TDDD55- Compilers and Interpreters
Lesson 2

November 11 2011

Kristian Stavåker (kristian.stavaker@liu.se)

Department of Computer and Information Science
Linköping University

PURPOSE OF LESSONS

The purpose of the lessons is to practice some theory, introduce the laboratory assignments, and prepare for the final examination.

Read the laboratory instructions, the course book, and the lecture notes.

All the laboratory instructions and material available in the ***course directory, ~TDDD55/lab/***. Most of the PDF's also available from the course homepage.

LABORATORY ASSIGNMENTS

In the laboratory exercises you should get some practical experience in compiler construction.

There are 4 separate assignments to complete in 4x2 laboratory hours. You will also (most likely) have to work during non-scheduled time.

LESSON SCHEDULE

November 1,	13.15 -15:	Formal languages and automata theory
November 11,	8.15 - 10:	Formal languages and automata theory, Flex
November 22,	15.15 - 17:	Bison and intermediate code generation
December 6,	15.15 - 17:	Exam preparation

HANDING IN AND DEADLINE

- Demonstrate the working solutions to your lab assistant during scheduled time. Then send the modified files to the same assistant as well as answers to questions if any (put *TDDD55 <Name of the assignment>* in the topic field). One e-mail per group.

- Deadline for all the assignments is: **December 15, 2011**.

- Remember to register yourself in the webreg system, www.ida.liu.se/webreg (closed, e-mail your laboratory assistant)

LABORATORY ASSIGNMENTS

Lab 1 Top-Down Parsing
Lab 2 Scanner Specification
Lab 3 Parser Generators
Lab 4 Intermediate Code Generation

1. Grammars and Top-Down Parsing

- Some grammar rules are given
- Your task:
  - Rewrite the grammar (eliminate left recursion, etc.)
  - Implement your grammar in a C++ class named **Parser**. The **Parser** class should contain a method named **Parse** that returns the value of a single statement in the language.

2. Scanner Specification

- Finish a scanner specification given in a *scanner.l* flex file, by adding rules for C and C++ style comments, identifiers, integers, and floating point numbers.

3. Parser Generators

- Finish a parser specification given in a *parser.y* bison file, by adding rules for expressions, conditions and function definitions, ... You also need to augment the grammar with error productions.
- More on bison in lesson 3.

4. Intermediate Code Generation

- The purpose of this assignment to learn about how abstract syntax trees can be translated into intermediate code.
- You are to finish a generator for intermediate code by adding rules for some language statements.
- More in lesson 3.

LABORATORY SKELETON

- ~TDDD55
  - /lab
    - /doc
      - Documentation for the assignments.
    - /lab1
      - Contains all the necessary files to complete the first assignment
    - /lab2
      - Contains all the necessary files to complete the second assignment
    - /lab3-4
      - Contains all the necessary files to complete assignment three and four

INSTALLATION

- Take the following steps in order to install the lab skeleton on your system:
  - Copy the source files from the course directory onto your local account:

```
mkdir TDDD55
cp -r ~TDDD55/lab TDDD55
```

  - You might also have to load some modules (more information in the laboratory instructions).

TODAY

- Introduction to the flex scanner generator tool.
- Introduction to laboratory assignment 2.
- Exercises in formal languages and automata theory.

FLEX

SCANNERS

Scanners are programs that recognize lexical patterns in text

- Its **input** is text written in some language.
- Its **output** is a sequence of tokens from that text. The tokens are chosen according with the language.
- Building a scanner manually is tedious.
- Mapping the regular expressions to finite state machine/automata is straightforward, so why not automate the process?
- Then we just have to type in regular expressions and actions and get the code for a scanner back.

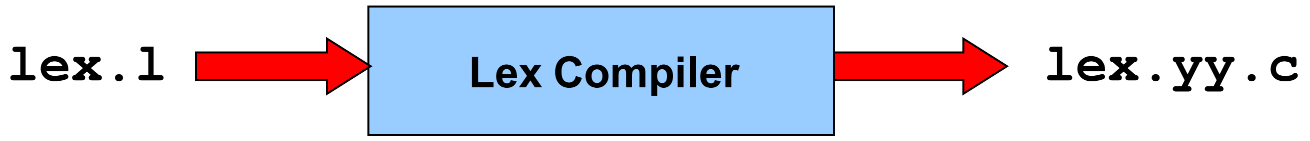
SCANNER GENERATORS

- Automate is exactly what **flex** does!
- **flex** is a fast lexical analyzer generator, a tool for generating programs that perform pattern matching on text
- **flex** is a free implementation of the well-known **lex** program

HOW IT WORKS

flex generates at output a **C** source file lex.yy.c which defines a routine yylex()

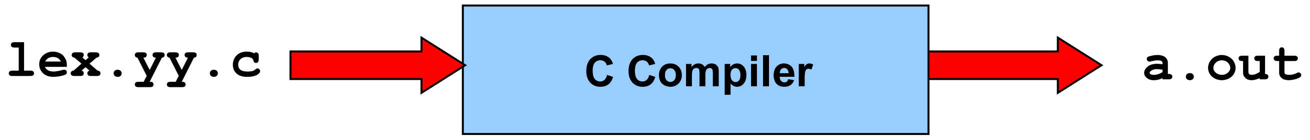
```
>> flex lex.l
```



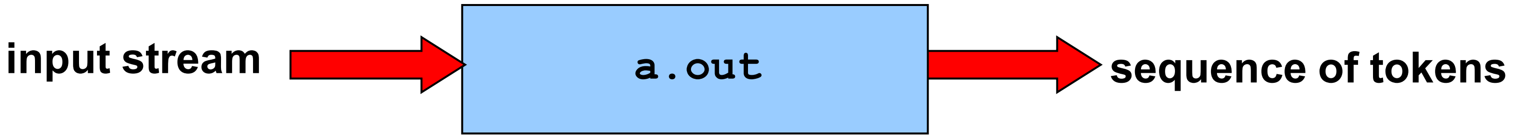
HOW IT WORKS

lex.yy.c is compiled and linked with the -lfl library to produce an executable, which is the scanner

```
>> g++ lex.yy.c -lfl
```



```
>> a.out < input.txt
```



FLEX SPECIFICATIONS

Lex programs are divided into three components

```
/* Definitions – name definitions
 *             – variables defined
 *             – include files specified
 *             – etc
 */

%%

/* Translation rules – regular expressions together with actions in C/C++ */

%%

/* User code – support routines for the above C/C++ code */
```

NAME DEFINITIONS

- *Definitions* are intended to simplify the scanner specification and have the form:

```
name     definition
```

- Subsequently the definition can be referred to by **{name}**, witch then will expand to the **definition**.
- Example:

```
DIGIT   [0-9]
{DIGIT}+"."{DIGIT}*
```

is identical/will be expanded to:

```
([0-9])+"."([0-9])*
```

PATTERN ACTIONS

- The *translation rules* section of the **lex/flex** input file, contains a series of rules of the form:

```
pattern  action
```

- Example:

```
[0-9]*    { printf ("%s is a number", yytext);     }
```

FLEX MATCHING

Match as much as possible.

If more than one rule can be applied, then the first appearing in the flex specification file is preferred.

SIMPLE PATTERNS

Match only one specific character

x	The character '**x**'
.	Any character except newline

CHARACTER CLASS PATTERNS

Match any character within the class

[xyz] The pattern matches either '**x**', '**y**', or '**z**'

[abj-o] This pattern spans over a range of characters and matches '**a**', '**b**', or any letter ranging from '**j**' to '**o**'

NEGATED PATTERNS

Match any character not in the class

[^z]	This pattern matches any character EXCEPT **z**
[^A-Z]	This pattern matches any character EXCEPT an uppercase letter
[^A-Z\n]	This pattern matches any character EXCEPT an uppercase letter or a newline

SOME USEFULL PATTERNS

r*	Zero or more '**r**', '**r**' is any regular expr.
\\0	**NULL** character (ASCII code 0)
\123	Character with octal value **123**
\x2a	Character with hexadecimal value **2a**
p|s	Either '**p**' or '**s**'
p/s	'**p**' but only if it is followed by an '**s**', which is not part of the matched text
^p	'**p**' at the beginning of a line
p$	'**p**' at the end of a line, equivalent to '**p/\n**'

FLEX USER CODE

Finally, the *user code* section is simply copied to lex.yy.c verbatim. It is used for companion routines which call, or are called by the scanner.

If the lex program is to be used on its own, this section will contain a main program. If you leave this section empty you will get the default main.

The presence of this user code is optional.

FLEX PROGRAM VARIABLES AND FUNCTIONS

yytext	Whenever the scanner matches a token, the text of the token is stored in the null terminated string yytext
yyleng	The length of the string yytext
yylex()	The scanner created by the Lex has the entry point yylex(), which can be called to start or resume scanning. If **lex** action returns a value to a program, the next call to yylex() will continue from the point of that return

FLEX PROGRAM VARIABLES AND FUNCTIONS

yymore()	Do another match and append its result to the current match
yyless(int n)	Push all but the first n characters back to the input stream (to be matched next time). yytext will contain only the first n of the matched characters.

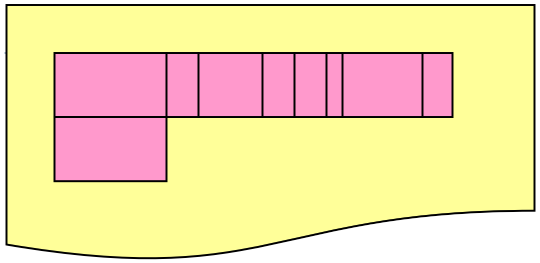
yymore() EXAMPLE

If the input string is “hypertext”, the output will be “Token is hypertext”.

```
%%

hyper yymore();

text printf(“Token is %s\n”, yytext);
```

FLEX EXAMPLES

EXAMPLE: RECOGNITION OF VERBS



```
%{
/* includes and defines should be stated in this section */
%}

%%

            /* ignore white space */

do|does|did|done|    { printf ("%s: is a verb\n", yytext); }
            { printf ("%s: is not a verb\n",yytext); }
.|\n        { ECHO; /* normal default anyway */ }

%%

main()          { yylex(); }
```

EXAMPLE: CHARACTER COUNTING

A scanner that counts the number of characters and lines in its input

```
int num_lines = 0, num_chars = 0; /* Variables */

%%

\n    { ++num_lines; ++num_chars; } /* Take care of newline */
.     { ++num_chars; }      /* Take care of everything else */

%%
main() { yylex();
  printf("lines: %d, chars: %d\n", num_lines, num_chars );
}
```

The printed output is the result.

EXAMPLE: CHARACTER COUNTING (2)

'\n' A newline increments the line count and the character count

'.' Any character other than the newline only increment the character count

EXAMPLE: SMALL LANGUAGE SCANNER

```
%{
  #include <math.h>
%}
DIGIT    [0-9]
ID       [a-z][a-z0-9]*

%%

{DIGIT}+  { printf("An integer: %s (%d)\n", yytext,  atoi( yytext ));          }

{DIGIT}+"."{DIGIT}*
        { printf("A float: %s (%g)\n", yytext, atof( yytext )); }

if|then|begin|end|procedure|function
  { printf("A keyword: %s\n", yytext); }

{ID}         { printf("An identifier: %s\n", yytext); }
```

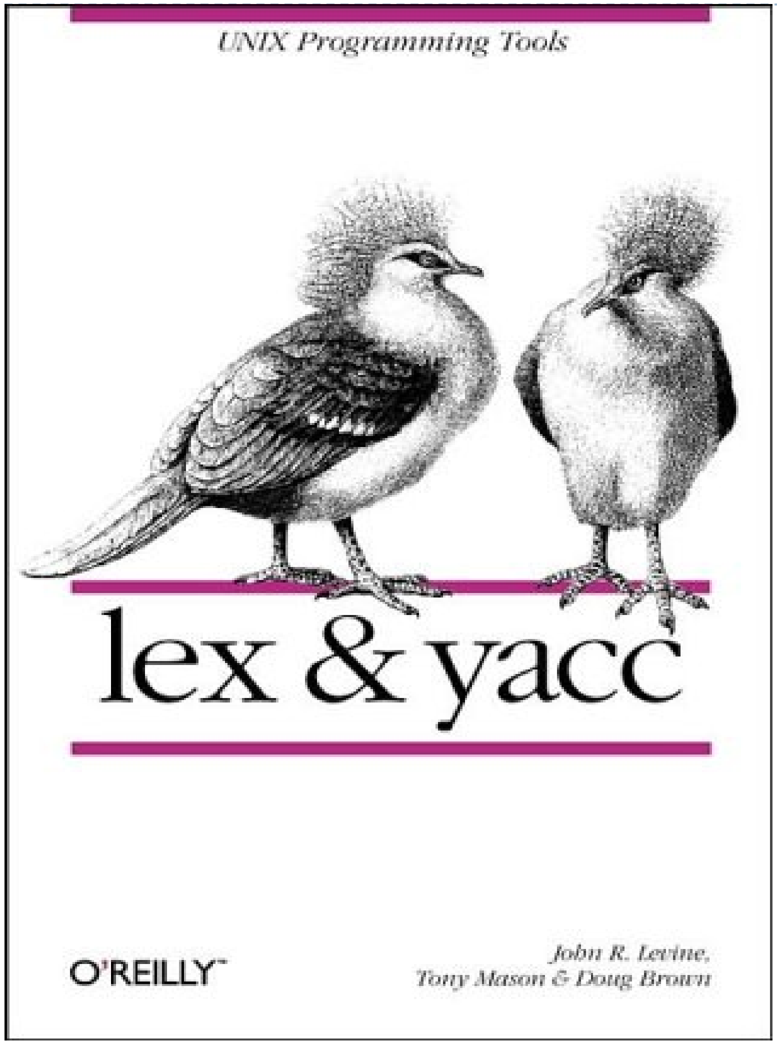
EXAMPLE: SMALL LANGUAGE SCANNER (2)

```
"+"|"-"|"*"|"/"                  { printf("An operator: %s\n", yytext); }


"{"[\^{$\;$}}\n]*"}"             /* eat up one-line comments */

[\t \n]+                         /* eat up whitespace */

.                                { printf("Unknown character: %s\n", yytext );}

%%

main(argc, argv) {
   ++argv, --argc;  /* skip over program name */
   if ( argc > 0 )  yyin = fopen( argv[0], "r" );
   else  yyin = stdin;
   yylex();
}
```

EXAMPLE: HTML TAGS

```
/*Declarations */
%{
#include <stdio.h>
%}

/*Exclusive, only rules specific to <html_tag> will match */
%x html_tag
%%

[^<]*                /* matches any char (zero or more times) except "<" */

"<"                  BEGIN(html_tag);  /*If we find "<" go into context <html_tag> */

<html_tag>[^>]*      printf("%s\n", yytext);

<html_tag>">"        BEGIN(INITIAL); /* Enter intial/normal context */
%%
```

MORE ON LEX



If you'll use flex in the future…

Lex & Yacc, 2nd ed
By, John R Levine, Tony Mason & Doug Brown
O'Reilly & Associates
ISBN: 1565920007

LABORATORY ASSIGNMENT 1 (from last time)

REWRITING THE GRAMMAR

- Use one non-terminal for each precedence level.

  E ::= E + E | E – E | T

  T ::= T * T | T / T | …

- (Left) Associativity:

  E ::= E + E | E – E | T => E ::= E + T | E – T | T

- See for instance:

http://www.lix.polytechnique.fr/~catuscia/teaching/cg428/02Spring/lecture_notes/L03.html

REWRITING THE GRAMMAR (2)

- Transform the grammar to right recursive form:

  A ::= A α | β (where β may not be preceded by A)

  is rewritten to

  A ::= β A'
  A' ::= α A' | ε

- See *Lecture 5 Syntax Analysis, Parsing*

IMPLEMENTATION

- You have been give a main function in *main.cc*.

```
int main(void) {
        Parser parser; double val;

        while (1) {

                try    {
                   cout << "Expression: " << flush;
                   val = parser.Parse();
                   cout << "Result:     " << val << '\n' << flush;
                }
                catch (ScannerError& e) {
                    cerr << e << '\n' << flush;
                    parser.Recover();
                }
                catch (ParserError) { parser.Recover(); }

                catch (ParserEndOfFile)  { cerr << "End of file\n" << flush; exit(0); }
                }
        }
}
```

IMPLEMENTATION (2)

- You have also been given files *lab1.cc* and *lab1.hh* for implementing your *Parser* class.
- In the function *Parse*, start the parsing.

```
double Parser::Parse(void) {
        Trace x(“Parse”);
        double val;
        val= 0;
        crt_token = the_scanner.Scan();
        switch (crt.token.type)
        {
                case kIdentifier:
                case kNumber:
                case kLeftParen:
                case kMinus:
                        val = pExpression();
                        if (crt_token.type != kEndOfLine) throw ParserError();
                        return val;
                default: throw ParserError();
        }
        return val;
}
```

IMPLEMENTATION (3)

- Add one function for each non-terminal in the grammar to your *Parser* class.
- See Lecture 5 *Syntax Analysis, Parsing*

```
double Parser::pExpression(void) {
        switch (crt_token.type) {
        ... ...
        }
}
```

IMPLEMENTATION (4)

- You don’t need to change anything in *lex.cc* and *lex.hh*.
- Also implement some simple error recovery in your *Parser* class.

LABORATORY ASSIGNMENT 2

LABORATORY ASSIGNMENT 2

- Finish a scanner specification given in a *scanner.l* flex file.
- Add regular expressions for floating point numbers, integer numbers, C comments (both /* */ comments and // one line comments), identifiers, empty space, newline.
- Rules for the language keywords are already given in the *scanner.l* file. Add your rules below them.

COMMENTS

- Skip comments, both single-line C++ comments and multi line C style comments.
- If the scanner sees /* within a C comment, print a warning message.
- If end of line is encountered within a C style comment, print an error message and then terminate.

COMMENTS EXAMPLE

Rules for comments.

```
“//”.*\n                              /* Do nothing */

“/*”                                   BEGIN(c_comment)

<c_comment> {

“*/”                    … …
“/*”                    fprintf(stderr, “Warning: Nested comments\n”);
… …

}
```

FLOATING POINT NUMBERS

- Floating-point numbers consist of an integer part followed by a period, decimal part and an exponent part.
- The integer and decimal parts are sequences of digits. The exponent part consists of the character *e* or *E* followed by an optional sign + or - and a sequence of digits.
- Either the integer or the decimal part (or both) must be given.
- The exponent is optional.
- If the integer part and exponent are both given, the decimal point and decimal part are optional.

FLOATING POINT NUMBERS EXAMPLE

- 1.1
- .1
- 1.
- 1E2
- 2E-3
- 1E-4

- Use **?** as *optional* pattern. Example: [+-]?

INTEGERS

- Integers are simply sequences of digits that are not part of identifiers or floating-point numbers.

IDENTIFIERS

- Identifiers must start with a letter, followed by any number of digits, letters or underscore characters.