msdscript documentation

Welcome to msdscript! If you are planning on using this tool in your program, follow this description of how to use the script, how to build the program, and an in-depth description of what the different classes and methods do if you intend to make changes to suit your needs.

The script:

The script itself quite basic. It is used to type mathematical expressions, including addition, multiplication, functions, Booleans, and if-then statements. There is no support for subtraction or division.

Basic arithmetic:

To perform simple scripts, such as adding or multiplying one number to another, you simply type in the numbers with a plus sign (“+”) or multiply sign (“\*”) between them, as follows:

1+1

Then, hit enter to go to a new line, and press ctrl+d to execute. The interpreted expression will appear on the next line, followed with a capital “D”, you can ignore that “D”. The expression result should appear like this after following those instructions:

1+1

2D

You can include spaces or not, msdscript will read the following script the same:

1 + 1

2D

When typing longer expressions, such as strings of simple addition and multiplication, there is no need to use parentheses, though you may if you wish. With or without them msdscript will follow the standard order of operations for addition and subtraction. For example:

2+3\*3

and

2+(3\*3)

will both interpret to

11

Multiplying pairs of numbers being added together do require parentheses, just like in the standard order of operations:

(3+3)\*(5+5)

60

while

3+3\*5+5

23

Nested expressions:

The way we accomplish the order of operations is with the idea of nested expressions. Each script you run is an expression, like a number or a variable, but it could also be an expression made up of other expressions. For example, we could think of the expression

1+1

as

[expression]+[expression]

Following this, writing the expression:

(3+3)\*(5+5)

But following the idea of nested loops would go like this:

[expression (3+3)\*(5+5)]

↓

[expression (3+3)] \* [expression (5+5)]

↓

([expression 3] + [expression 3]) \* ([expression 5] + [expression 5])

Using variables:

msdscript supports the use of variables, but only if those variables are a lower-case or upper-case letter. Variables must be defined at some point. For example, running a script with just a variable would throw an error as follows:

x

libc++abi.dylib: terminating with uncaught exception of type std::runtime\_error: free variable: x

To define variables, you need to use a let statement. A let statement takes a variable and sets it equal to an expression within another expression. The format is as follows:

\_let [variable] = [expression] \_in [expression]

The keywords \_let and \_in preceded by an underscore are essential. Filling in values for the example would look like this:

\_let x = 12 \_in 3 + x

15

If you don’t put the variable in the \_in statement, then it would interpret like this:

\_let x = 12 \_in 3

3

You can nest expressions in let statements like this:

\_let x = 12 \* 5 \_in x + 12

72

Boolean and comparative expressions:

msdscript has support for Boolean expressions, using the keywords \_true or \_false. Used on its own, a Boolean expression will interpret to itself, like this:

\_true

\_true

Or

\_false

\_false

Booleans get used in other expressions, such as if-then statements.

Conditional expressions:

There is support for conditional expressions, specifically if-then statements. There are three parts to a if-then statement, \_if, \_then, and \_else, in this format:

\_if [expression] \_then [expression] \_else [expression]

If the \_if expression interprets as true, then the \_then expression is executed. If the \_if expression interprets to false, then the \_else expression executes. These make use of Boolean expressions. Example:

\_if \_true \_then 1 \_else 2

1

\_if \_false \_then 1 \_else 2

2

In the \_if expression you can also use conditional statements that, when in an if statement, will interpret to either true or false. These use the double equals sign (“==”). Example:

\_if 1 == 1 \_then 1 \_else 2

1

\_if 1 ==2 \_then 1 \_else 2

2

You can complicate this by using nested expressions here as well:

\_if \_let x = 1 \_in x + 2 == 3 \_then 4 \_else 5\*5

4

To build msdscript:

If you want to experiment with msdscript, you will need to make a main.cpp file in the src folder. That file can be as simple as this:

#include <iostream>

#include "cmdline.h"

#include "catch.h"

int main(int argc, char \*\*argv) {

use\_arguments(argc, argv);

return 0;

}

On an apple operating system: navigate to the folder where you opened this file, and run the command:

make

That’s it! The Makefile found in the msdscript folder will run the commands needed to make the object files and build the executable msdscript. If you add a main.cpp file, the Makefile will automatically add it to the commands and build it.

To use msdscript in the terminal:

To start the script interpreter in the terminal after you’ve built it with a main.cpp file, run the command:

./msdscript --step

This will enter “step interp mode”, which is the base mode for the program, and what you would need to follow along with the examples in above. There are other modes, to enter a certain mode run a command following this format:

./msdscript [tag]

here is a list with a description of each tag and the mode that you enter using it:

|  |  |  |
| --- | --- | --- |
| Tag | Mode | Description |
| --step | Step interp mode | Runs the interpreter using continuation loops, which avoid traditional recursion and allows more complex programs to be run, such as counting down from 1,000,000. |
| --interp | Interp mode | Runs the interpreter using traditional recursion, which works fine for most things, but not for scripts that need a lot of steps, such as counting down from 1,000,000. |
| --print | Print mode | Will just take what is typed, parse it into expressions, and then print it back. |
| --pretty-print | Pretty-print mode | Will do that same as above, but will print using a custom formatting that you can implement if you choose. Not currently implemented, it just runs like print mode. |
| --help | Help mode | Not currently implemented. Should print a list of commands and mode options. |
| --test | Test mode | Runs built-in tests using the catch.h file. This is very useful if you are altering the program for your own purposes. |

Some example scripts:

This script will compute 10 factorial, which is 3628800:

\_let factrl = \_fun (factrl)  
        \_fun (x)  
         \_if x == 1  
         \_then 1  
         \_else x \* factrl(factrl)(x + -1)  
\_in factrl(factrl)(10)

This script will count down from 1,000,000. It will interpret to 0. This only works using step interp mode, not with interp mode:

\_let countdown = \_fun(countdown)  
          \_fun(n)  
           \_if n == 0  
           \_then 0  
           \_else countdown(countdown)(n + -1)  
\_in countdown(countdown)(1000000)

For further information about how the program works and what individual classes and methods do, see the file msdscriptAPI.docx