TDDD55- Compilers and Interpreters
Lesson 1

Zeinab Ganjei (zeinab.ganjei@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 are 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.

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 22 2015**.

- Remember to register yourself in the webreg system, www.ida.liu.se/webreg

Laboratory Assignments

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

1. Attribute Grammars and Top-Down Parsing

- Some grammar rules are given
- Your task:
  - Rewrite the grammar (eliminate left recursion, etc.)
  - Add attributes and attribute rules to the grammar
  - Implement your attribute 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 comments, identifiers, integers, and reals.
- More on flex in lesson 2.

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 parse 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

Hints for Laboratory Assignment 1

Grammar for simple mathematical expressions

S -> E <end of line> S	Single expression
| <end of file>	No more input
E -> E + E	Addition
| E - E	Subtraction
| E * E	Multiplication
| E / E	Division
| E ^ E	Exponentiation
| - E	Unary minus
| (E)	Grouping
| id (E)	Function call
| id	Symbolic constant
| num	Numeric value

Not Suitable for a Top-Down Technique

- Left recursion
- Ambiguous
- No operator precedence
  - 9 + 1 – 5 ^2*1
- No operator associativity
  - 9 - 1 - 5 -2-1-4^2^3

Rewriting the Grammar

- Use one non-terminal for each precedence level.

  E ::= E + E | E – E | T

  T ::= T * T | T / T

- (Left) Associativity: using (left-)recursive production

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

- More details about ambiguity and associativity in grammars:

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

Rewriting the Grammar (2)

- The grammar obtained so far has left recursion
  - Not suitable for a predictive top-down parser
- 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*
- *More details: http://en.wikipedia.org/wiki/Left_recursion*

Attribute Grammars

- Define attributes for the productions of a formal grammar
- Example:

```
S ::= E          { display( E.val ); }
E ::= E1 + T     { E.val = E1.val + T.val; }
    | T          { E.val = T.val;  }
T ::= T1 * F     { T.val = T1.val * F.val; }
    | F          { T.val = F.val; }
F ::= ( E )      { F.val = E.val; }
    | num        { F.val = num.val; }
```

- See course book and *Lecture 8 Semantic Analysis, Intermediate Representation, and Attribute Grammars*.
- See also: http://en.wikipedia.org/wiki/Attribute_grammar

Implementation: 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); }
                }
        }
}
```

You have been given a main function in *main.cc*.

Implementation: lex.cc, lex.hh

- These files implement the lexer
- You don’t need to change anything in *lex.cc* and *lex.hh*.

Tips before implementation

- Write down the grammar, make all changes needed to enforce precedence, etc
- Add attribute to your grammar for computing the numerical value of the mathematical expression
- Prepare your parsing table

Sample: A context-free grammar and its corresponding parsing table for a LL(1) parser (inherently top-down):

1. $S \rightarrow F$
2. $S \rightarrow (S + F)$
3. $F \rightarrow a$

	(	**)**	**a**	**+**	**eol**
S	2		1		
F			3		

Note: Blank entries in the table, indicate error situations

- Then start implementation!

Implementation : lab1.cc, lab1.hh

```
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;
}
```

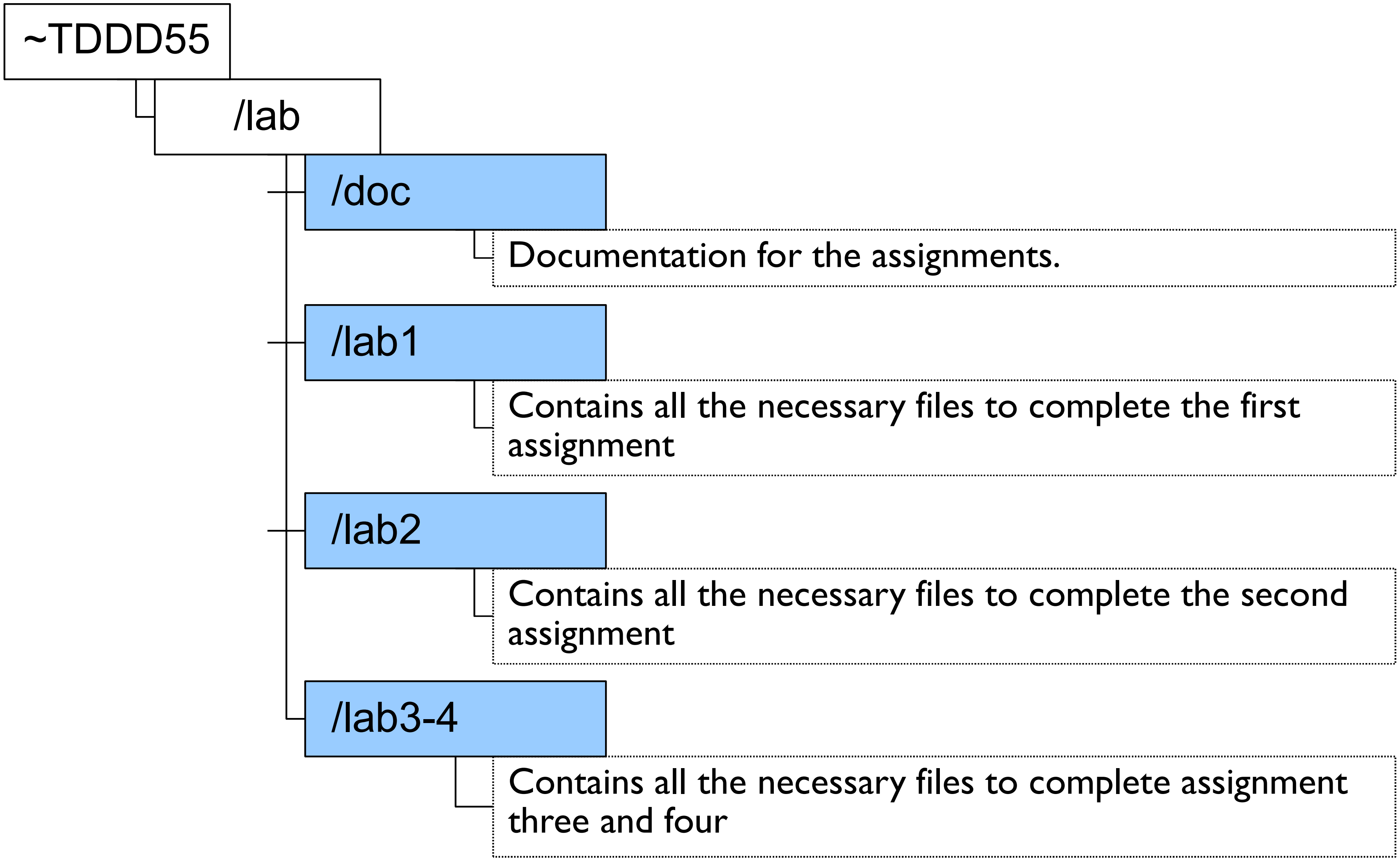
- *lab1.cc* and *lab1.hh* for implementing *Parser* class.
- In the function *Parse(),* start the parsing.

Implementation…

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

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

LABORATORY SKELETON



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

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).

Questions?