Lex/Flex: Lab Activity

mariaiuliana.dascalu@gmail.com

FILS, 2012

What is it?

- a tool used to write lexical analyzers/lexers/scanners
- receives, as input, code in Scanner Description Language (SDL)
- returns, as output, code in C, which can be taken by a C compiler to obtain the exe code

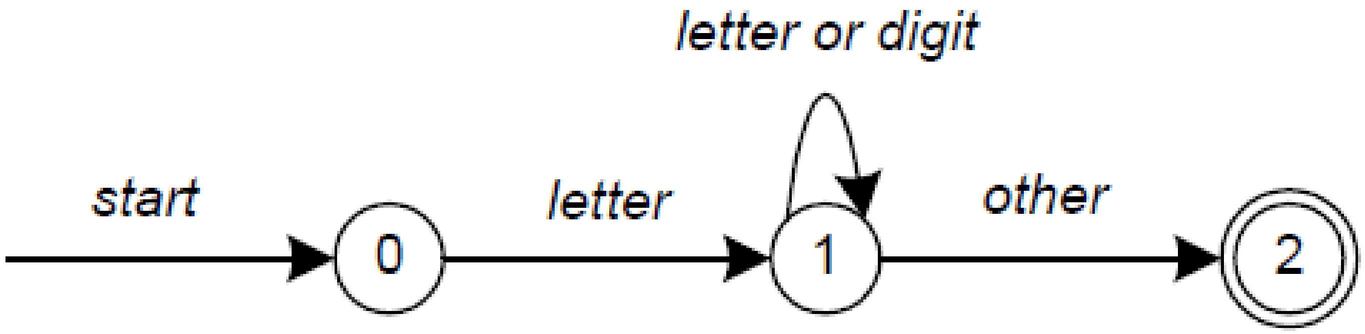
What does a scanner do?

- analyzes strings from an input source by applying **pattern matching**
- each pattern has an associated action:
  - returns tokens/lexicons/ language elements which are part of **regular language** class (e.g.: operators, constants, keywords, identifiers)
  - replace a pattern
  - count something,…

How it works?

- Input: regular expression
- Transforms it into a program which mimics the finite state automaton
- Output: tokens

Example: letter(letter|digit)*



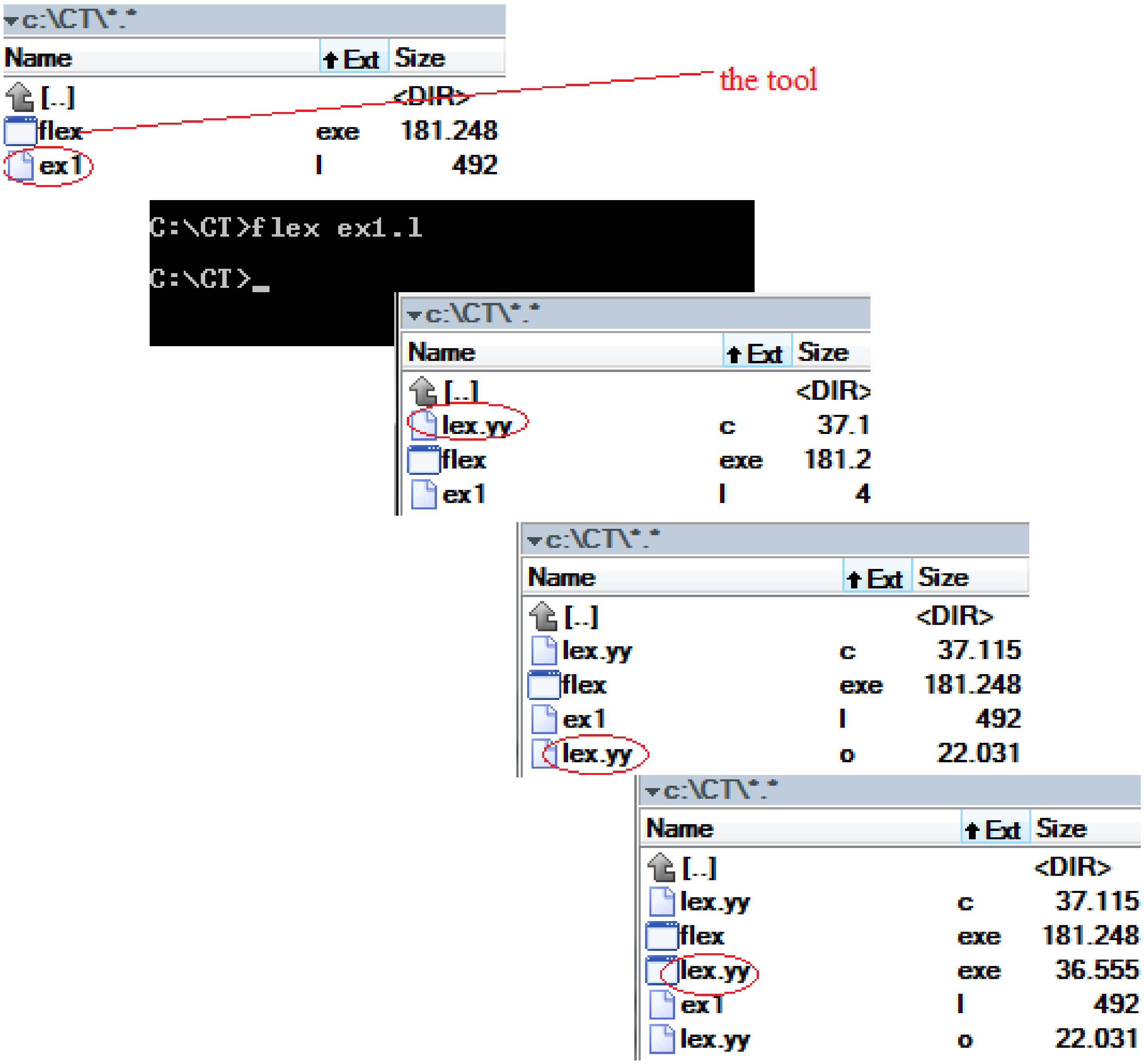
Download from…

- http://www.monmouth.com/~wstreett/lex-yacc/flex.exe
- http://flex.sourceforge.net/#downloads
- http://gnuwin32.sourceforge.net/packages/flex.htm

Flex is the free variant for Lex, available also for Win OS.

How to work with a Lex/Flex file?

- save the file with the extension **l**: ex1.l (written in SDL)
- go to the directory where your file is located and execute the following command: flex ex1.l
- lex.yy.c is created (written in C)
- compile the lex.yy.c with any C/C++ compiler and obtain the lex.yy.o file
- execute the lex.yy.c and obtain lex.yy.exe, your lexical analyzer



the tool

▾c:\CT\*.*

Name	↑Ext	Size
[..]		<DIR>
flex	exe	181.248
ex1	l	492

```
C:\CT>flex ex1.l

C:\CT>_
```

▾c:\CT\*.*

Name	↑Ext	Size
[..]		<DIR>
lex.yy	c	37.1
flex	exe	181.2
ex1	l	4

▾c:\CT\*.*

Name	↑Ext	Size
[..]		<DIR>
lex.yy	c	37.115
flex	exe	181.248
ex1	l	492
lex.yy	o	22.031

▾c:\CT\*.*

Name	↑Ext	Size
[..]		<DIR>
lex.yy	c	37.115
flex	exe	181.248
lex.yy	exe	36.555
ex1	l	492
lex.yy	o	22.031

How to write the input in SDL code?

- Input to Lex/Flex is divided into three sections, with %% dividing the sections:

... *definitions* ...

%%

... *rules* ...

%%

... *subroutines* ...

- The first “%%” is mandatory, as it shows that the rules section begins.

Definitions’ Section

- contains macros
  (substitutions), statements of start conditions, other preliminary C code, which is simply copied to the top of the generated C file; the preliminary code must be put between *%{* and *%}*
- Example: defining macros for letters and digits, defining a variable which will be used in the rules section

```
LETTER [a-zA-Z]
DIGIT [0-9]
%{
int counter=0;
%}
```

Rules' Section

- contains the patterns' descriptions and the actions which are made if the patterns are found
- the patterns are written with POSIX regular expressions
- the actions are pieces of C code which will be executed if the patterns are found
- the patterns are separated from the actions by **tabs**
- Example: counting the number of identifiers (an identifier has to start with a letter or an underscore and can contain only digits and letters)

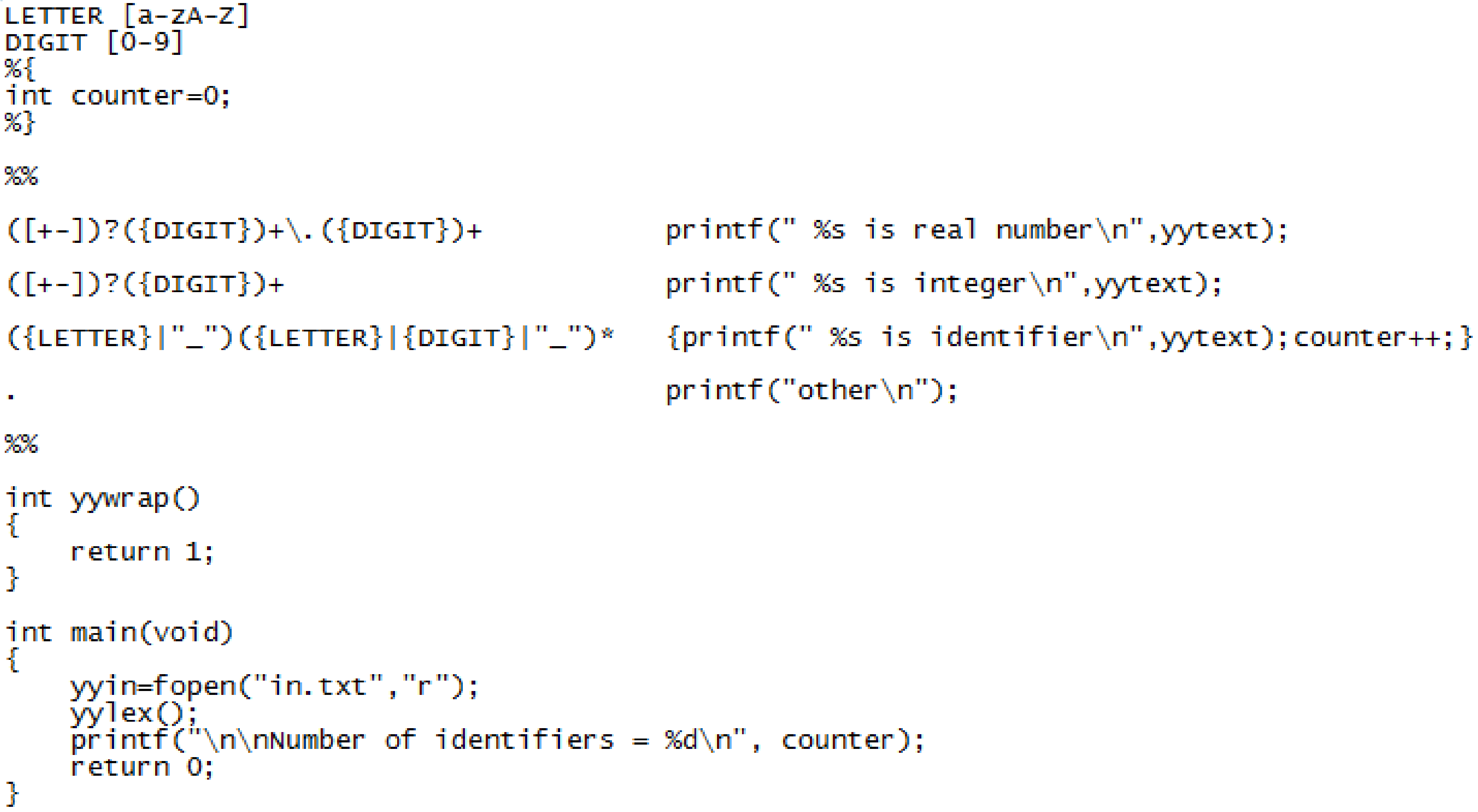
```
/* match identifier */
({LETTER}|"_")({LETTER}|{DIGIT}|"_")*		{printf(" %s is identifier\n",yytext);counter++;}
```

Subroutines’ Section

- C routines used by the actions defined in the rules section

```
int main(void)
{
   yyin=fopen("in.txt","r");
   yylex();
   printf("\n\nNumber of identifiers = %d\n", counter);
   return 0;
}
```

Example



```
LETTER [a-zA-Z]
DIGIT [0-9]
%{
int counter=0;
%}

%%

([+-])?({DIGIT})+\.({DIGIT})+           printf(" %s is real number\n",yytext);

([+-])?({DIGIT})+                       printf(" %s is integer\n",yytext);

({LETTER}|"_")({LETTER}|{DIGIT}|"_")*   {printf(" %s is identifier\n",yytext);counter++;}

.                                       printf("other\n");

%%

int yywrap()
{
    return 1;
}

int main(void)
{
    yyin=fopen("in.txt","r");
    yylex();
    printf("\n\nNumber of identifiers = %d\n", counter);
    return 0;
}
```

Exercises (1)

1. Find the number of lines, words and characters in a given file.

yyleng= the length of the string yytext

2. Count all instances of *she* and *he*, including the instances of *he* that are included in *she* from a text file. Use REJECT.

REJECT directs the scanner to proceed on to the "second best" rule which matched the input (or a prefix of the input).

3. Extract all html tags in a given file.

Exercises (2)

4. Extend the example from slide 12, to recognize the following tokens:

- Brackets : **( )**
- Operators: **+ - * /**
- Tests: **== != < <= > >=**
- C comments: **/* … */**
- C++ comments :**//…..**
- C assignments : **=**
- Reserved words: **if then else while for ...**

Alfabetul peste care se definesc expresiile regulate sunt caractere text

```
{a,b,...,z} ∪ {A,B,...,Z} ∪ {0,1,...,9} ∪ {_}
```

si caractere operator.

```
" [ ] ^ - ? . * + | ( ) , / { } % < >
```

- Operatori text: **"** si \ , sunt folositi la scrierea caracterelor operatori ca fiind caractere text.
  ```
  "#" sau # inseamna caracterul #, "\" inseamna caracterul \, iar \" inseamna caracterul ".
  ```
- Operatorul de compactare: **[ ]** este folosit la compactarea a doua patternuri.
  ```
  sit si sat se poate inlocui cu s[ai]t.
  ```
- Operatorul de negare: **^** este folosit la complementarierea unei multimi.
  ```
  [^ \t \n] semnifica orice caracter diferit de de spatiu, tab, sau <CR>.
  ```
- Operatorul de continuitate: **-** este folosit la precizarea unui domeniu continuu de valori.
  ```
  [0-9] se poate folosi in loc de [0123456789]; [a-zA-Z] in loc de [ab...zAB...Z];[a-z] este totuna cu [z-a].
  ```
- Operatorii de repetitie: *****, **+** si **{ }**.
  ```
  Patternul A* se potriveste cu orice numar de A-uri, chiar niciunul;
  Patternul A+ se potriveste cu orice numar de A-uri, dar cel putin unul;
  AAA si A{3} sunt echivalente; [a-z]{1,5} inseamna cuvintele cu litere mici cu lungimi de la 1 pana la 5.
  ```
- Operatorul universal: **.** se potriveste cu orice caracter diferit de de sfarsitul liniei.
  ```
  a.b se potriveste, de exemplu, cu aab, a0b, a\ b.
  ```
- Operatorul de alternare: **|** indica alternarea (sau exclusiv).
  ```
  ab|cd inseamna ab sau cd.
  ```
- Operatorul de grupare: **( )** indica concatenarea unor patternuri.
  ```
  ab|cd, (ab|cd) si (ab)|(cd) sunt echivalente; [a-c] si (a|b|c) sunt echivalente;
  (abc)+ se potriveste, de exemplu, cu abc, abcabc, abcabcabc etc.
  ```
- Operatorul de optionalitate: **?** indica faptul ca elementul precedent este optional.
  ```
  ab?c se potriveste cu ac sau abc; a(b|c)?d se potriveste cu ad, abd si acd.
  ```
- Operatorii de senzitivitate de context: **^** si **$**
  Sintaxa unui pattern format din contexte este <context stang> <context drept> urmat eventual de o actiune.