Course 10

LEX & YACC

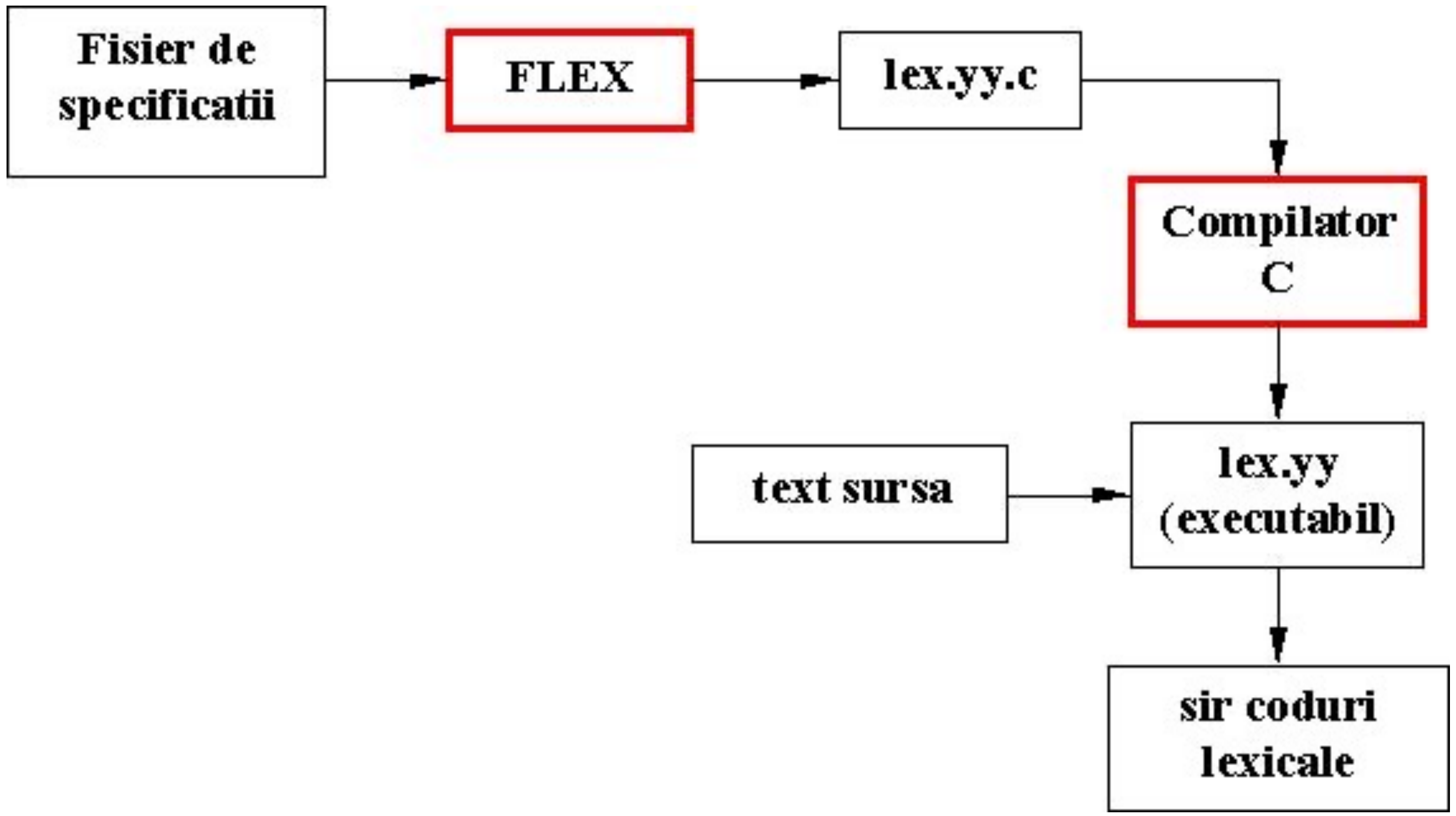
1. Have you heard about these tools?

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

Course 11

Push-Down Automata
(PDA)

Intuitive Model

Definition

- A push-down automaton (APD) is a 7-tuple $M = (Q,\boldsymbol{\Sigma},\boldsymbol{\Gamma},\boldsymbol{\delta},q_0,Z_0,F)$ where:
  - Q – finite set of states
  - $\boldsymbol{\Sigma}$ - alphabet (finite set of input symbols)
  - $\boldsymbol{\Gamma}$ – stack alphabet (finite set of stack symbols)
  - $\boldsymbol{\delta} : Q \times (\boldsymbol{\Sigma} \cup \{\boldsymbol{\varepsilon}\}) \times \boldsymbol{\Gamma} \to \mathcal{P}(Q\times \boldsymbol{\Gamma}^*)$ –transition function
  - $q_0 \in Q$ – initial state
  - $Z_0 \in \boldsymbol{\Gamma}$ – initial stack symbol
  - $F \subseteq Q$ – set of final states

Push-down automaton

Transition is determined by:

- Current state
- Current input symbol
- Head of stack

Reading head -> input band:

- Read symbol
- No action

Stack:

- Zero symbols => pop
- One symbol => push
- Several symbols => repeated push

Configurations and transition / moves

- Configuration:

$$(\mathbf{q}, \mathbf{x}, \boldsymbol{\alpha}) \in Q \times \boldsymbol{\Sigma}^* \times \boldsymbol{\Gamma}^*$$

where:

- PDA is in state **q**
- Input band contains **x**
- Head of stack is $\boldsymbol{\alpha}$

- Initial configuration (q_0, w, Z_0)

Configurations and moves(cont.)

- Moves between configurations:

p,q $\in$Q, a$\in\boldsymbol{\Sigma}$, Z $\in\boldsymbol{\Gamma}$, w $\in\boldsymbol{\Sigma^*}$,$\boldsymbol{\alpha}$,$\boldsymbol{\gamma}$ $\in\boldsymbol{\Gamma^*}$

(q,aw,Z$\boldsymbol{\alpha}$) $\vdash$ (p,w,$\boldsymbol{\gamma}$Z$\boldsymbol{\alpha}$) iff $\boldsymbol{\delta}$(q,a,Z) $\ni$ (p,$\boldsymbol{\gamma}$Z)

(q,aw,Z$\boldsymbol{\alpha}$) $\vdash$ (p,w, $\boldsymbol{\alpha}$) iff $\boldsymbol{\delta}$(q,a,Z) $\ni$ (p, $\boldsymbol{\varepsilon}$)

(q,aw,Z$\boldsymbol{\alpha}$) $\vdash$ (p,aw,$\boldsymbol{\gamma}$Z$\boldsymbol{\alpha}$) iff $\boldsymbol{\delta}$(q,$\boldsymbol{\varepsilon}$,Z) $\ni$ (p,$\boldsymbol{\gamma}$Z)

($\boldsymbol{\varepsilon}$-move)

- $\overset{k}{\vdash}$, $\overset{+}{\vdash}$, $\overset{*}{\vdash}$

Language accepted by PDA

- Empty stack principle:

$L_{\boldsymbol{\varepsilon}}(M) = \{w|w \in \boldsymbol{\Sigma^*}, (q_0,w,Z_0) \vdash^* (q,\boldsymbol{\varepsilon},\boldsymbol{\varepsilon}), q \in Q\}$

- Final state principle:

$L_f(M) = \{w|w \in \boldsymbol{\Sigma^*}, (q_0,w,Z_0) \vdash^* (q_f,\boldsymbol{\varepsilon},\boldsymbol{\gamma}), q_f \in F\}$

Representations

- Enumerate
- Table
- Graphic

Construct PDA

- $L = \{0^n 1^n \mid n \geq 1\}$
- States, stack, moves?

1. States:
   - Initial state:q_0 – beginning and process symbols '0'
   - When first symbol '1' is found – move to new state => q_1
   - Final: final state q_2
2. Stack:
   - Z_0 – initial symbol
   - X – to count symbols:
     - When reading a symbol '0' – push X in stack
     - When reading a symbol '1' – pop X from stack

Exemple 1 (enumerate)

$M = (\{q_0,q_1,q_2\}, \{0,1\}, \{Z_0,X\},\boldsymbol{\delta},q_0,Z_0,\{q_2\})$

$\delta(q_0,0,Z_0) = (q_0,XZ_0)$

$\delta(q_0,0,X) = (q_0,XX)$

$\delta(q_0,1,X) = (q_1,\varepsilon)$

$\delta(q_1,1,X) = (q_1,\varepsilon)$

$\cancel{\delta(q_1,\varepsilon,Z_0) = (q_2,Z_0)}$

$\boxed{\delta(q_1,\varepsilon,Z_0) = (q_1, \varepsilon)}$

Empty stack

$\vdash (q_1, \varepsilon, \varepsilon)$

$(q_0,0011,Z_0) \vdash (q_0,011,XZ_0) \vdash (q_0,11,XXZ_0) \vdash (q_1,1,XZ_0) \vdash (q_1, \varepsilon, Z_0) \vdash (q_2, \varepsilon, Z_0)$

Final state

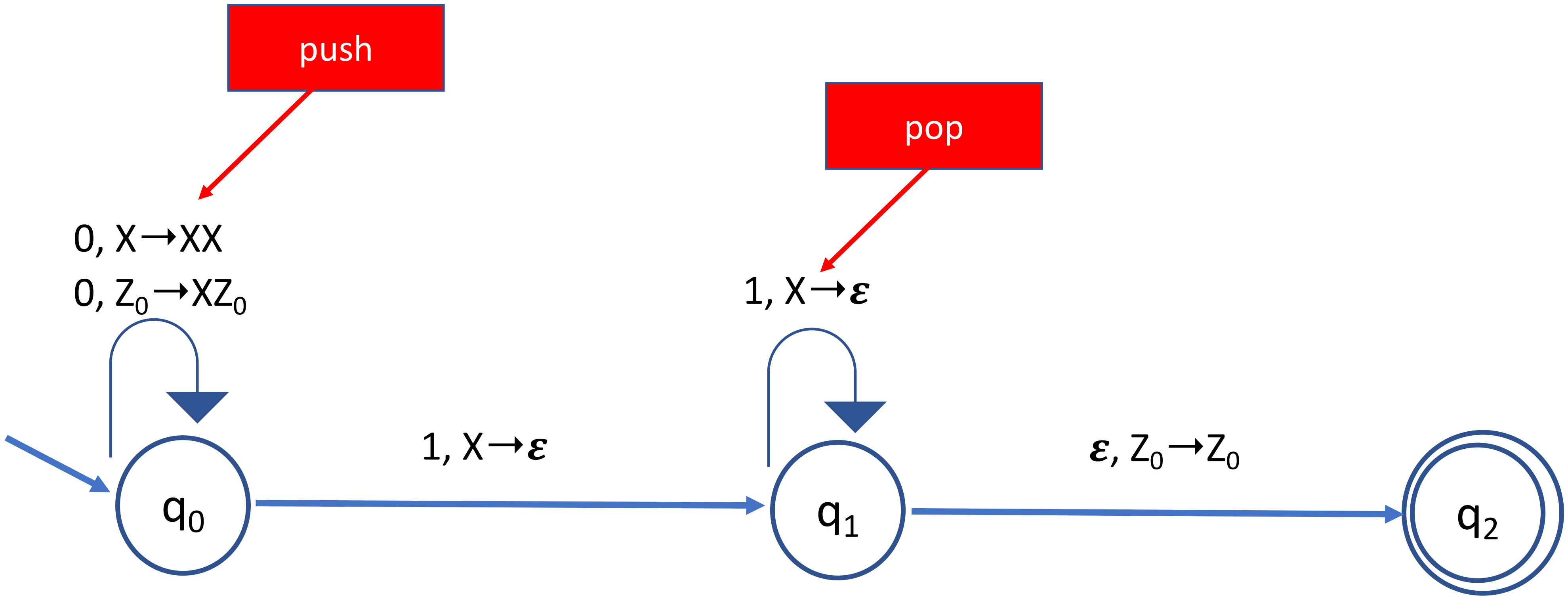
Exemple 1 (table)

		0	1	ε
q_0	Z_0	q_0,XZ_0		
	X	q_0,XX	q_1,ε	
q_1	Z_0			q_2,Z_0 (q_1, ε)
	X		q_1,ε	
q_2	Z_0			
	X			

(q0,0011,Z0) |- (q0,011,XZ0) |- (q0,11,XXZ0) |- (q1,1,XZ0)
|- (q1, ε,Z0) |- (q2, ε,Z0) q2 final seq. is acc based on final state

(q0,0011,Z0) |- (q0,011,XZ0) |- (q0,11,XXZ0) |- (q1,1,XZ0)
|- (q1, ε,Z0) |-(q1, ε, ε) seq is acc based on empty stack

Exemple 1 (graphic)



Properties

Theorem 1: For any PDA M, there exists a PDA M' such that

$$L_{\varepsilon}(M) = L_f(M')$$

Theorem 2: For any PDA M, there exists a context free grammar such that

$$L_{\varepsilon}(M) = L(G)$$

Theorem 3: For any context free grammar there exists a PDA M such that

$$L(G) = L_{\varepsilon}(M)$$

HW

- Parser:
  - Descendent recursive
  - LL(1)
  - LR(0), SLR, LR(1)

Corresponding PDA