Programming Exercise 1: Attribute Grammars and Top-Down Parsing

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-doown 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 associateive. 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. Add attributes to the grammar that specify the semantics of the language.

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:

- The grammars produced in each step. There should be one with left recursion and one with attributes that is free of left recursion.
- 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.

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 proogram 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.

5 Extra Credit Work: User-Defined Variables

Implement user-defined variables according to the following grammar (E is as before):

```
S       ->  Assign | E
Assign  ->  id ':=' E
```

After an assignment, a variable must be usable in the same way as a symbolic constant in the basic exercise. The predefined constants should be implemented in the same way as user-defined variables, but must not be changeable by the user.

The scanner is already capable of recognizing an assigment operator, so there should be no need to modify it.

To receive credit for this assignment you must implement a reasonably efficient symbol table. Lookup, insertion and deletion should be constant time operations. Solutions with a simple linked list and linear search are not acceptable.

Hand in your revised grammars, implementations and test sets.

6 Extra Credit Work: User-Defined Functions

Implement user-defined functions according to the examples below:

```
Expression: f(x) := sin(x) * sin(x)
Expression: f(1)
Result:     0.70807342
```

```
Expression: g(x,y) := sin(x) * cos(y)
Expression: g(1,2)
Result:     -0.35017549
```

To do this you will have to revise the grammar to allow function definitions and calls to functions with more than one argument. You will also need a symbol table that fulfills the requirements of the previous extra credit exercise.

The predefined functions are to be implemented using the same mechanisms as user-defined functions, but the user is not allowed to redefined them.

Hand in your revised grammars, implementations and test sets.