Recursive Descent Parsing Calculator Assignment

Terence Munro

Contents

1. Problem Statement

The goal of Question 2 was to create a calculator that uses a technique referred to as Recursive Descent Parsing to parse an infix specified mathematical expression and produce a result. The calculator is only very basic and supports ‘+’, ‘-‘, ‘*’, ‘/’, ‘^’, ‘(‘, ‘)’ and decimal digits.

2. User Requirements

The following outlines the user requirements for the program:

The user should be able to:

- Calculate any well-formed mathematical expression that is created using the allowed symbols and operators.

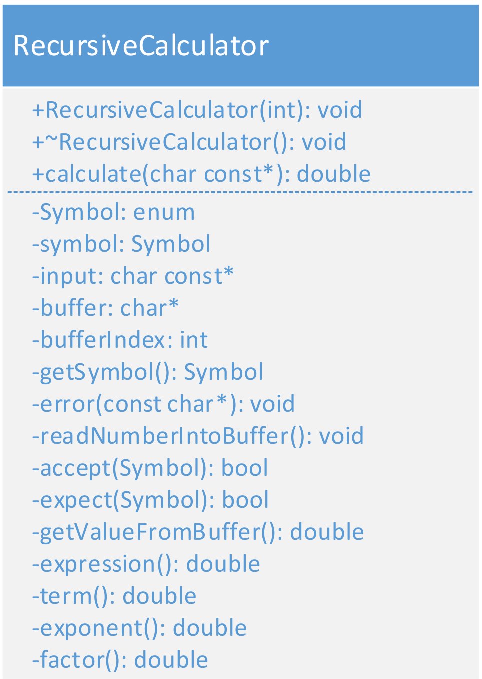
3. Software Requirements

The following outlines the software requirements for the program:

- The result should be displayed with 3 decimal precision
- The calculation should obey precedence rules.
- Needs to handle floating point numbers

4. Software Design

UML Diagram



RecursiveCalculator

+RecursiveCalculator(int): void
+~RecursiveCalculator(): void
+calculate(char const*): double

-Symbol: enum
-symbol: Symbol
-input: char const*
-buffer: char*
-bufferIndex: int
-getSymbol(): Symbol
-error(const char*): void
-readNumberIntoBuffer(): void
-accept(Symbol): bool
-expect(Symbol): bool
-getValueFromBuffer(): double
-expression(): double
-term(): double
-exponent(): double
-factor(): double

By using the recursive descent parsing methodology the form of the application looks quite close to the intended grammar it supports.

The grammar is as follows:

```
answer = expression
expression = term { [+|-] term }
term = exponent { [*|/] exponent }
exponent = factor { ^ factor }
factor = [+|-] digit | ( expression )
```

Where anything between the curly ({, }) braces are repeatable and | means or.
So any expression can be made up and will be evaluated obeying the order of operations as it returns back down from reaching the inner most function (factor).
The way it works recursively is that anytime it reaches a left parenthesis it will call the first expression function again and will continue until it reaches its ending right parenthesis (assuming all syntax is correct). This means you can have any number of parenthesisised expressions (also assuming you do not overflow your program stack) and the expression will always be handled obeying the order of operations.

This technique is quite easy to implement and additional features could be added quite easily such as variables and logic gates. A simple interpretter could even be made using with only a few additional lines of code.

Data structures in the software

Only data structures required for the calculator are:

- a pointer to where the parser is up to on the given input.
- a variable to hold the last found symbol
- a buffer to store numbers while they're being parsed (and this could have been removed also with a different technique).
- an index for the buffer (also only needed because of the choosen method to parse digits)

Everything else is handled by the program stack and recursion.

Detailed Design

Since all the intermediate steps between calculate and factor are very similar I would have liked to have implemented a generic function and used a function pointer for the differences but for now this is good enough.

```
Calculate
        Store pointer to start of user input.
        Get the first symbol
        Answer = expression
        Expect End symbol
        Return answer

Expression
        Get term and save into value
        While symbol is plus or minus
        Get next symbol
        Get next term
        If the symbol was plus
            Add term to value
        Else
            Minus term from value
    Return value

Term
    Get exponent and save into value
    While symbol is multiply or divide
        Get next symbol
        Get next exponent
        If the symbol was multiply
            Multiply value and exponent
        Else
            Divide value and exponent
    Return value

Exponent
    Get factor and save into value
    While symbol is Power (^)
        Get next symbol
        Get next factor
        Value = pow(value, factor)
    Return value

Factor
    If symbol is a digit
        Return buffer converted to decimal
    If symbol is (
        Return expression
    Otherwise
        Syntax error
```

5. Requirement Acceptance Tests

Software Requirement No	Test	Implemented (Full/Partial/None)	Test Results (Pass/Fail)	Comments (for partial implementation or failed test results)
1	User can enter expressions	Full	Pass	
2	User receives error message if there is a syntax error	Full	Pass	

6. Detailed Software Testing

Unit tests are only available in the 2013 project as the 2008 edition of Visual Studio uses a different framework and it would have been to much extra time to convert the syntax

No	**Test**	**Expected Results**	**Actual Results**
1.0	**Test Program**		**Screenshot provided in appendix**
1.1	2^2 + 3 – 4	3	3.000
1.2	(2 + 3) * 5	25	25.000
1.3	(2 + 3) * (5- 1)^2	80	80.000
1.4	2 + ((3+1)*(6-2)-1)/(4-2)	9.5	9.500
1.5	2 + 3 * (1 + 4) – 9 / 2 * 3 + 1	4.5	4.500
2.0	**Full Unit Testing examples given below**	**Results in the unit test**	**As expected**

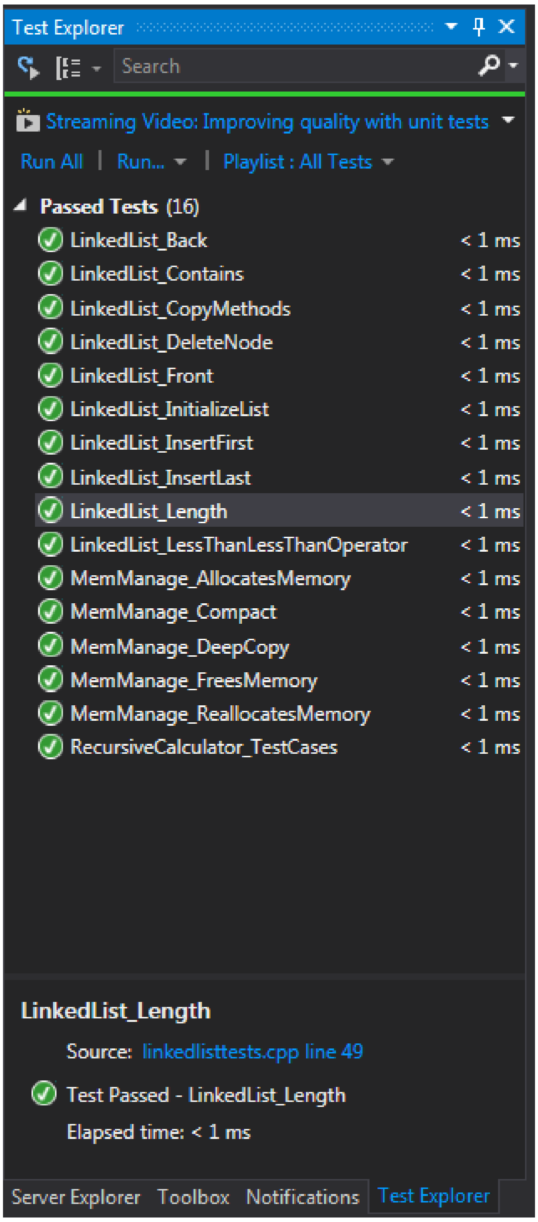
Example unit test:

```
#include ”stdafx.h”
#include ”CppUnitTest.h”
#include ”..\RecursiveCalculator\RecursiveCalculator.h”

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTests
{
    struct Answer
    {
        const char* expression;
        double          answer;
    };

    TEST_CLASS(RecursiveCalculatorTests)
    {
    public:
        TEST_METHOD(RecursiveCalculator_TestCases)
        {
            RecursiveCalculator calc;

            Answer answers[] = {
                { ”1 + 1”,                2 },
                { ”3 + 2”,                5 },
                { ”1 + 23 + 102”,         126 },
                { ”12 - 3”,               9 },
                { ”3 - 4”,                -1 },
                { ”4 - 1 - 2”,            1 },
                { ”5 * 2”,                10 },
                { ”5 * 2 * 3”,            30 },
                { ”3 / 1”,                3 },
                { ”1 / 4”,                0.25 },
                { "12 / 3 / 2",           2 },
                { "1 + 2 - 3",            0 },
                { "1 + 2 * 3",            7 },
                { "6 / 2 - 1",            2 },
                { "3 * 2 / 3",            2 },
                { "0.2 + 0.3",            0.5 },
                { "1 + 0.2 - 0.3",  1 + 0.2 - 0.3 },
                { "3+4/2*4", 3 + 4 / 2 * 4 },
                { "2^2 + 3 - 4", pow(2, 2) + 3 - 4 },
                { "(2 + 3) * 5", (2 + 3) * 5 },
                { "(2 + 3) * (5 - 1)^2", (2 + 3) * pow((5 - 1), 2) },
                { "2 + ((3 + 1) * (6 - 2) - 1) / (4 - 2)", (2.0 + ((3.0 + 1.0) * (6.0 - 2.0) - 1.0) / (4.0 - 2.0))
},
                { "2 + 3 * (1 + 4) - 9 / 2 * 3 + 1", (2.0 + 3.0 * (1.0 + 4.0) - 9.0 / 2.0 * 3.0 + 1.0) },
                { "-1 + -4", -1 + -4 }
            };

            for (int i = 0; i < sizeof(answers) / sizeof(Answer); ++i)
            {
                size_t n;
                wchar_t errMsg[255];
                mbstowcs_s(&n, errMsg, 255, answers[i].expression, strlen(answers[i].expression));

                Assert::AreEqual<double>(
                    answers[i].answer,
                    calc.calculate(answers[i].expression),
                    errMsg);
            }
        }
    };
}
```

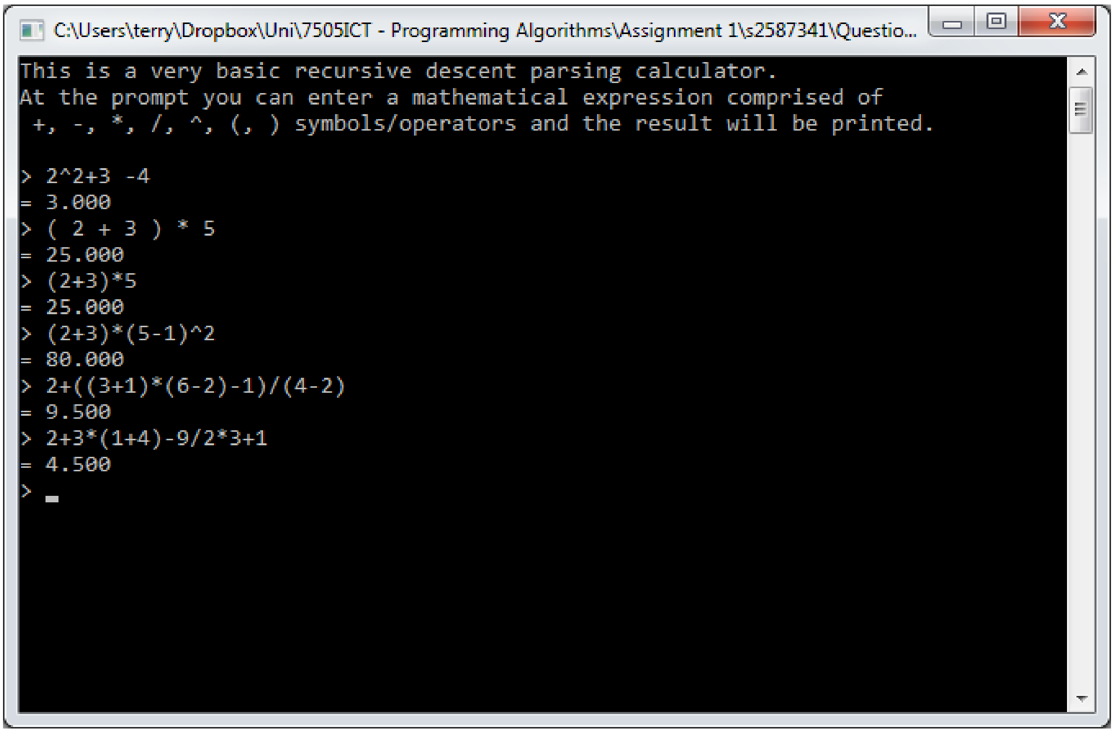
7. User Instructions

- User can load up the project using the Visual Studio 2008 solution file in the same directory as this Documentation.
- Statically compiled executable is available in the Release folder
- If the user wanted run user tests themselves they need to use Visual Studio 2013 and load up the VS2013 solution in a folder external to the question 1 and 2 folders labelled VS2013

8. Appendix



```
Passed Tests (16)
  LinkedList_Back                           < 1 ms
  LinkedList_Contains                       < 1 ms
  LinkedList_CopyMethods                    < 1 ms
  LinkedList_DeleteNode                     < 1 ms
  LinkedList_Front                          < 1 ms
  LinkedList_InitializeList                 < 1 ms
  LinkedList_InsertFirst                    < 1 ms
  LinkedList_InsertLast                     < 1 ms
  LinkedList_Length                         < 1 ms
  LinkedList_LessThanLessThanOperator       < 1 ms
  MemManage_AllocatesMemory                 < 1 ms
  MemManage_Compact                         < 1 ms
  MemManage_DeepCopy                        < 1 ms
  MemManage_FreesMemory                     < 1 ms
  MemManage_ReallocatesMemory               < 1 ms
  RecursiveCalculator_TestCases             < 1 ms

LinkedList_Length
    Source: linkedlisttests.cpp line 49
    Test Passed - LinkedList_Length
    Elapsed time: < 1 ms
```


C:\Users\terry\Dropbox\Uni\7505ICT - Programming Algorithms\Assignment 1\s2587341\Questio...

```
This is a very basic recursive descent parsing calculator.
At the prompt you can enter a mathematical expression comprised of
 +, -, *, /, ^, (, ) symbols/operators and the result will be printed.

> 2^2+3 -4
= 3.000
> ( 2 + 3 ) * 5
= 25.000
> (2+3)*5
= 25.000
> (2+3)*(5-1)^2
= 80.000
> 2+((3+1)*(6-2)-1)/(4-2)
= 9.500
> 2+3*(1+4)-9/2*3+1
= 4.500
> _
```