TDD55: Compilers and Interpreters

Lesson 2

Filip Strömbäck (filip.stromback@liu.se)
Department of Computer and Information Science
Linköping University

Schedule

1. Formal languages and automata theory
2. Formal languages and automata theory, Flex
3. Bison and intermediate code generation
4. Exam preparation
5. Exam preparation

Laboratory assignments

- Goal: Get some practical experience in compiler construction
- 4 assignments to complete in 4x2 sessions → non-scheduled time required
    1. Attribute Grammars and Top-Down parsing
    2. Scanner Specification
    3. Parser Generators
    4. Intermediate Code Generation

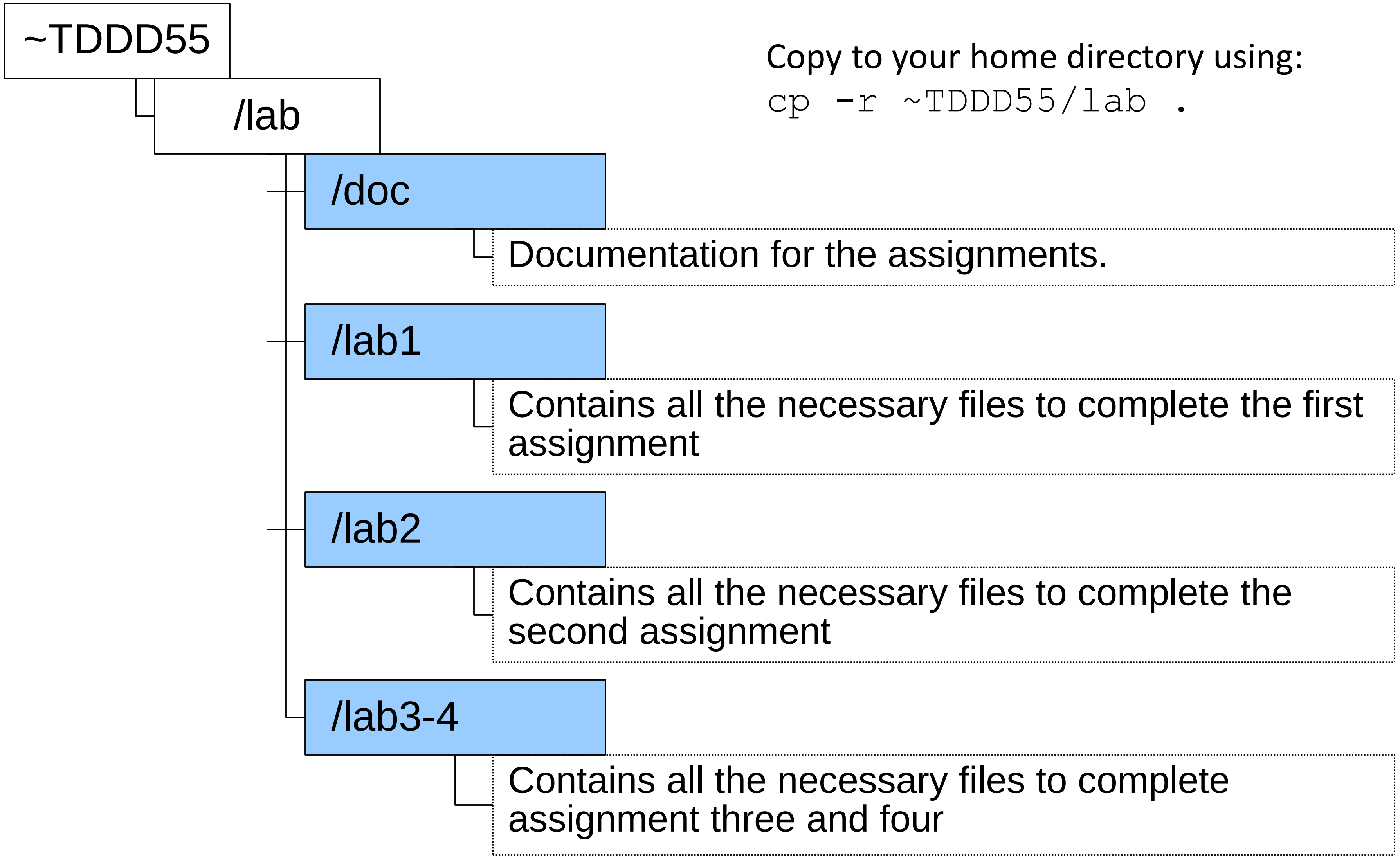
Handing in and deadline

- Demonstrate during scheduled sessions
- Then, hand in code and answers to any questions in the assignment via e-mail. One e-mail per group, subject: TDDD55: Lab n. From your LiU-email.
- Deadline: 21st December

Skeleton

Copy to your home directory using:

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



Documentation for the assignments.

Contains all the necessary files to complete the first assignment

Contains all the necessary files to complete the second assignment

Contains all the necessary files to complete assignment three and four

Lab 1

- Some grammar rules given:
- Rewrite the grammar to LL(1)
- Add attribute rules to the grammar
- Implement the LL(1) grammar and the attributes in a C++-class named Parser. Parser shall contain a method Parse() which returns the value of a single statement in the language.

Lab 2

- Finish a scanner specification in Flex (scanner.l) by adding rules for comments, identifiers, integers and reals.

Lab 3

- Finish a parser specification in Bison (parser.y) by adding rules for expressions, conditions, function definitions, etc.
- You also need to add error productions.
- More details in lesson 3.

Lab 4

- Generate intermediate code from a parse tree.
- Finish a generator for intermediate code by adding rules for some language statements.
- More details in lesson 3.

Flex

Scanners

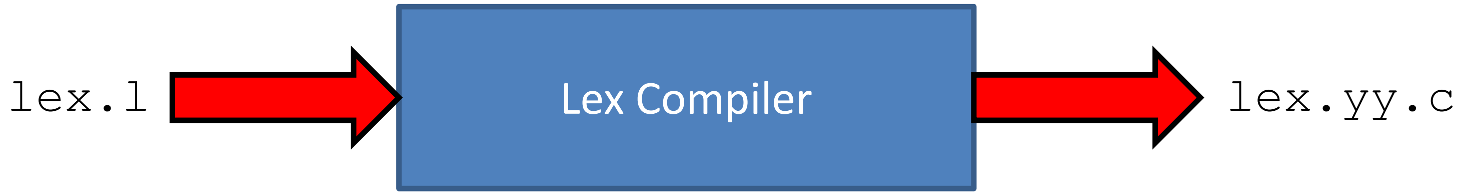
- Scanners are programs that recognize lexical patterns in text
- Input: text written in some language
- Output: sequence of tokens
- Mapping regular expressions to finite state machine is straightforward.
- Automate the process with flex

Scanner generators

- 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

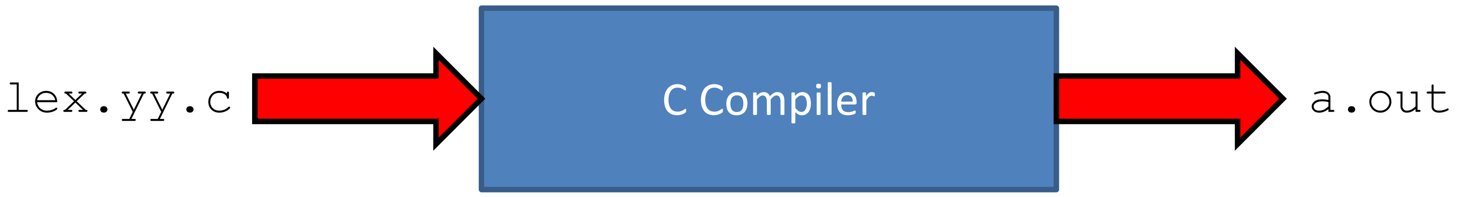
Flex generates a C source file, lex.yy.c, which defines yylex()



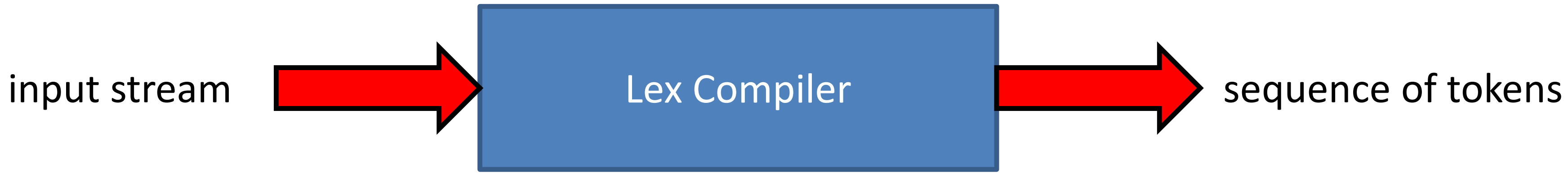
```
>> flex lex.l
```

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
```

Lex specifications

Lex programs are divided into three components

```
/* Definitions -     name definitions
 *                   variable definitions
 *                   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

- Intended to simplify the scanner specification and have the form:

```
name        definition
```

- It can then be referred to using {name}
- Example:

```
DIGIT        [0-9]
{DIGIT}+"."{DIGIT}*
which is equivalent to
([0-9])+"."([0-9])*
```

Lex specifications

Lex programs are divided into three components

```
/* Definitions -     name definitions
 *                   variable definitions
 *                   include files specified
 *                   etc.
 */

%%

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

%%

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

Pattern actions

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

  pattern action

- Example:

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

Flex matching

- Match as much as possible.
- If more than one rule can be applied, the **first appearing** in the specification file is preferred.

Patterns

- Match a specific character

"x"	match the character x
.	match any character except newline

Patterns

- Match any character inside the class

[xyz]	Either ‘x’, ‘y’, or ‘z’
[abj-o]	Either ‘a’, ‘b’ or any letter in the range ‘j’ to ‘o’

Patterns

- Match any character not in a class

[^z]	Any character except z
[^A-Z]	Any character except an uppercase letter
[^A-Z\n]	Any character except an uppercase letter and a newline.

Some useful 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 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**’

Lex specifications

Lex programs are divided into three components

```
/* Definitions -     name definitions
 *                   variable definitions
 *                   include files specified
 *                   etc.
 */

%%

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

%%

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

Flex user code

- The presence of user code is optional
- The user code section is copied veriatim to lex.yy.c. It is used for companion routines wich call or are called by the scanner.
- If the lex program is to be used on its own, this section will contain a main function. If you leave this section empty, you get the default main.

Flex variables and functions

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

Flex variables and functions

yymore()	Do another match and append its result to the current yytext (instead of replacing it).
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

```
%%

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

What is printed if the input is “hypertext”?

Example: Recognize verbs

```
%{
#include <stdio.h>
%}

%%

[ \t]+   /* ignore white space */

”do”|”does”|”did”|”done”|”has” {
            printf(“%s: is a verb\n”, yytext); }
[a-zA-Z]+ { printf(“%s: is not a verb\n”, yytext); }
.|\n      { ECHO; }

%%

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

Example: Character counting

```
%{
#include <stdio.h>
%}

int num_lines = 0, num_chars = 0;

%%

\n         { ++num_lines; ++num_chars; }
.          { ++num_chars; }

%%

main() {
    yylex();
    printf(“lines: %d, characters: %d\n”,
        num_lines, num_chars);
}
```

Example: HTML Tags

```
%{
#include <stdio.h>
%}

/* Exclusive start condition, ie. only rules specific
 * to <html_tag> will match */
%x html_tag
%%

[^<]*     /* matches any char (zero or more times) except "<" */
"<"       BEGIN(html_tag); /* activate start condition <html_tag> */

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

%%

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

Lab 2

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

Lab 2

- Skip characters in comments, both single-line and multi-line comments.
- If the scanner sees /* within a C comment, it must print a warning message.

Comments example

```
%{
#include <stdio.h>
%}

%x c_comment
%%
...

“//”.*\n  /* Do nothing */
“/*”      BEGIN(c_comment);

<c_comment> {
“*/”      ...
“/*”      fprintf(stderr, “Warning: Nested comments!\n”);
...
}
%%

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

Integers and identifiers

- Integers are just a sequence of numbers
- Identifiers must start with a letter, followed by any number of digits, letters or underscore characters.

Floating point numbers

- Floating-point numbers consist of an integer part followed by a decimal point, decimal part and an exponent part.
  - eg. 56.11E-2
- The integer and decimal parts are sequences of digits. The exponent part consists of the character 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.

Examples

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