Course 10

Important notice

- 9.12.2021
  - 7.30 - Course Formal Languages and Compiler Design
  - 9.20 - Course Formal Languages and Compiler Design

- 16.12.2021
  - 7.30 – Course Parallel and Distributed Programming
  - 9.20 – Course Parallel and Distributed Programming

LEX & YACC

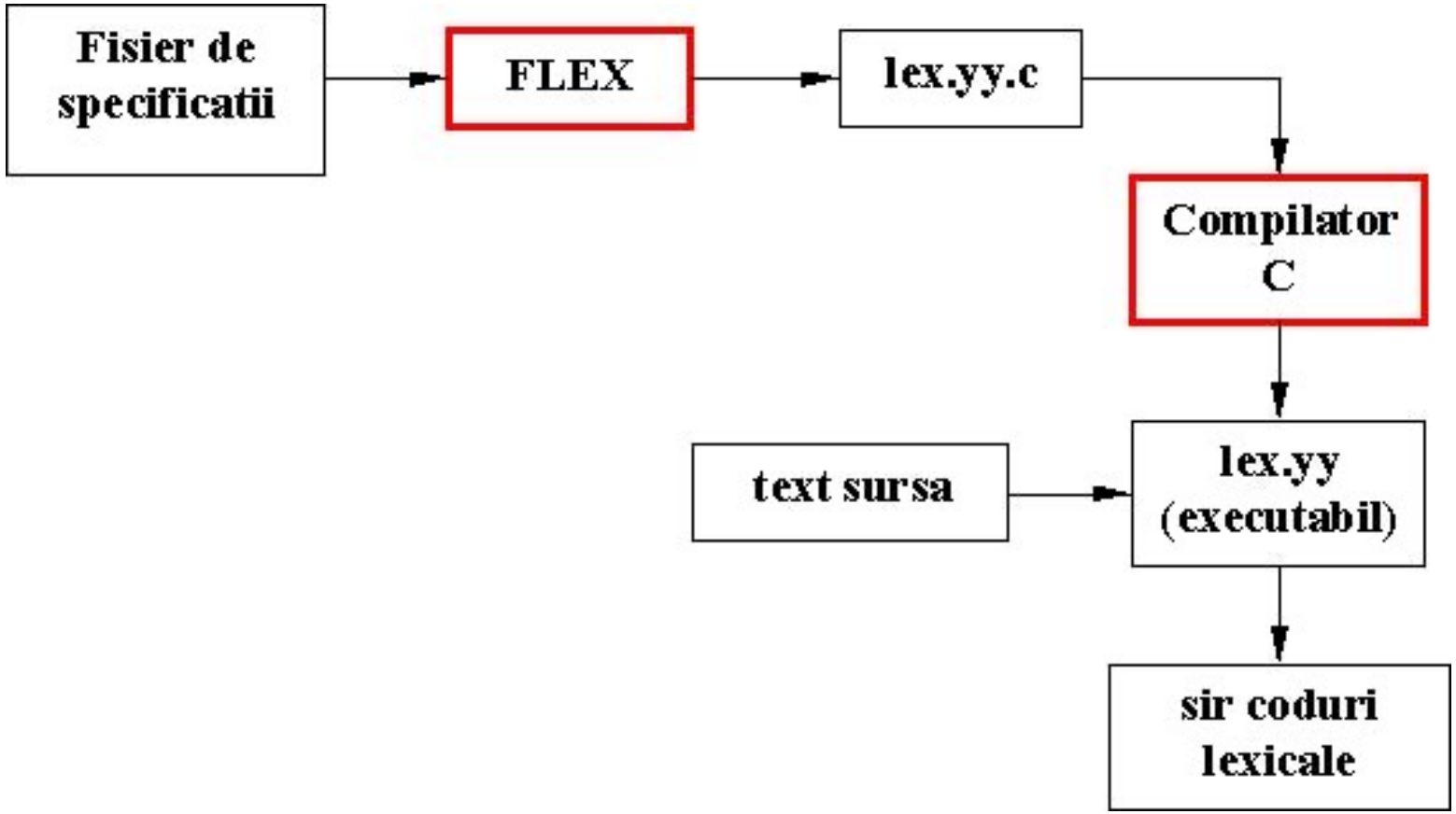
1. Have you heard about these tools?

2. Have you used any of them?

Scanning & Parsing Tools

- Scanning => lex
- Parsing => yacc

Lex – Unix utilitary (flex – Windows version)



INPUT FILE FORMAT

- The file containing the specification is a text file, that can have any name. Due to historic reasons we recommend the extension **.lxi**.
- Consists of 3 sections separated by a line containing %%:

```
definitions
%%
rules
%%
user code
```

Example 1:

```
%%

 username printf( "%s", getlogin() );
```

specifies a scanner that, when finding the string "`username`", will replace it with the user login name

Definition Section:

- C declarations

+

- declarations of simple *name definitions* (used to simplify the scanner specification), of the form

```
name definition
```

- where:
  - **name** is a word formed by one or more letters, digits, '_' or '-', with the remark that the first character MUST be letter or '_' and must be written on the FIRST POSITION OF THE LINE.
  - **definition** is a regular expression and is starting with the first nonblank character after name until the end of line.
  - declarations of *start conditions*.

Rules Section

- to associate semantic actions with regular expressions. It may also contain user defined C code, in the following way:

pattern action

where:

- **pattern** is a regular expression, whose first character MUST BE ON THE FIRST POSITION OF THE LINE;

- **action** is a sequence of one or more C statements that MUST START ON THE SAME LINE WITH THE PATTERN. If there are more than one statements they will be nested between {}. In particular, the action can be a void statement.

User Defined Code Section:

- Is optional (if is missing, then the separator %% following the rules section can also miss). If it exists, then its containing user defined C code is copied without any change at the end of the file lex.yy.c.
- Normally, in the user defined code section, one may have:
  - function *main()* containing call(s) to *yylex()*, if we want the scanner to work autonomously (for ex., to test it);
  - other called functions from *yylex()* (for ex. *yywrap()* or functions called during actions); in this case, the user code from definitions section must contain: either prototypes, either ***#include*** directives of the headers containing the prototypes

Launching the execution:

lex [*option*] [*name_specification _file*]

where *name_specification _file is an input file (implicitly,* stdin)

$ lex spec.lxi

$ gcc lex.yy.c -o your_lex

$ your_lex<input.txt

options: http://dinosaur.compilertools.net/flex/manpage.html

Example

yacc

Parsing (syntax analysis) modeled with cfg:

cfg G = (N, $\boldsymbol{\Sigma}$,P,S):

- N – nonterminal: syntactical constructions: declaration, statement, expression, a.s.o.
- $\boldsymbol{\Sigma}$ – terminals; elements of the language: identifiers, constants, reserved words, operators, separators
- P – syntactical rules – expressed in BNF – simple transformation
- S – syntactical construct corresponding to program

THEN

Program syntactical correct <=> w ∈ L(G)

yacc – Unix tool (Bison – Window version)

- **Yet Another Compiler Compiler**

- LALR
- C code

A yacc grammar file has four main sections

```
%{
C declarations
%}

yacc declarations

%%
Grammar rules
%%

Additional C code
```

contains declarations that define terminal and nonterminal symbols, specify precedence, and so on.

The grammar rules section

- contains one or more yacc grammar rules of the following general form:

```
result: components...     {C statements}

        ;

exp:      exp '+' exp
  ;

result:    rule1-components...
      | rule2-components...
      ...
      ;
result:                  /*empty */
      | rule2-components...
      ;
```

Example: expression interpreter

- input

```
%token DIGIT

%%
line : expr '\n'          { printf("%d\n", $1);}
     ;
expr : expr '+' expr      { $$ = $1 + $3;}
     | expr '*' expr      { $$ = $1 * $3;}
     | '(' expr ')'       { $$ = $2;}
     | DIGIT
     ;
%%
```

grammar

semantics

- Yacc has a stack of values - referenced ‘$i’ in semantic actions

- Input file (desk0)

```
%%
line : expr '\n'          { printf("%d\n", $1);}
     ;
expr : expr '+' expr      { $$ = $1 + $3;}
     | expr '*' expr      { $$ = $1 * $3;}
     | '(' expr ')'       { $$ = $2;}
     | DIGIT
     ;
```

```
> make desk0
bison -v desk0.y
desk0.y contains 4 shift/reduce conflicts.
gcc -o desk0 desk0.tab.c
>
```

Conflict resolution in yacc

- Conflict **shift-reduce** – prefer **shift**

- Conflict **reduce-reduce** – chose first production

```
%%
line : expr '\n'          { printf("%d\n", $1);}
     ;
expr : expr '+' expr      { $$ = $1 + $3;}
     | expr '*' expr      { $$ = $1 * $3;}
     | '(' expr ')'       { $$ = $2;}
     | DIGIT
     ;
%%
```

- Run yacc
- Run desk0

```
> desk0
2*3+4
14
```

Operator priority in yacc

- From low to great

```
%token DIGIT
%left '+'
%left '*'

%%
line : expr '\n'          { printf("%d\n", $1);}
     ;
expr : expr '+' expr      { $$ = $1 + $3;}
     | expr '*' expr      { $$ = $1 * $3;}
     | '(' expr ')'       { $$ = $2;}
     | DIGIT
     ;
%%
```

- Use

```
>lex spec.lxi
>yacc –d spec.y
>gcc lex.yy.c y.tab.c -o result –lfl
>result<InputProgram
```

- More on
  http://catalog.compilertools.net/lexparse.html

Example