TDDD55- Compilers and Interpreters
Lesson 1

Zeinab Ganjei
(zeinab.ganjei@liu.se)

Department of Computer and Information Science
Linköping University

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

Not Suitable for a Top-Down Technique

- No operator precedence
  - e.g E -> E + E
    | E * E
- No operator associativity
  - e.g. E ^ E
- Left recursion
  - e.g. E -> E + E
- Ambiguity

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

See for instance:
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 ::= E + T  { E.val = E.val + T.val; }
    | T      { E.val = T.val;  }
T ::= T * F  { T.val = T.val * F.val; }
    | F      { T.val = F.val; }
F ::= ( E )  { F.val = E.val; }
    | num    { F.val = num.val; }
```

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); }
                }
        }
}
```

Implementation: lex.cc and lex.hh

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

Implementation : lab1.cc, lab1.hh

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

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 for details.

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

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).