Software Project Spring 2005

Symbolic Math

1 Overview

The goal of this project is to implement a symbolic math interpreter and evaluator. Your program will parse symbolic function definitions and will be able to evaluate these functions given a parameter value; it will also compute the function derivative. Here is an example of a session with the symbolic interpreter. The lines starting with `'>'` are the input lines for the interpreter typed by the user, and all others are the output of the interpreter.

```
> f(x)=x^2+3
> f(x)

  f(x) = x^2 + 3

> Eval(f,4)

  f(4) = 19

> g(x) = f(x+1)
> Eval(g,4)

  g(4) = 28

> df(x) = f'(x)
> df(x)

  df(x) = 2 * x
```

2 Syntax

An input line for the interpreter can be one of the following:

1. Function definition
2. Function query
3. Evaluation
4. The command `who`
5. Remark

2.1 Function definition

Function definition will look like this:

```
FunctionName(x) = expression
```

The function name can be an arbitrary-length string containing capital and small letters only. All functions take a single formal parameter *x*. White spaces inside the function name are not allowed. The name `x` (small) as a function name is not allowed The interpreter is case sensitive; capital and small letters are considered *different* (thus, the function name `X` is allowed). White spaces after the function name, after the parameter or anywhere inside the expression are ignored. For the definition and syntax of expressions see Section 2.6. Examples:

The following are legal definitions:

```
> f (x) = x+1
> BabaYaga( x )=17
```

The following are *not* legal:

```
> g (y) = y+1
> BabaYaga(x) = y+17
> Serious27(x) = x^2
```

The function exists since the moment of its definition. Redefinitions of functions are possible, in which case the new definition replaces the old one. Example:

```
> f(x) = 2*x
> f(x)
  f(x) = 2 * x

> f(x) = 3 + x^2
> f(x)

  f(x) = 3 + x ^ 2
```

It is possible to define new functions using previously defined functions. The function expression is parsed immediately, so that if we define function *g* using function f and later change f, the function *g* will not be affected. Example:

```
> f(x) = 2*x
> f(x)
  f(x) = 2 * x

> g(x) = f(x+1)
> g(x)

  g(x) = 2 * (x + 1)

> f(x) = x + 3
> g(x)

  g(x) = 2 * (x + 1)
```

2.2 Function query

When we input the name of the function, followed by `(x)`, the expression of the function is printed out. Here too, white spaces around the function name and the formal parameter *x* are ignored. The output of the interpreter will be of the format `FunctionName(x) = function-expression`

Example:

```
> f(x) = 2*x
> f (  x)

  f (x) = 2 * x
```

2.3 Evaluation

We can ask the interpreter to evaluate a previously defined function at a given parameter value. The result of an evaluation is always a number. The syntax is as follows:

```
Eval(FunctionName, parameter_expression)
```

The `parameter_expression` must evaluate to a number. Therefore, it cannot contain the variable *x*, but only numbers, operators and `Eval` expressions. For more information on expressions see Section 2.6. The output of the interpreter will be of the following format:

```
FunctionName(number1) = number2
```

where `number1` is the number to which `parameter_expression` evaluates, and `number2` is the result of the function evaluation.

Examples:

```
> f(x) = 2*x
> Eval(f, 4)

  f(4) = 8

> Eval(f, 1*3 + 17)

  f(20) = 40

> g(x) = x+1
> Eval(f, Eval(g, 2))

  f(3) = 6

> Eval(f, 3 + Eval(g, 2))

  f(6) = 12
```

2.4 The command `who`

The command `who` prints the names of all currently defined functions. Example:

```
> f(x) = 2*x
> gaGa(x) = x+1
> gaga(x) = x-3
> who

  f gaGa gaga
```

2.5 Remark

A remark starts with the symbol # followed by any set of characters (until Enter). The interpreter ignores remarks and outputs nothing. No remarks are allowed in the middle of a line.

```
> # The following is the Taylor expansion of e^x till 4-th order
> f(x) = 1 + x + x^2/2 + x^3/6 + x^4/24
```

2.6 Expressions

An expression can consist of numbers, the variable *x*, allowed mathematical operations listed below, `Eval` (as explained above), derivation and function expressions. All white spaces within an expression are ignored.

Numbers are floating point of the form:
`[-]N[.M]`, that is: an optional '-' sign, followed by an integer, followed by an optional `.integer`. All of the following are legal: 1, -2, 3.1, -543.123

2.6.1 Allowed mathematical operations

In the following definitions, `a` and `b` will denote expressions.

- Plus. `a + b`
- Minus. `a - b`
- Mult. `a * b`
- Div. `a / b`
- Power. `a ^ b` Note: the exponent must be an expression that evaluates to a number. In other words, `b` cannot contain `x`, just like the `param-expression` in `Eval`.
- Sine. `sin(a)` During computation, the argument will be regarded in radians.
- Cosine. `cos(a)` During computation, the argument will be regarded in radians.
- Derivation. `FunctionName'(x)`. It is not allowed to put an expression instead of `x`.
- Function expression. `FunctionName(a)`. `FunctionName` should be the name of a previously defined function and `a` should be a legal expression (may contain `x`). The meaning is: substitute `x` with `a` in the expression of the function.

2.6.2 Operator precedence and brackets

Evaluating an expression means substituting any occurrence of `x` with a number, and then computing the result. Expressions should be evaluated using the regular precedence of mathematical operators:

```
17 + 6 / 3 = 17 + (6/3) = 19
2 ^ 2 + 1 = (2 ^2) + 1 = 5
2^2^3 = 2^(2^3) = 256
x^2/3 = (x^2)/3
```

Brackets () change the regular operator precedence. Only round brackets () are allowed in the syntax. It is allowed to put brackets around any expression.

```
(3+5)*2 = 8*2 = 16
```

2.6.3 Examples of legal expressions

Suppose we defined a function `f(x)` as `f(x) = x^2`. The following inputs to the interpreter are legal:

```
> g(x) = 3 + 17*(f(x/3) + 5.7) - sin(f(3/x))
> h(x) = g'(x) - 11 * Eval(f, Eval(g,4))
> k(x) = 3 + -17.5*(x + 2)                    the '-' is for the number value: -17.5
> d(x) = 3 * -17.5*(x + 2)                    the '-' is for the number value: -17.5
```

2.6.4 Examples of illegal expressions

```
> g(x) = Eval(f, x+1)   because the expression x+1 contains x
> h(x) = f'(x+1)        because derivation is only allowed at x
> k(x) = f'(4)          because derivation is only allowed at x
> m(x) = x - + 4        no unary '+' operator is defined (+4 is illegal)
> h(x) = x + - -5       -5 is OK, but two operators one after the other (+ -) are not
> s(x) = -x + y         no unary '-' operator is defined
```

3 Input and output

The program will run in two modes: interactive mode and batch mode. In the interactive mode, the interpreter prints the prompt `'>'` to the standard output and waits for a command. The user types in the command followed by Enter, and then the interpreter parses the commands and outputs to the standard output according to the definitions above. In the batch mode, the interpreter receives a file containing the commands (each in one line, followed by line break) and executes them one by one. The output is written to a specified file. The input file may contain remark lines starting with `'#'`. Empty command lines are ignored in both modes.

The command line that runs your project has two possible forms:

- `SymbMath -i`
  This opens the interactive mode, where the interpreter accepts commands from the standard input and outputs to the standard output.
- `SymbMath -b <input-script> <output-file>`
  Batch mode. The interpreter will read commands from `input-script` file, line by line, and output to `output-file`.

The number of arguments and their order amount should be strictly as written above. Example:
`SymbMath -b my_input.txt my_output.txt`
If the output file already exists, it will be overwritten.

3.0.5 Run-time output

- In interactive mode: if the interpreter detects a parsing error, it should output a description of the error and return to the prompt state `'>'`. If the error occurred in a line that was supposed to define a function, then the function will, of course, *not* be defined or redefined. In batch mode, also output the line number in the original input file, where the error occurred. Use the following to do so:

  ```
  printf("Error in line:%d, Description:%s\n", line_number, reason);
  ```

- Same instructions as above are also valid for illegal operations, such as attempt to divide by zero, attempt to evaluate a function with a parameter which is an expression that does not evaluate to a number, etc.
- If illegal arguments are passed to the program (like input file that does not exist or a wrong option), print what's wrong and exit.

4 Requirements

The project grade is made of the following parts:

1. Initial design document. 5% (not checked)
2. Working makefile 5%
3. Working SymbMath project. 50%
4. Documentation (described below) 30%
5. Three (3) non-trivial input scripts 10%

To pass the course, the project must work. To pass the course, **no part of the project can be copied.**

5 Programming Guidelines

The coding quality is an important part of this project. Following are some general guidelines for correct coding. Try following these guidelines, as well as employing any additional knowledge you have in writing software.

- Modularity. Related functions should reside in the same file, while unrelated functions are to reside in different files.
- Functionality. Each function should perform one *thing*. This *thing* should be the function name. Functions should be short and clear, function length should not exceed one page. Any repeated code parts should be concentrated in functions.
- The maximum line length in your code should not exceed 80 characters.
- Documentation. The code should be clear. Therefore, short comments should appear in various places, such as preceding functions or complicated blocks, when defining variables, etc. If there are assumptions regarding the input / output of the functions, they should appear too.
- Naming conventions. A significant part of the documentation resides in the code itself. Proper name choosing is one way of doing it. The names of the variables and functions should imply their role. Each group of names should have the same convention. For example, function names can have capital letters leading each word in the name, variables can use lowercase characters with underscores between words, and constants can use all capital letters: `GetNextLine (line_buffer, LINE_LEN);`
- Simple data-structures, algorithms and implementations are preferred.
- It is recommended to handle data structures only via dedicated functions. For example, if there is a linked list used in some places, and some algorithms go serially over the list, it is a good idea to have functions such as GetFirst and GetNext. The lists' elements should be accessed only via these functions.
- No compilation errors or warnings are permitted (you must compile with the `-Wall` flag).
- Obviously, no run time errors are acceptable (the program may not exit abnormally).
- A working project is more important than a fancy project. Make sure the basic features that are required here are working before adding anything to the project. Also, a not-so-efficient working project is much better than a very efficient non-working project.

6 Documentation

In addition to the code, you should also write a short description of the project. The documentation file should describe the entire project. It should be clear and built of nicely separated sections. Its length should be about 10 two-sided pages.

The idea is that when someone else (beside the author of the project) reads the documentation, she or he will be able to understand, modify and extend your project.

The documentation should include:

- Header - including the same information as in the partners file (see directory structure section bellow).
- A list of all the files in the project, and a short (one line) description of each one of them.
- The modules of the project and their relationships.
- A general description of the project logic and the application flow.
- A list of all the data structures in use, together with a short explanation why they were chosen. Include a description of the alternatives to the decision, and list the pros and cons of each alternative.
- For each data structure, a short time complexity analysis for each of the implemented actions.
- Comments and suggestions about the project.
- The documentation should be printed in English.

The documentation is not part of the code.

7 Administrative information and rules

- You should write the project in "C", or if you wish, "C++". The project should run on nova; try it on the UNIX machines in class Schreiber-04.
- Submission deadline: Wednesday 25-May-2005, 10am
- You **must** work in groups of **two** people.
- On the second class (one week after the semester begins), submit an initial design document of the project (1-2 pages). This document should include the participants' names, main modules and relations between them, and data structures that will be used. You will also be required to submit a toy project with a working makefile, as will be explained in the first class.
- **You should check the web page at least once a week for any updates on the project. The webpage is:**

  `http://www.cs.tau.ac.il/~sorkine/courses/proj05/`

  Some of the definitions of the project may change. Pay special attention to the forum. The questions and answers that are posted in the forum complement this document.
- After you submit your project, check your email regularly, you should expect a message with your run-time grade.

7.1 Directory Structure

Following all the guidelines is absolutely critical. Wrong directory structure will cause the run-time test to fail and will result in a zero run-time grade.

- There should be one directory under your account, named "SymbMathProject05" (case is important!) with no sub-directories.
- The name of the executable is `SymbMath`.
- The directory should contain a file named "`partners`" that includes:
  One line for each team member with his/her login name, ID number, and full name separated with white spaces.
  The files of the project should reside in the directory of the team member in the first line. Other member should have the same directory that includes only the `partners` file.
- All of the files (sources, headers, makefile) should be under that same directory. There should be no object files.
- There should be a working makefile named "`Makefile`". The makefile should generate the executable of the project. In addition, there should be a "clean" target in the makefile that will remove the object files from the directory.
- You must set permission for the `SymbMathProject05` directory and all the files under this directory. In the command prompt, type:

```
> cd
> chmod a+x ~
> chmod 755 SymbMathProject05
> chmod 644 SymbMathProject05/*
```

7.2 Before you submit, make sure

- There is no debug output to the project, unless it is for debug purposes, disabled by appropriate #if clauses.
- Your project works on the school UNIX systems.
- Your documentation includes the general description, as explained in the documentation section.
- Your code is well commented, as explained in the Programming Guidelines section.
- The project does not depend on any specific information in your account. A good idea is to copy the `SymbMathProject05` directory temporarily to another account and compile it there.

7.3 What to submit

- A hardcopy of the documentation.
- **No hardcopy of the source files please!** The checker will look at your code online.
- A hardcopy of the `partners` file.
- The electronic version of your work should reside in the proper location, untouched after the due date.

8 Grading Criteria

- Project submitted on time.
- Fully functional according to the specifications.
- The project compiles flawlessly.
- Code quality.
- Documentation.
- Reasonable efficiency: make a tradeoff between implementation time and efficiency, but be aware of it. When you make such a choice, explain your considerations in the documentation.
- Initial design submitted on time.
- It is OK to share ideas with other groups, but it is forbidden to share code.

Some remarks on getting a good grade:

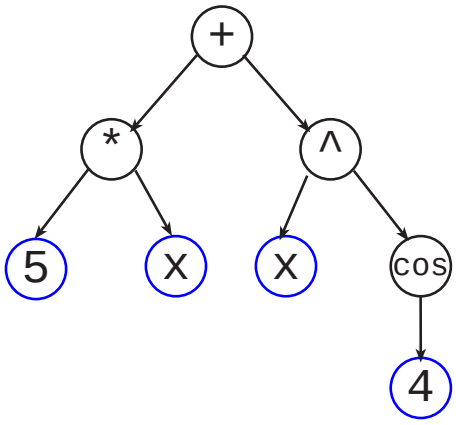
- Start working on the first day. The end of the semester is more stressed than the beginning.
- If something is not clear, ask.
- Do not leave things to the last minute. Make sure you have a running project ahead of time and test it.
- The documentation is an important part of the project. Write it during the coding process and not the night before you submit the project.
- Work with a team member you trust, check that both of you are on the right track every once in a while.

A Expression trees

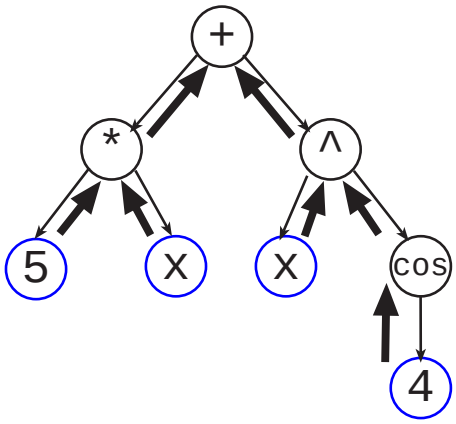
Expressions can be represented by trees. This enables their automatic evaluation and manipulation (such as computing the derivative). The inner nodes of the tree represent operators and the leaves contain numbers or the formal variable `x`.

Here is an example of an expression tree:

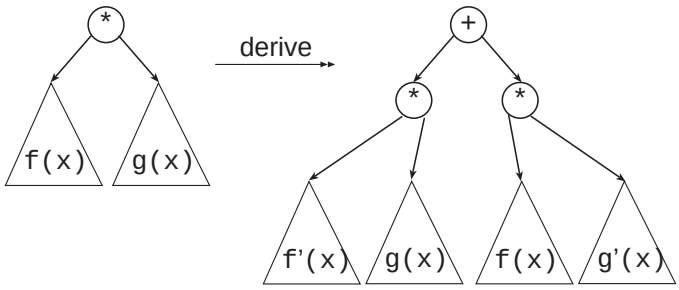
```
f(x) = 5*x + x^cos(4)
```



Evaluating an expression, when we replace all occurrences of `x` by a number, requires recursive traversal of the tree from the bottom up:



To perform symbolic derivation of a function expression, we need to change the topology of the tree (careful!):



B Changes

When	Where	What