# TDDD55 Programming Exercise 1: Attribute Grammars and Top-Down Parsing (HT 2013)

## 1 Introduction

Although not as flexible as bottom-up parsers, top-down parsers can easily be implemented by hand, and as such they may be more convenient than a bottom-up parsers. In this exercise you will specify a language of mathematical expressions using an attribute grammar, and then write a top-down parser to calculate the value of expressions in the language.

The language consists of numbers, symbolic constants, single-argument functions, one unary and five binary operators. A grammar for the language is given below, but this grammar is not suitable for implementation using a top-down technique since it is ambiguous and contains left recursion.

```
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
```

## 2 Requirements

Rewrite the grammar in the previous section so that the precedence and associativity of all operators becomes obvious. Your grammar may contain left recursion. The operator precedence is unary negation before exponentiation before multiplication and division, before addition and subtraction. Addition, subtraction, multiplication and division are left associative. Exponentiation is right-associative.

Eliminate left recursion from your grammar and revise it so it is suitable for implementation in a predictive top-down parser.

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. Your interpreter should understand the following symbolic constants and functions:

```
pi          3.14159265
e           2.71828183
ln()        Natural logarithm
log()       Base 10 logarithm
exp()       Powers of e
sin()       Sine
cos()       Cosine
tan()       Tangent
arcsin()    Arc sine
arccos()    Arc cosine
arctan()    Arc tangent
```

All the functions are available in the standard math library. See the Unix manual pages for details.

Implement error recovery in your parser. The simplest form of error recovery is to scan tokens to the end of a line and then resume parsing. Feel free to implement a smarter error recovery strategy.

**Hand in the following:**

- Printouts of all the files you modified or created.

- Answers to the questions in the next section.

- Test data that show that the program works as specified. Be sure to test error recovery, both from parser and scanner errors. Be sure to check that error recovery does not interfere with the next input line. Check that precedence and associativity rules are followed.

Demonstrate your solution to your lab assistant during a laboratory session. Send an e-mail (one e-mail per group) with your modified code and answers to the questions to the same assistant, put TDDD55, assignment number and your LiU logins in the e-mail subject line.

## 3   Questions

**Question 1** Define a regular expression for numeric constants. It should allow integers, numbers with a fractional part and numbers with an exponent. A number containing a decimal point must have at least one digit before or after the decimal point (or both). The exponent may have a sign, plus or minus, and is always an integer.

```
Allowed                 Not Allowed
1234                    A123
3.14                    .
.112                    112.a
112.                    1E2.3
12.34                   2.3e3.
```

```
34E-23                          23E 54
34.E+3
2.2e5
```

**Question 2** Construct a DFA that accepts the same language as the regular expression you defined in the previous question. Suggest how to implementa a scanner based on your DFA.

# 4   Supporting Programs

The files `lab1.cc` and `lab1.hh` contain a skeleton for the parser class and a class called `Trace` that can be used to trace invocation of functions. See the `Parser` method for an example of how to use it. Objects of the class print an entry message when created and an exit message when destroyed.

The files `lex.cc` and `lex.hh` contain a scanner class. To use it create an object of type `Scanner` and call its `Scan` method to get a token. Tokens returned are of type `Token`. See the comments in `lex.hh` for a description of how they work.

The file `main.cc` contains a sample main program. You may have to modify it depending on how you choose to report errors from your parser.

If the scanner encounters an error it will throw an object of type `ScannerError`. Your main program should catch this exception (the sample main program does), print an error message (you can print a `ScannerError` object using stream operators) and then perform error recovery.