A Maths Interpreter

Charlie R

1st February 2023

CONTENTS

Abstract

...

CHAPTER 1

ABOUT

1.1 What is this?

This is a simple, command-line, tool to parse mathematical expressions.

Features

In the following:

- '*' indicates it is not yet implemented.
- Inputs are red, Outputs are blue.

This program provides the following features:

- Evaluating expressions (ie., $2 * (4 + 9) = 26$).
* Comparing expressions (ie., $2 * (a + 3) = 2 * a + 6$ true).
* Symbol assignment and memory (ie., $a = 6$; $2 * a = 12$).
* LaTeXsupport (ie., `3 \cdot 9` $= 27$)

1.2 But why?

Because I wanted to.

1.3 Capabilities

As of 1st February 2023, the program is capable of the following.

Tokenising

Type	Regex[1]
Integers	`/-{0,1}[0-9]*\.{0,1}[0-9]+/`
Symbols	`/[a-zA-Z]+/`
Operators	`/[+\-/*^=%]/`
Braces	

Evaluating

- Operators whose children are all numeric.

[1] This program doesn't use regexes but it is an easy way to describe strings.

CHAPTER 2

OVERVIEW

This chapter will focus on an overview of how the program functions and is split into sections based on the directory listing:

```
C-Maths
├── Errors.h
├── Evaluate.h
├── Functions.h
├── Maths.c
├── Structures.h
├── Symbols.h
├── Tokenise.h
└── ToTree.h
```

2.1 Maths.c

'Maths.c' is the main program that draws everything together. The header files can be used in isolation and this file simply provides an interface with the following.

2.2 The Main Parts

2.2.1 Tokenising

The first step in interpreting maths is to tokenise the string; this is done in 'Tokeniser.h'. At a high level, this works by iterating over the string and forming 'tokens' from the characters/sub-strings therein. Token types used in this program are:

- Null (`T_NULL`)
- Functions (`T_FUNC`)
- Operators (`T_OP`)
- Symbols[1] (`T_SYM`)
- Open braces (`T_OPEN`)
- Close braces (`T_CLOSE`)
- Numbers (`T_NUM`)

2.2.2 Converting to a Tree

To parse the expression, this program uses an abstract syntax tree[2]. To form this form an array of tokens, the program uses:

1. The Shunting Yard Algorithm, then
2. A stack-based algorithm.

Shunting Yard

The Shunting Yard Algorithm[3] is used to convert an 'infix' expression to a 'postfix' expression.

Infix is the common form we use to express maths – with an operator between the operands; postfix, however, is easier for programs to parse and has the operator after the operand. For example, the infix expression $2 \cdot (3 + a)$ would be equivalent to the postfix $3\ a\ +\ 2\ \cdot$.

[1] 'Symbols' in this program are akin to variables. For example, in $n_a + 7 = y$, n_a and y are symbols.
[2] See: https://en.wikipedia.org/wiki/Abstract_syntax_tree.
[3] See: https://en.wikipedia.org/wiki/Shunting_yard_algorithm.

Forming The Tree

Once the tokens are in postfix form, the tree can be formed. This is done through a simple algorithm that iterates over the tokens and:

- If the token is an operand (number or symbol), a node with the same value is pushed to the stack.
- If the token is an operator, nodes are popped from the stack (the amount of nodes is determined by how many operands the operator expects) and a new node is pushed with the value being the operator, and the children being the operands.

Once complete, the last (and only) value on the stack is the head of the tree.

2.2.3 Evaluation

Evaluating the tree is the (in my opinion, at least) most important part of this program – and the part I did without consulting any normal standards; the following algorithms I made up myself. This part works in 5 main "rounds".

Round 1 – Replacing Symbols

To begin, the tree is traversed and – consulting the symbol table – symbols are replaced with their respective value. This value will only exist if it has been set in a previous input. For example, the user enters $a = 5 + b$ and then $a \cdot 2$; after this round, the tree would be equivalent to $(5 + b) \cdot 2$.

Round 2 – Numeric Evaluation

The second round evaluates functions/operators whose children are all numeric. For example, this would parse $3 + 9$ but not $a - 2$.

Round 3+

To be implemented.

2.3 Misc.

2.3.1 Errors.h

'Errors.h' provides functions for when a program produces a fatal error. For example, '`err_red_location`' which outputs the location and error passed (with colour-based formatting) and then exits.

2.3.2 Structures.h

'Structures.h' provides the majority of structures used throughout, as well as the necessary accompanying functions. Such structures include:

- Tree nodes,
- Stacks,
- Queues, and
- Symbols.

CHAPTER 3

DEVELOPMENT

The following are instructions for adding to this program.

3.1 Adding an Operator (length 1)

To add an operator that is represented by one character, you must do the following:

Functions.h

- Implement the necessary functions for the backend of the operator; these functions must have the following signature[1][2]:

  ```
  T_Tree_Node function(T_Tree_Node *args, unsigned int arg_n)
  ```

- Add to the switch-case in the appropriate function:

Numeric	`get_numeric_func`
Symbolic	Not yet implemented

[1] `arg_n` will be known for most functions but is there to allow for functions with multiple, different, numbers of operators.

[2] Examples of how these functions should work can be seen in Functions.h.

Tokenise.h

- Add to the `OPERATORS` array.
- Increment `OP_NUM`.

ToTree.h

- Add to the `N_ARG_LOOKUP` array accordingly ({`<op/func>, <op/func length>, <number of arguments>`}).
- Increment `N_ARG_LOOKUP_LEN`.
- Add to the switch-case in the `precedence` function.

CHAPTER 4

APPENDIX

...