Syntax parser - BISON

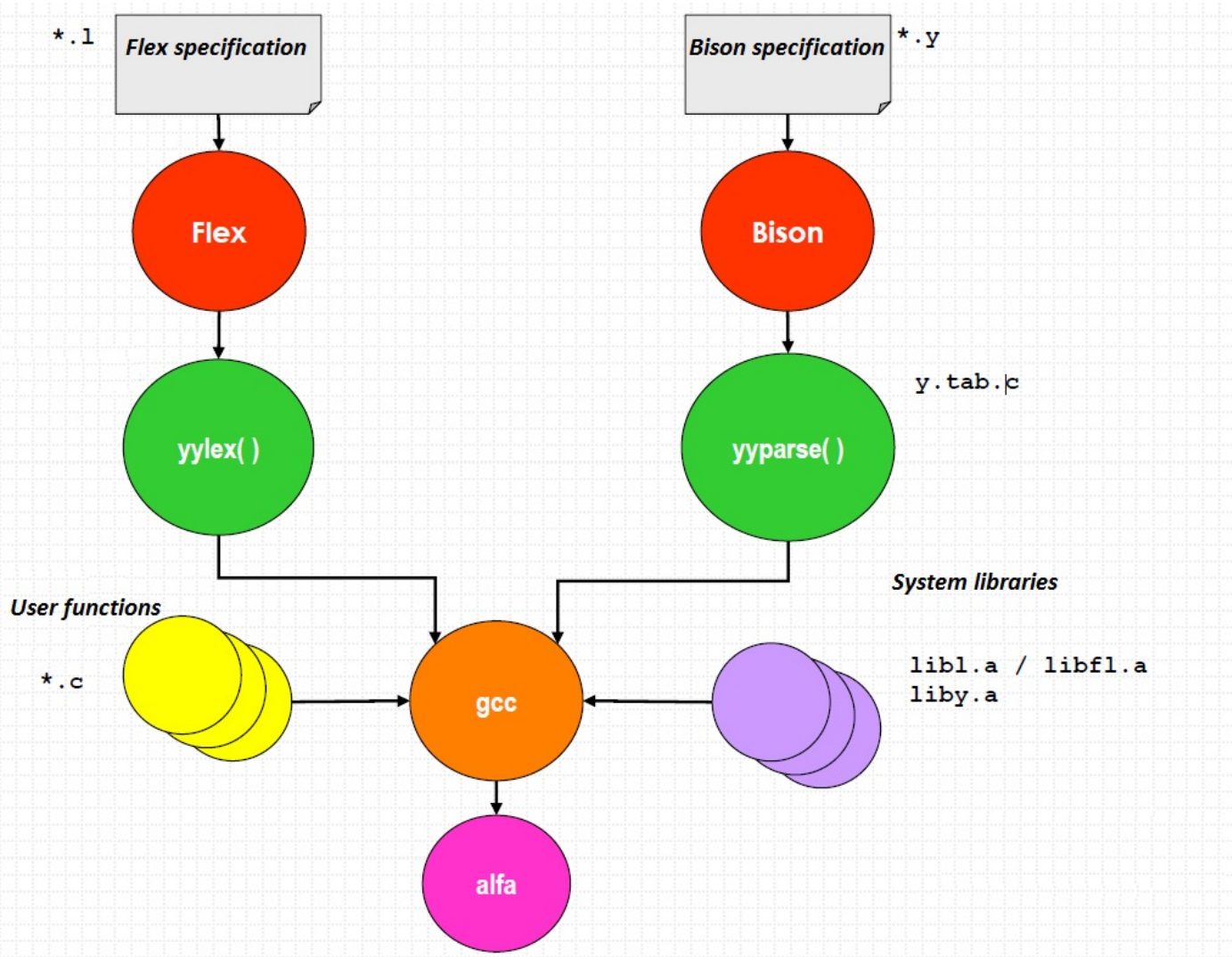
- Introduction
- How to use Bison with Flex
  - How to build an "alfa" target program
  - Communication between the functions main(), yylex() and yyparse()
- Bison specification file
  - File structure
  - Bison section definitions
  - Rules section
  - User functions section

Introduction

- Bison is a LALR(1) syntax parser generator

- We usually use Bison along with Flex
  - Flex generates a morphological/lexical parser: yylex()
  - Bison generates a syntax parser: yyparse()

How to build a target ALFA program I



How to build a target ALFA program II

- Flex builds a **yylex()** function to apply a morphological parse based on a corresponding specification file. The file where the yylex() function is placed is called **lex.yy.c**

- Bison builds a **yyparse()** function to apply a syntax parse from a file based on a corresponding specification file. The yyparse() function is placed in the file **y.tab.c**

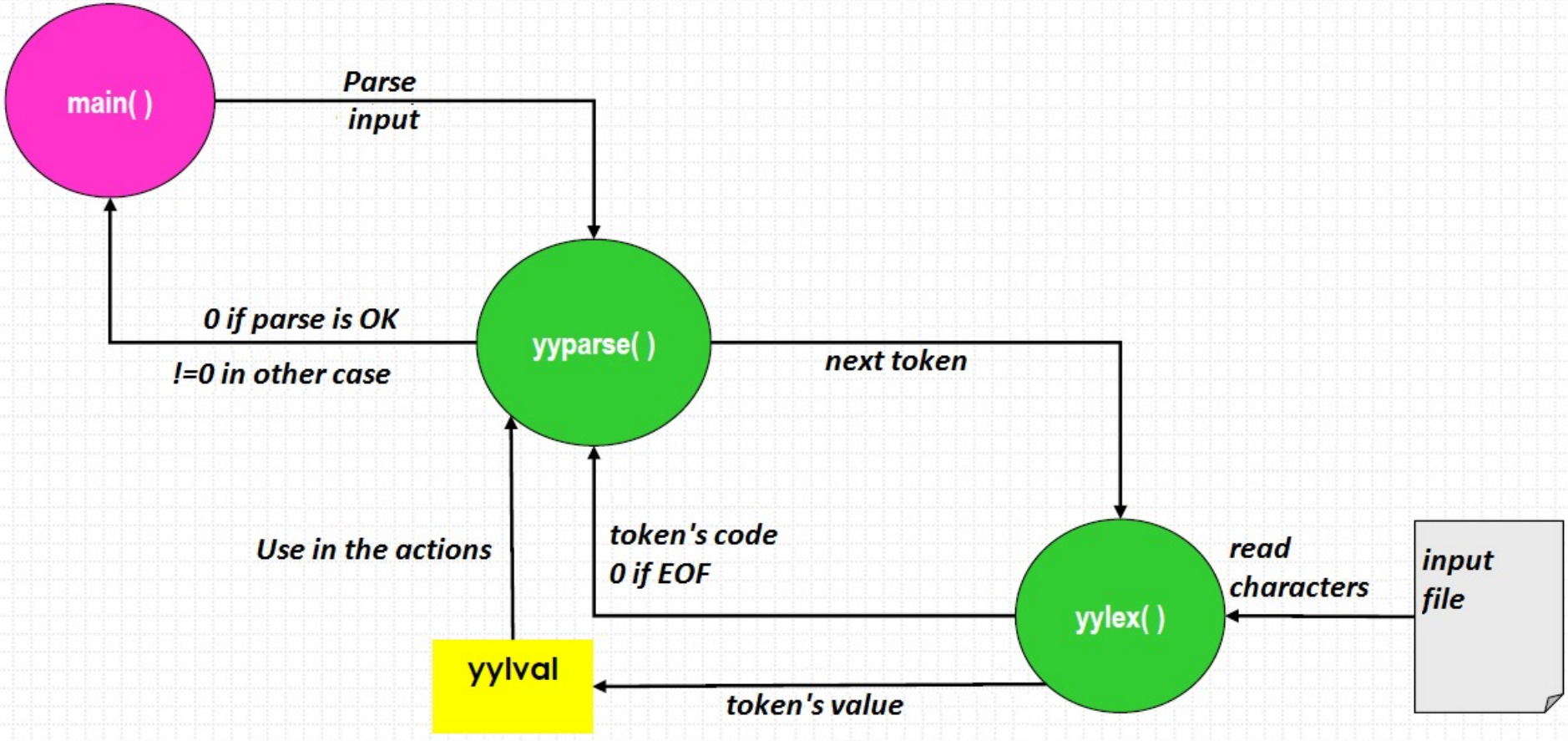
- The **user functions** include functions to provide assistance to code generation functions, or functions invoked by Flex and Bison, for example **yywrap()** or **yyerror(),** and also the main function invoked by the syntax parser to perform a comparison

How to build a target ALFA program III

- The **system libraries** provide a set of simple versions of the functions that are invoked by Flex and Bison, as well as a minimum version of the main function that invokes the lexical parser or the syntax. If the user provides these functions, there is no need to use the libraries. Depending on the operative system, these libraries may exist or not, then to facilitate the code's portability, it is recommended that the user provides their own functions version, or to set the properties of the Flex and Bison tools in an appropriate way

- To obtain the target alfa program, these configuration files are compiled and linked

Communication between main(), yylex() and yyparse() I



Communication between main(), yylex() and yyparse() II

- The **main()** function invokes the function to perform the syntax parse **yyparse()** that returns a 0 if the analysis ends with success, i.e., if the input is synctactically correct. In other case it returns a value distinct to 0

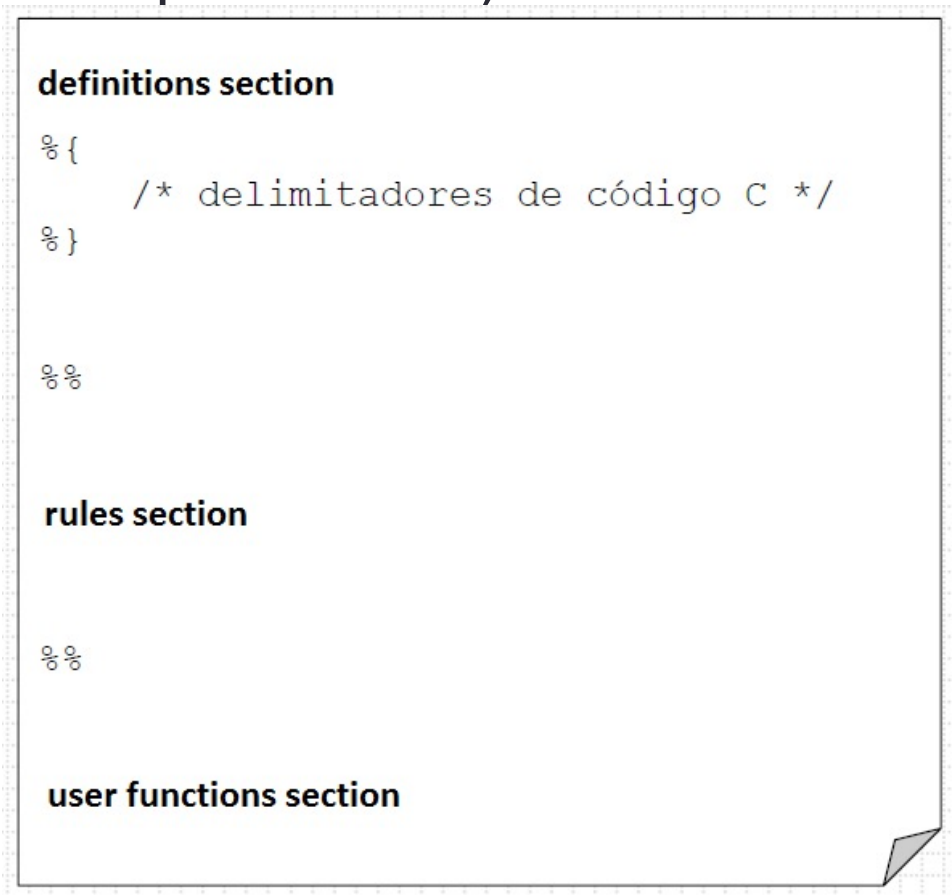
- The function **yyparse()** asks the function **yylex()** the input tokens and checks if they conform a valid construction regarding the input gramar rules specified in the corresponding Bison file. When a syntact error is detected, the **yyerror()** function ends and the analysis process returns a distinct value to 0

Communication between main(), yylex() and yyparse() III

- The function **yylex()** reads an input file to identify the tokens described in the corresponding specified file. Each time a function **yylex()** returns a token to the syntact parser, if the token has an associated value, **yylex()** stores this value in the **yylval** variable before it ends. For example, an identifier of a variable has a token identification code and an associated value or attribute, that it is the identifier lexema, too. Nevertheless, it can be considered that a parentheses doesn't have any value or attribute. The function **yyparse()** uses the value of the variable **yylval** in the actions of the grammar rules

Bison specification file structure

- A file with a Bison specifications has three sections separated with lines that have a %% (it is very similar to the input Flex specification)



Bison definitions section I

- In this section you can:
  - Include C code blocks that will be literally copied in the output file
  - Define the type of the **yylval** variable
  - Define the terminal and non-terminal symbols of the grammar
  - Define the grammar main rule
  - Define the associativity and precedence of the operators

- C code blocks:
  - This block includes:
    - Macro definitions
    - Variable declarations
    - Function declarations
    - #include directives

- The content of this section is copied literally at the beginning of the **y.tab.c** file that Bison generates

Bison definitions section II

- **%union** declaration
  - By default, the yylval value to pass the semantical values by Flex to Bison is of int type. But usually, the semantic values of the tokens are of different types. For example, a token of type identifier has a semantical value of type char*, whereas a token of type numerical constant has a semantical value of int type. With the %union declaration a union C structure is indirectly defined with a field for each semantic value type

- For example, if the types of the semantic values are int and char*, then a structure as this is declared:

```
%union {
  char *string;
  int number;
}
```

Bison definitions section III

- **%token** declaration (i)
  - It is used to define the terminal symbols (tokens) of a grammar
  - The most simple way is:

```
%token TOKEN_NAME
```

  - A more complete way is:

```
%token <union_field> TOKEN_ID
%token <numero> TOKEN_NUMBER
```

  - Example:

```
%token IF
%token THEN
…
```

Bison definitions section IV

- **%token** declaration (ii)
  - From Flex, all the qualified tokens will be returned, for example:

```
[A-Z]+        { strcpy(yylval.cadena, yytext}; return ID; }
[0-9]+        { yylval.number = atoi(yytext); return NUM; }
```

Bison definitions section V

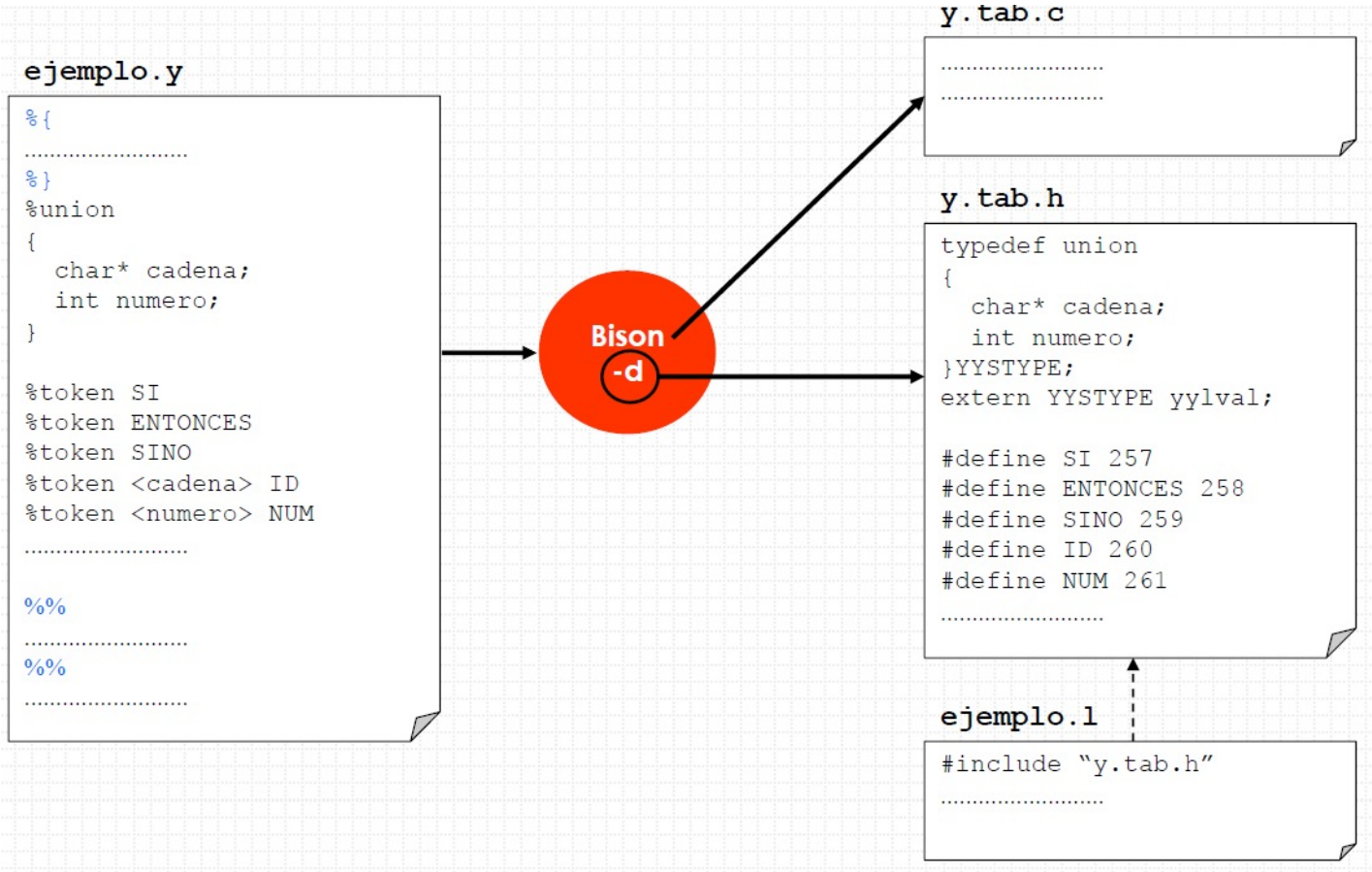
- **%token** declaration (iii)
  - Several tokens can be grouped in a line if they are of the same kind
  - There is no need to declare an only character, because they are implicitily declared in Bison with their corresponding ASCII value
  - From Flex, this kind of tokens are returned, for example:

```
"+"        {return '+'; } or {return yytext[0]; }
"("        {return '('; } or {return yytext[0]; }
";"        {return ';'; } or {return yytext[0]; }
```

- The **compilation of a *.y** specification file is performed with the **option -d** to make Bison generate the y.tab.h file that contains the tokens definitions. Next, this file is included in the Flex specification with the purpose that Bison and Flex share the tokens definitions

Bison definitions section VI

- Example of the **y.tab.h** file



Bison definitions section VII

- **%type** declaration
  - It will be used to specify multiple types of values when the declaration **%union** has been included
  - It allows the type delcaration of the **non-terminal** symbols. It is not required to declare the non-terminal symbols that **do not** have an assigned value with the $$ (see the grammar rules section)
  - This %type declaration has this format:

    %type <union_field> non_terminal_name
  - Several non-terminal symbols may be grouped in a line if they are of the same type

Bison definitions section IX

- Operators precedence
  - When an expression contains several operators, the precedence of the operators determines the order to evaluate the individual operators. For example, the expression x + y / z is evaluated as x + (y / z)
- Operators associativity
  - When an operand is found between two operators with the same priority order, the associativity of the operands determines the order in which the operations are executed. For example, the expression x * y / z is evaluated as (x * y) / z

Bison definitions section X

- Declarations **%left** and **%right**
  - They allow the declaration of the associativity of the grammar operators
  - The declaration **%left** specifies an associativity from the left
  - The declaration **%right** specifies an associativity from the right
  - The precedence of the operators is established by the aparition order of the associativity declarations in the specification file, being the firstly declared operator the one with the lowest precedence
  - For example, these declarations

```
%left '+' '-'
%left '*' '/'
```

set that the four operators are associated from the left, and that '+' and '-' are of the same precedence, but lower than '*' and '/' (this operators have the same precedence)

Rules section I

- This is the **"most important"** section
- It contains the grammar rules that are written in a specific format and optionally with the actions associated to the rules
- Rules format:
  - `nonTerminalSymbol: simb1 simb2 ... simbM {action1}`
- If several rules have the same left hand side part, they may be grouped:
  - `nonTerminalSymbol: rightHandSidePart {action1}`
  - `| rightHandSidePart {action2}... ;`
- To clarify, we include comments for the lambda rules:
  - `nonTerminalSymbol: /* empty */ {action1}| ...;`

Rules section II

- The set of rule actions:
  - The actions are a set of C code instructions enclosed between brackets **'{}'**, and that are executed each time an instance of a rule is parsed
  - These actions usually are placed at the end of the rule, though other positions are admitted
  - Most of the times the actions work with semantic values of the symbols placed in the right hand side part. They are accesible with pseudo-variables of the type **$N**, where N represents the symbol position. The semantic value of the non-terminal symbol of the left hand side part of the rule is refered as **$$**
  - The type of the semantic value of a symbol is the one associated with a %token declaration (terminal) or %type (non terminal)
  - The default action is:
    - `$$ = $1`
  - Example:
    - `exp: ... | exp '+' exp { $$ = $1 + $3 }`

Rules section III

- It is assumed that the grammar main rule is the first non-terminal symbol of the rules section

- programa: TOK_MAIN TOK_LLAVEIZQUIERDA declaraciones funciones sentencias TOK_LLAVEDERECHA {fprintf(out, ";R1:\t<programa> ::= main { <declaraciones> <funciones> <sentencias> }\n");}

- See the alfa gramar doc!

User functions section I

- The content of this section is literally copied to the output file

- It is better to group all the support functions in a file or a set of files instead of including them in the user functions section when we are working on a more complex application

User functions section II

- Several support functions are placed in this section
  - Functions designed by the user to be used in the rules section
  - The **yyerror()** function
    - When the **yyparse()** function detects a syntactic error, it invoques the function **yyerror()**. This function has to be provided by the user, and can be incorporated in the file section of this specification (in Linux we are allowed to not declare it, thus it is provided by the Bison library)
    - This function prototype can be:
      - `void yyerror(char *s)`
    - Or
      - `int yyerror(char *s)`
    - The only parameter of this function is the synctactic error's message
    - The minimum version of this function prints to the output stream the message received as an argument