlex**:

- Letters and Numbers in the input string are described by the characters themselves (the regexp **`val1`** matches the input sequence **`'v' 'a' 'l' '1'`**)
- Non alphabetical chars must be written in quotation marks (to avoid the meaning of operators) [`xyz"++"` matches the input sequence `'x' 'y' 'z' '+' '+'`] or preceded by \ (`xyz\+\+` matches the input sequence `'x' 'y' 'z' '+' '+'`)
- **`[0123456789]`** is a character class (this expression matches a digit in input); **`[0-9]`** has the same meaning; **`[a-zA-Z0-9]`** matches lowercase char, uppercase char and digit
  - ⚠ To have the character '**`-`**' we need to put it first (otherwise it will be a separator between a range)
- `[^0-9]` means not a digit (exclusion), while ~ "`*/`" means all characters except from `*/`
- '**`.`**' identifies all the chars except new line (`\n` or `\r` or `\r\n`) [`\t` is a tab]
- `[0-9]+` identifies 1 or more digits at once, while `[0-9]*` identifies nothing, 1 or more digits
- a**`b?`**c means both `ac` or `abc` (**`b`** is optional)
- a**`b{3}`**c means a**`bbb`**c (**`b`** is repeated 3 times), while a**`b{2,4}`**c means a**`bbc`**, a**`bbbc`**, a**`bbbbc`** (**`b`** is repeated between a range of 2-4)
- `ab|cd` means `ab` or `cd`

Some examples:

- Unsigned integer
  - [0-9]+
- Unsigned integer without leading zeros
  - [1-9][0-9]*
- Signed integer
  - ("+"|"-")? [0-9]+
- Floating point number
  - ("+"|"-")? ([1-9][0-9]* "." [0-9]*) | ("." [0-9]+) | (0 "." [0-9]*)

⚠ The regexp of a variable name is **`[_a-zA-Z][_a-zA-Z0-9]*`**

A **JFlex source file** has 3 sections separated by %%:

Code section → all the code lines in this section are copied without modification in the generated scanner (as import statements of Java libraries)

%%

Declarations section → to simplify complex or repetitive regexp, we can define variables to contain sub-regexp (es. **`integer = [+-]?[1-9][0-9]*`**) and we can use these variables in the rules section using them between **{...}**:

```
{integer} {
      System.out.print("integer found\n");
}
```

⚠ We can also have Java code included in the declarations section by writing it between **`%{ code }%`**

%%

Rules section → as we said before, each regexp is associated to an action/rule (see before with integer) that is executed when the input matches the regexp (each action/rule is Java code [see above])

SCANNER METHODS & fields accessible in actions:

- `string` **`yytext()`** → returns the matched string
- `int` **`yylength()`** → returns number of matched characters
- `char` **`yycharat(int pos)`** → returns the `char` at position `pos`
- `int` **`yyline`** → current line of input file (value only if **`%line`** directive is declared)
- `int` **`yycolumn`** → current column of input file (value only if **`%column`** directive is declared)
- `int` **`yychar`** → current char count in input [from 0] (value only if **`%char`** directive is declared)

Example:

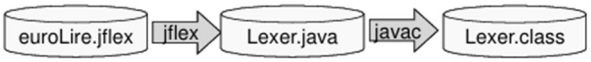
```
%%
%class Lexer
%standalone

curr    =   [1-9][0-9]*"."[0-9][0-9] | 0?"."[0-9][0-9]
Int     =   0 | [1-9][0-9]*
%%
{curr}          { System.out.println( "Curr: "+ yytext() ); }
{int}           { System.out.println( "Int: " + yytext() ); }
```

INPUT	OUTPUT
0.02	Curr: 0.02
.10	Curr: .10
2000.30	Curr: 2000.30
1.50	Curr: 1.50
15000	Int: 15000

⚠ We have:

- **`%standalone`** → generates the main method (which accepts as input the list of file to be scanned) [default behavior is print the unmatched chars to `stdout`]
- **`%class Lexer`** → generate class named `Lexer.java`



- Compiling steps:

```
jflex euroLire.jflex
javac Lexer.java
java Lexer <nome_file_1> … <nome_file_n>
```

⚠ The matched sequence is the one with higher length matched (> number of chars matched) and not only the sequence with all the chars matched; if we have 2 sequence with same length of chars matched, we choose the first defined **(2 important rules)**

⚠ Example of regexp of a comment "`// commento`" is **`\/\/.*`**

⚠ Da un file.jflex se voglio ottenere lo scanner ed eseguire il mio txt nella stessa cartella tramite lo scanner, devo usare la sequenza di comandi: **`jflex file.jflex ; javac Calc.java ; java Calc file_to_analyze.txt`**

Esempio (trasformare codice C in HTML, aggiungendo i colori ai singoli testi a seconda di che costrutto sono):

```
%%

%standalone
%class Calc

%init{ // -> codice da eseguire prima di tutto
    System.out.println("<HTML>");
%init}
%eof{ // -> codice da eseguire dopo tutto
    System.out.println("</CODE></BODY></HTML>");
%eof}

nl          =  \n|\r|\r\n
letter  = [a-zA-Z]
number  = [0-9]*
```

```
other   = [^a-zA-Z0-9\*/]
type    = "int"|"float"|"char"|"double"
header  = "#include <"{letter}*".h>"
construct = "if"|"else"|"return"|"for"|"while"|"break"|"continue"|"switch"
comment = "/\*"({letter}|{number}|{other})*"*/"
bracket = [\{\}]

%%

{number} {
    System.out.println("<FONT COLOR=\"#FF0000\">" + yytext() + "</FONT>");
}

{type} {
    System.out.println("<FONT COLOR=\"#0000FF\">" + yytext() + "</FONT>");
}

{header} {
    System.out.println("<FONT COLOR=\"#00FF00\">" + yytext() + "</FONT>");
}

{construct} {
    System.out.println("<FONT COLOR=\"#0000FF\">" + yytext() + "</FONT>");
}

{comment} {
    System.out.println("<FONT COLOR=\"#C0C0C0\">" + yytext() + "</FONT>");
}

{bracket} {
    System.out.println("<BR>" + yytext() + "<BR>");
}

{nl}|" "|\t  {;}

.           {System.out.print(yytext());} // -> è il default case
```

Let's see an EXAMPLE with STATES and **%caseless**:

```
%%

%standalone
%caseless                // tratta maiuscolo e minuscolo uguali
%class Calc

%{                       // attributi della classe Calc (mi servono per printare dopo i results)
    int totalTags = 0;
    int tableTags = 0;
    int h1Tags = 0;
    int h2Tags = 0;
    int h3Tags = 0;
    int h4Tags = 0;
%}

%eof{
    System.out.println("\nTotal number of tags: " + totalTags);
    System.out.println("Total number of table tags: " + tableTags);
    System.out.println("Total number of h1 tags: " + h1Tags);
    System.out.println("Total number of h2 tags: " + h2Tags);
    System.out.println("Total number of h3 tags: " + h3Tags);
    System.out.println("Total number of h4 tags: " + h4Tags);
%eof}

%state COMMENT
%state TAG
```

```
%%

// Regole generali
<YYINITIAL> {

    "<!--"  {
        yybegin(COMMENT);
    }

    "<"     {
        yybegin(TAG);
        totalTags++;
        System.out.print("<");
    }
    [^<]+   {
        System.out.print(yytext()); // testo generico
    }
}

// Regole COMMENT
<COMMENT> {
    "-->"   {
        yybegin(YYINITIAL); // fine commento
    }
    .|[\n\r]  {
        /* Ignora contenuto commento */
    }
}

// Regole TAG
<TAG> {
    "table"([ \t\n\r][^>]+)?     { tableTags++; System.out.print("table"); }
    "/table"    { tableTags++; System.out.print("/table"); }
    "h1"([ \t\n\r][^>]+)?        { h1Tags++; System.out.print("h1"); }
    "/h1"       { h1Tags++; System.out.print("/h1"); }
    "h2"([ \t\n\r][^>]+)?        { h2Tags++; System.out.print("h2"); }
    "/h2"       { h2Tags++; System.out.print("/h2"); }
    "h3"([ \t\n\r][^>]+)?        { h3Tags++; System.out.print("h3"); }
    "/h3"       { h3Tags++; System.out.print("h3"); }
    "h4"([ \t\n\r][^>]+)?       { h4Tags++; System.out.print("h4"); }
    "/h4"       { h4Tags++; System.out.print("/h4"); }
    [^>]+       { System.out.print(yytext()); } // altri contenuti dei tag
    ">"         { System.out.print(">"); yybegin(YYINITIAL); } // fine tag
}
```

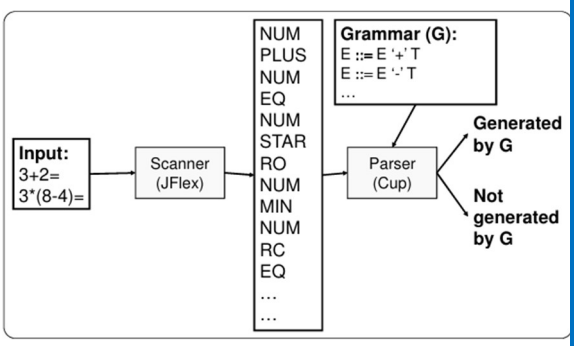
⚠ **STATES** allow you to define different behaviors for the lexer depending on the context (es. you can have a state to recognize HTML comments and another to recognize tags):

- Define states with `%state STATE_NAME`
- Change the lexer's state with `yybegin(STATE_NAME);`
- Each state has its own set of rules

2) CUP (PARSER [Syntax analyzer])

Given a non-ambiguous grammar and a sequence of input symbols, a **PARSER** (**Syntax analyzer**) is a program that verifies whether the sequence can be generated by means of a derivation from the grammar (is a program capable of associating to the input sequence the correct parse tree). As we saw in theory, parsers can be:

- TOP-DOWN (parse tree built from the root to the leaves)
- **BOTTOM-UP** (parse tree built from the leaves to the root) [**CUP**]



The difficult part is **writing the CF** (Context-Free) **GRAMMAR** $G = (T, NT, S, PR)$ = (Terminal symbols, Non-terminal symbols, Start symbol [$\in NT$], Production Rules [$PR: NT ::= T \mid NT$]).

As we saw in theory, the bottom-up parsing is **SHIFT/REDUCE**:

- stack (initially empty) is used to keep track of symbols already recognized
- T symbols are pushed in the stack (**shift**), until the top of the stack contains a handle (*rhs* [right hand side] of a production): the handle is substituted by the corresponding NT (**reduce**) [reduce may only be applied to the top of the stack]
- SUCCESS only when at the end of the input stream the stack contains only S (start symbol)

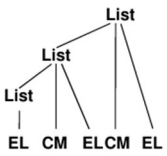
Input String:
a1 , a2 , a3

Scanner:
a1 , a2 , a3 → EL CM EL CM EL

Recursive Left Grammar

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

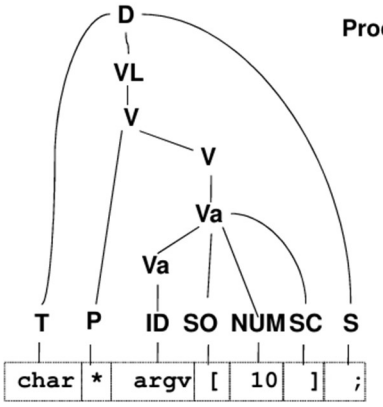
Parse Tree



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

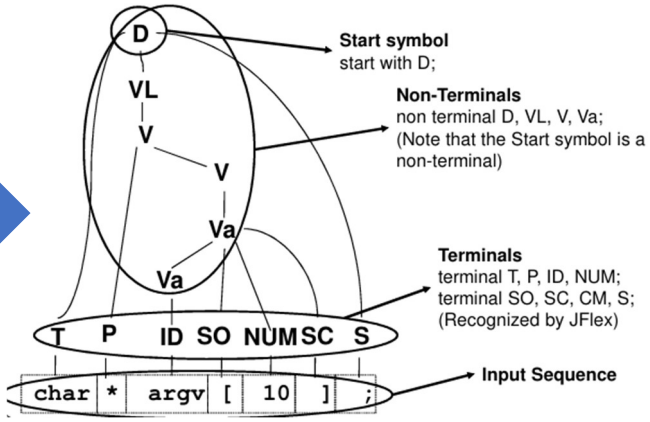
CUP (Java Parser): a CUP file has syntax similar to Java programs; it can be divided in sections (**setup, T & NT symbols, precedences, rules**):

- **Setup** → contains all the directives needed for the parser (`import java_cup.runtime.*;`) + user ridefinitions
- **T & NT section** → contains the definitions of Terminals (passed by JFlex), Non-terminals and S (start symbol [root of the parse tree; only 1 occurence is allowed])



Productions (grammal rules):
D → T VL S
VL → V
VL → VL CM V
V → P V
V → Va
Va → Va SO NUM SC
Va → ID

Input string:
char *argv[10];



- **Rules section** → contains 1 or more productions as `NT ::= rhs` (*rhs* = sequence of 0 or + symbols); to each production, we can associate an action (as JFlex) [es. `D ::= T VL S {: System.out.println(…); :} ;`]. If we have **more production for a single NT**, we have to group them ans separate them by '|'. 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 → T VL S
VL → V
VL → VL CM V
V → P V
V → Va
Va → Va SO NUM SC
Va → ID

⚠ **INTEGRATING JFlex and CUP**: they must agree on the values of each **T** symbol; when JFlex recognizes a T, it must pass a suitable value to the parser through 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)`

We use **`return new Symbol(sym.ID_symbol)`** in an action of a rule of JFlex; but to do so, we must include the **`%cup`** at the beginning of the file.jflex. Reusing the example before, we see the **scanner.jflex** and the **parser.cup** files:

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

List → List CM EL
List → EL

```
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 → List CM EL
List → EL

We have also to copy and paste the **Main.java** that the professor gave us, in all our project:

```
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();
        }
    }
}
```

How to compile all these file from terminal? `jflex scanner.jflex; java java_cup.MainDrawTree parser.cup; javac *.java; java Main name_file`

⚠️ If we have an ambiguos grammar (2 different parse tree for the same grammar) [shift/reduce or reduce/reduce conflicts], we have to use `java java_cup.Main -expect number_of_conflicts parser.cup`

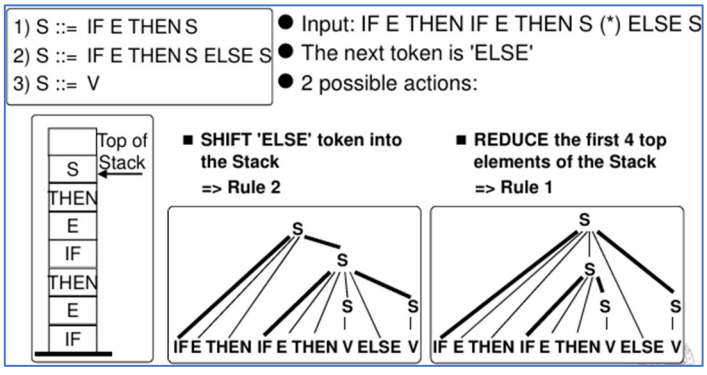
SE VUOI UN ESERCIZIO COMPLETO (SENZA PRECEDENZE) SU JFLEX + CUP, prendi LAB03/es2

2.1) CUP ADVANCED USE [in EXAM we can have examples to copy and paste]

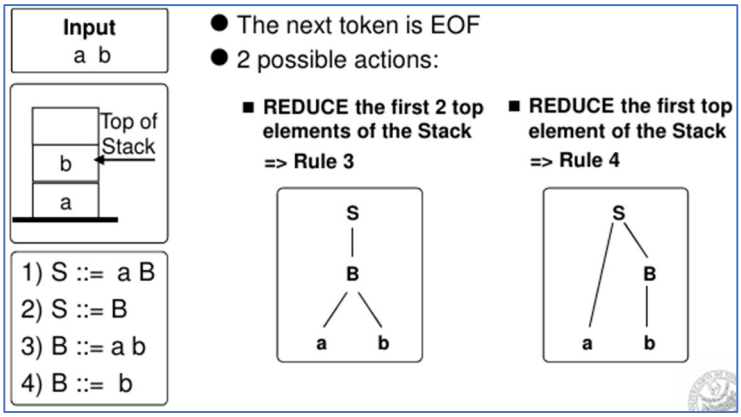
When the grammar is ambiguous, there can be **CONFLICTS**: so the parser must choose between 2 or more actions. So we can modify grammar or instruct parser to handle ambiguity (this option requires that the parsing algorithm is fully understood) [REMEMBER: a grammar is ambiguous if there is at least 1 sequence of symbols for which 2 or more parse trees exist]

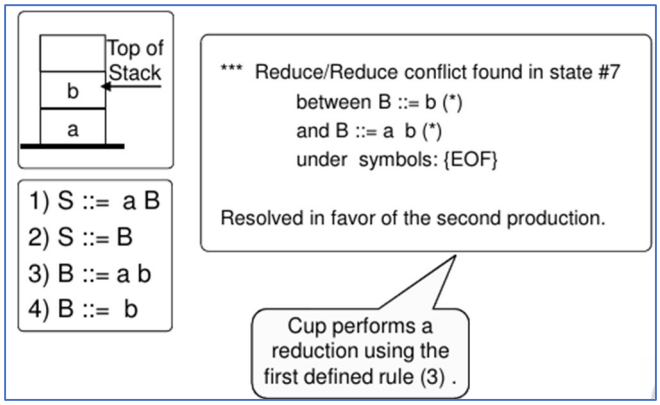
⚠️ In the LAB03, we have seen the ambiguity in if-else statements and we must have done something like this (use non-terminal symbol to split the "string" in 2 parts to know exactly when the 1st part is finished). Let's see this **SHIFT-REDUCE** conflict in the if-else statement in CUP (**Cup choose shift**):

```
S    ::= M | U
U    ::= 'if' C S
U    ::=  'if' C M 'else' U
M    ::= 'if' C M 'else' M
M    ::= ID '=' NUM ';' | ID '=' ID ';'
C    ::= '(' ID '==' NUM ')'
```



In the case of **REDUCE-REDUCE** conflict in CUP:





At the same time, **AMBIGUOUS GRAMMARS** can have **fewer** and **simpler** rules than non-ambiguous grammars:

```
Non-ambiguous grammar
S ::= E
E ::= E '+' T
E ::= E '-' T
E ::= T
T ::= T '*' F
T ::= T '/' F
T ::= F
F ::= '(' E ')'
F ::= INTEGER
```

```
Ambiguous grammar
E ::= E '+' E
E ::= E '-' E
E ::= E '*' E
E ::= E '/' E
E ::= '(' E ')'
E ::= INTEGER
```

But this grammar has 2 different behaviors with:

- left-associative operator → 1+2+3 = 3+3 = 6
- right-associative operator→ 1+2+3 = 1+5 = 6

⚠ The assignment operator ('=') and the power operator ('^') are right-associative

So in the case where the operator's associativity is not specified (es. **+**, *), we must provide disambiguating rules through **PRECEDENCE SECTION** [in shift-reduce conflict, the **action with highest precedence production is exectuted**; if precedence is the same, associativity is used: left-associativity → reduce, right-associativity → shift]:

```
terminal uminus;

precedence left PLUS, MINUS;  /* Low priority */
precedence left STAR, DIV;
precedence left uminus;     /* High priority */

start with E;

E ::=   E PLUS E
      | E MINUS E
      | E STAR E
      | E DIV E
      | MINUS E     %prec uminus
      | '(' E ')'
      | INTEGER
;
```

We can insert **USER CODE** directly in the parser through **4 DIRECTIVES**:

- **`init with {: … :}`** → code executed before anything (used to initialize something)
- **`scan with {: … : }`** → indicates to parser which procedure use to request next terminal to scanner (if we don't use JFlex)
  When CUP generates the java file that implements the parser, 2 classes are defined:
  - **`public class parser extends java_cup.runtime.lr_parser`** → parser is the java class that implements the parser and inherits different methods from the java_cup.runtime.lr_parser class
  - **`class CUP$parser$actions`** → where declared grammar rules are translated into a java program
- **`parser code {: … :}`** → the code is copied in the parser class; it's used to override parser methods (redefining **`report_error()`** function) and to define global data structures
- **`action code {: … :}`** → the code is copied in the CUP$parser$actions class; it's used to define procedures and variables to be used in the actions associated to the grammar (es. symbol table [theory part])

We can insert **line number and column number** of the symbols to locate errors (we use `%line` and `%column` in the upper section and we refer to line and column with the variables **`yyline`** and **`yycolumn`**) [sx = JFlex; dx = CUP]:

```
import java_cup.runtime.*;
…
%%
%cup
%line
%column

%{
    private Symbol my_symbol(int type){
            return new Symbol(type, yyline, yycolumn);
    }
    private Symbol my_symbol(int type, Object value){        //Semantic analysis
            return new Symbol(type, yyline, yycolumn,value);
    }
%}
…
%%
[a-z]           { return my_symbol(sym.EL); }
','             { return my_symbol(sym.CM); }
```

Symbol constructors:
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)

```
import java_cup.runtime.*;

parser code {:
    public void report_error(String message, Object info) {
        StringBuffer m = new StringBuffer(message);
        if (info instanceof Symbol) {
            if (((Symbol)info).left != -1 && ((Symbol)info).right != -1) {
                int line = (((Symbol)info).left)+1;
                int column = (((Symbol)info).right)+1;
                m.append(" (line "+line+", column "+column+")");
            }
        }
        System.err.println(m);
    }
:}
```

As we also see in theory part, we can also use **`error`** symbol in order to enable the parser to continue execution when an error is encountered (es. **`ass ::= ID EQ E S | ID EQ error S;`**). How it works? (looks theory):

1) When an error occurs, the parser start emptying the stack until find a state in which **`error`** is allowed (in the es above, symbol sequences that can't be reduced as **`E`** are removed from the stack, until the terminal **`EQ`** is on top of stack)
2) **`error`** token is shifted in the stack
3) if the next token is ok, the parser resumes; else parser continue to discard token until an acceptable is found

```
stmt ::= exp ';'
        | compound
        | error ';'  {: System.err.println("Syntax error in statement"); :}
;

compound ::= '{' stmts '}'
                | '{' stmts error '}'  {: System.err.println("Missing ; before '}' "); :}
;

exp ::= …
        | '(' error ')'  {: System.err.println("Syntax error in expression"); :}
;
```

⚠ All'esame non dovremo usare error nella nostra grammar (eccetto se specificatamente richiesto per un caso specifico dove va usato error); un'altra **sintassi comune** (che ho già usando nel LAB03 è nelle definizioni di CUP usare `definition ::= /* empty */ | definition2 …`; ovvero usare la **`/* empty */`** se voglio tornare nulla [ε])

⚠ In fact, when a CUP parser finds an error, it doesn't terminate immediately the execution: calls **`syntax_error`** (defined in `java_cup.runtime.lr_parser class` → write "**Syntax error**" in `stderr`) → if error not managed by the parser through the default error symbol, the parser calls the **`unrecovered_syntax_error`** (write "**Couldn't repair and continue parse**" in `stderr` and stops) [we can also redefine this with `parser code {: … :}`]

EXAMPLEs of **LIST definitions**:

Define empty list → **`EL ::= ε | EL E`** (l'empty può essere scritto con il commento come sopra **`/* empty */`**)

Define list of at least 3 EL (>= 3) → **`LE ::= E E E | LE E`** (extendable to N elements)

Define list of at least 5 EL in **odd number** (ovvero dimensione di 5, 7, 9, …) → **`LE ::= E E E E E | LE E E;`**

List of at least 4 EL (separeted by commas) in even number that could be empty (0, 4, 6, 8, …):

`EL ::= ε | LE;`

`LE ::= E C E C E C E C E | LE C E C E;`

3) Attributes of Symbols

Attributes can be associated to each symbol; attributes can be:
- **SYNTHESIZED** = calculated from the values of the attributes of the node's children in the parse tree
- **INHERITED** = calculated from the values of the parents/siblings in the parse tree

A set of rules (to specify how attributes are calculated) is associated to each production.

SYNTHESIZED:

S-attribute grammar = grammar whose attributes are all synthesized; in this case, the values of all attributes can be calculated in bottom-up (from leaves to root of parse tree):

$E ::= E_1$ '+' T	`E.value = E1.value + T.value`
E ::= T	`E.value = T.value`
T ::= number	`T.value = number.value`

In CUP, each symbol in stack is an object of class Symbol and contains: **`sym`** (int id value), **`parse_state`**, **`left`** & **`right`** (2 int to pass line and column number from scanner to parser) and **`value`** (a object of class Object to handle semantics).

When we pass symbols to Cup, CUP must know the type of the semantic rule of each symbol (come in un normale linguaggio di programmazione: **`terminal tipo var1, var2…`**; or **`non terminal tipo var1, var2…`**).

We can view the value of a symbol in a production with the syntax: **`E ::= E:n1 PLUS T:n2;`** where **`n1`** and **`n2`** are the values of the symbols. So, in this way we can do operations in parsing and putting the result value in **`RESULT`** (an object related to the left side of a production where is the result value of the operation):

```
non terminal Integer E, T;
E ::= E:n1 PLUS T:n2 {:RESULT= n1 + n2; :}
        | E:n1 MINUS T:n2 {:RESULT= n1 -n2; :} ;
```

If we want to keep more results of operations in a single non-terminal object? We can use an array of Object[] in the declaration of the non-terminal [another solution can be writing a class that contains the required info]:

```
terminal RO, RC;
terminal String identifier;
terminal Integer Args;
non terminal Object[ ] Func;
non terminal goal;

goal ::= Func:a {:
      System.out.println( "Function name: " + a[0] + "Number of parameters: " + a[1] );
:} ;

Func ::= identifier:a RO Args:b RC {:
            RESULT = new Object[2];
            RESULT[0] = new String(a);
            RESULT[1] = new Integer(b);
:} ;
```

When we use the console to compile our program, we can insert a series of options (**PARSER DEBUGGING**):
- **`-dump_grammar`** → prints the list of terminals, non-terminals and productions of CUP
- **`-dump_states`** → prints the state graph
- **`-dump_table`** → prints the ACTION TABLE and the REDUCE TABLE
- **`-dump`** → prints all the informations above

To view these options in the terminal, we have to use the Debug mode in the **`Main.java`** instead of normal mode:

Normal mode:
```
Yylex l = new Yylex(new FileReader(file));
parser p = new parser(l);
Object result = p.parse();
```

Debug mode:
```
Yylex l = new Yylex(new FileReader(file));
parser p = new parser(l);
Object result = p.debug_parse();
```

INHERITED:

L-attributed grammar = attributes' values can be calculated by means of a depth-first visit of the parse tree; information propagates from left to right; CUP manages only L-attributed grammar. Example:

int a, b;

D → T L ';'	`L.type = T.type`
L → L_1 ',' id	`L1.type = L.type` `put(id.name,L.type)`
L → id	`put(id.name,L.type)`
T → 'integer'	`T.type = type_int`

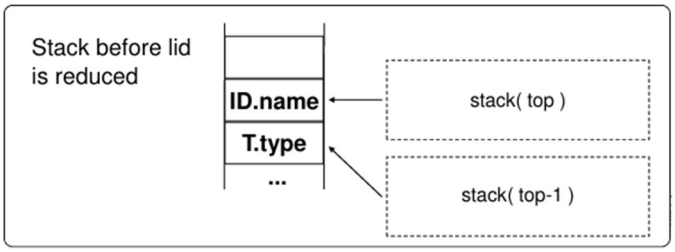
In a bottom-up parser, memory is not allocated in the semantic stack until the corresponding symbol is recognized; this is troublesome for handling inherited attributes; if the grammar is L-attributed, this can be solved using markers ("**marker**" = **non-terminal symbol expanded with *ε* symbol**).

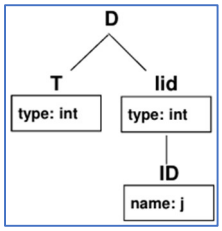
To access to the semantic values stored in the stack in a given position, use function `Object stack(int position)` [where `stack(0)` is the semantic value associated with the symbol on **top** of stack; `stack(n)` is the semantic value associated with the symbol in the position **top+n** of stack].

```
parser code {:
……..
public Object stack ( int position){
   // returns the object at the specified  position
   // from the top (tos) of the stack
      return(((Symbol)stack.
      elementAt(tos+position)).value);
}
……..
:}
```

So, how to calculate inherited attributes? Example:

D → T lid S lid → ID	`lid.type = T.type` `put(ID.name,lid.type)`

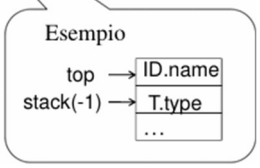




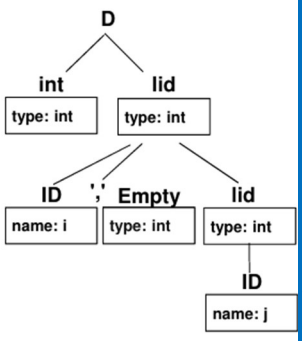
The attribute `type` of `lid` is inherited and its value is in the semantic stack (in the position of `T`) before `lid` is created; but it's beyond the semantic scope of the `lid` production.

So, with the assumption that `lid` symbol is always preceded by a `type` identifier, we can do this:

```
lid ::= ID:name {:
        String type = (String) parser.stack(-1);
        RESULT = new String (type);
        put(name, RESULT);
:} ;
```



But if we add the usual rule of the lists `lid ::= ID CM lid`, it's not true the assumption made above (that `lid` is always preceded by a `type` idenfier): if I have `lid ::= ID`, the symbol preceding `ID` in stack before reducing is `CM`. So we can add an empty rule (marker) to ensure that rule `lid ::= ID` is preceded by a `type` semantic rule.



From a code point we can see this in the example below:

```
lid ::= ID:name {:
      RESULT = (String) parser.stack(-1);
      put(name, RESULT);
:} ;

lid ::= ID:name CM Empty lid {:
      RESULT = (String) parser.stack(-1);
      put(name, RESULT);
:} ;

Empty ::= {:
      RESULT = (String) parser.stack(-2);
:} ;
```

GRAMMAR

```
D   ::= T lid S ;
Lid ::= ID CM Empty lid
      | ID ;
Empty ::= /* ε */ ;
```

To avoid explicit introduction of a non-terminal with an empty production (marker), we can use in the rhs of the production an **INTERMEDIATE ACTION** (these are automatically substituted with a non-terminal symbol). Example (from marker [sx] to intermediate action [dx]):

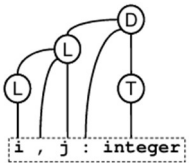
```
lid ::= ID:name CM Empty lid ;
Empty ::= ;
```

→

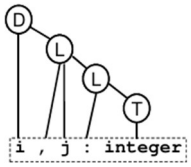
```
lid ::= ID:name CM {:
      RESULT = (String) parser.stack(-2);
    :}
    lid {:
      RESULT = (String) parser.stack(-1);
      put(name, RESULT);
:} ;
```

⚠ We can also avoid inherited attributes changing the grammar (obvious!!!):

D → L ':' T
T → **integer** | **real**
L → L ',' **id** | **id**



D → **id** L
L → ',' **id** L | ':' T
T → **integer** | **real**

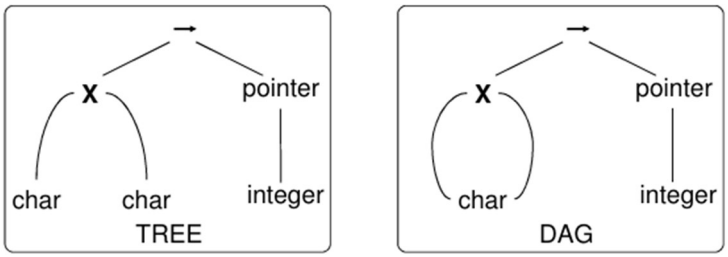


4) TYPE CHECKING

Type Checking is the process used for the verification of types constraints (can be done at ***compilation time*** [**static check** → 1 of main semantic tasks done by a compiler] or execution time [dynamic check]).
Using **base types** (*primitive* types [with also **`void`** and **`type_error`**) **+ type-constructors**, each expression can become a **TYPE EXPRESSION** (that is a base type or is derived applying a type constructor to another type-expression). Es:

Declaration:	Type expression:
`char v[10]`	`array(10,char)`
`struct { int i; char s[5]; }`	`struct((i x int) x (s x array(5,char)))`

As we saw in theory part, a **function** maps an element of a domain to another (es. **`int* f(char a, char b)`** is represented through a type expression: **`(char x char)`** → **`pointer(int)`**); we can represent type expressions through trees/DAGs:



Constructing a **type checker** for C language could be like this:

D → T Vl ';'	`Vl.type=T.type`
Vl → V	`V.type=Vl.type`
Vl → Vl_1 ',' V	`V.type=Vl.type`

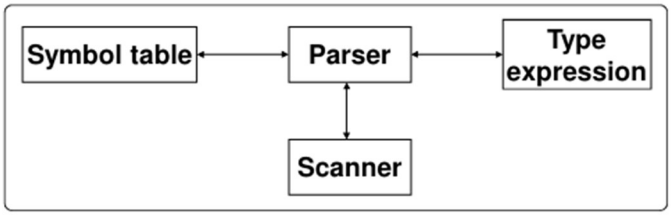
V → P id A	`P.base=V.type`
	`A.base=P.type`
	`put(id.value,A.type)`

P → ε	`P.type=P.base`
P → P_1 '*'	`P.type=pointer(`P_1`.type)` P_1`.type=P.base`
A → ε	`A.type=A.base`
A → A_1 '[' num ']'	`A.type=array(num.val,`A_1`.type)` A_1`.type=A.base`

Structural equivalence → 2 expressions are equals if:
- belong to the same primitive type
- are based on the application of the same types constructors to equivalent types

⚠ Using a tree representation for type expressions, we can use a recursive visit algorithm to verify equivalence

A **TYPE CHECKER** has this structure:



SYMBOL TABLE → associates values to names to have info about the identifier outside of context of declaration.
This table can be implemented with various data structures (es. **`HashMap`**). Es:

```
import java.util.HashMap;
// Initializing the table
HashMap<String, String> symTable = new HashMap<String, String> ();
```

```
// Inserting entries: int a; float b;
symTable.put("a", "int");
symTable.put("b", "float");

// Get the value related to key "a"
String tipo = (String) symTable.get("a");
System.out.println(tipo);

// Deleting entry
symTable.remove("a");

// Deleting all entries
symTable.clear();
```

How to implement type expressions?

```
public class te_node {
  public int tag;    // BASE, ARRAY, POINTER, …
  public int size;   // Number of elements in array
  public int code;   // Base type: INT, CHAR, FLOAT, …

  // Only for structs
  public String name;// Struct name

  // Left and right children
  private te_node left, right;

  public static te_node te_make_base(int code);
  public static te_node te_make_pointer(te_node base);
  public static te_node te_make_array(int size, te_node base);

  // Only for structs and functions
  public static te_node te_make_product(te_node l, te_node r);
  public static te_node te_make_name(String name);
  public static void te_cons_struct(te_node str, te_node flds);
  public static te_node te_make_fwdstruct(String name);
  public static te_node te_make_struct(te_node flds, String n);
  public static te_node te_make_function(te_node d, te_node r);
}
```