MSDscript is a (very) limited general purpose programming language that runs as a command line tool. It can take in code, parse it, and return output. It is built on C++ and can be used as a C++ library.

It can be used as a (very) basic calculator. If you pass in 1+1, it will return 2. It has integers, Booleans, variables, addition, multiplication, double equals (==), if statements, functions, function calls, and recursion (through function calls). It can either interpret (which returns a value if possible, else errors), or optimize (which can simplify statements).

**Installing MSDscript:**

This works for macOS, I tried on a Linux subsystem for windows10 and it did not work.

(make sure you have g++ installed)

Navigate into DocumentationAndPackaging/ after unzipping and compile with the following command.

$ g++ -std=c++17 src/ArithmeticParser/\*.cpp -o MSDscript

You now have an executable in the current directory, run it with the following instructions.

**Running MSDscript:**

$ ./MSDscript <optional flags>

10 + 10 user input (std::cin)

<control d> this ends the input

20 user output (std::cout)

You may see a D after the output if the output is one character long, ignore this. It is because of the control d.

**Using MSDscript as a library:**

**Parsing:**

MSDscript takes in a std::cin using parse(std::cin) or a string using parseStr(std::string string). They both return an object expression representing the parsed input.

Ex. parseStr(“10 + 1”) would return an add object containing two number objects, 10 and 1 respectively.

**Interpreting and optimizing parsed expressions:**

There are currently 3 functions to work with expression objects, interpret, interpret by steps, and optimize. They are covered more in detail in their respective sections below. Here is how you use them in a library.

The expression above can be combined with ->interpret(NEW(EmptyEnv)()) to get the actual value (11) as a number object. It can then be combined with a ->toString() to get the string representation.

parseStr(“10 + 1”)->interpret(NEW(EmptyEnv)())->toString() == “11”;

The NEW(EmptyEnv)() makes a new empty environment, it is not used in this case, this is useful for more complex expressions to make them work as one might expect.

Interpret by steps is also an option. Step::interpBySteps( parseStr(“10 + 1”) )->toString() == “11”;

It could also be combined with ->optimize() to optimize the expression. In the case above where two numbers are added ->optimize() does the same thing as interpret. This is explained in more detail below in the interpret and optimize sections below.

**Linking MSDscript as a library:**

This is the simplest way to link libraries.

clang++ -std=c++17 -c <your file>.cpp

//*This will make an object file (.o) to be used later*

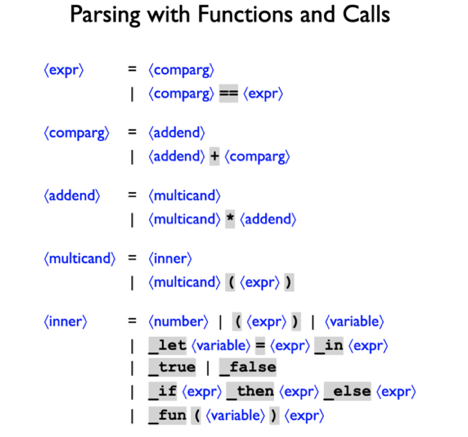
clang++ -std=c++17 -o <new program name> <file to be linked>.o library.a

//*This will link your file with the library*

You can now ./<new program name> to run it.

**Details**

**Grammar:**



**Integers**:

Regular signed integers, c++ style. Negative numbers are allowed, subtraction is not directly allowed, but you can do 1 + -2 instead of 1 – 2.

**Booleans:**

Regular Booleans \_true = true, \_false = false

**Variables:**

Variables are a string that can have an assigned primitive value. They can also be assigned to a different variable or function.

Variables are assigned with the following syntax.

\_let <variable> = <expression> \_in <expression>

\_let x = 5 \_in x

This assigned the variable x the integer value 5 in the expression x.

A variable bound by `\_let` or `\_fun` is visible only in the `\_let` or `\_fun` body.

Functions are described more in detail below.

Variables are case-sensitive.

**Operations**:

Addition/Multiplication/double equals

These are standard operations in most cases. They follow the standard order of operations (PEMDAS, except its more like PMA). Parenthesis are allowed to change the order. 2 + 3 \* 3 = 11

<expression> + <expression>

1 + 1 => 2

<expression> \* <expression>

10\*10 => 100

<expression> == <expression>

3 == 3 => \_true

It gets more complicated with things that are not primitives and will never evaluate to primitives.

Most of the time it will either return \_false or throw an error. If you try to interpret unassigned variables it will throw and error. Optimizing is fine though.

x + 5 => x + 5 (optimize)

x + 5 => throws (interpret)

If x is assigned a value for either, it will interpret as you would expect, substituting the value with x and evaluating.

**If statements:**

Typical if statements with the following syntax.

\_if <conditional> \_then <if true expression> \_else <if false expression>

For example.

\_if \_true \_then 3 \_else 4 => 3

\_if \_false \_then 3 \_else 4 => 4

**Functions and function calls:**

Single argument functions only.

The following is a lambda function. It has no name. It is also not called.

\_fun(<parameter>) <expression>

This is also a lambda function, but it is called. The <value to be passed in> is passed into the <parameter>. When a function is called, the <parameter> is assigned the value <value to be passed in> and all instances of the <parameter> in the <expression> are replaced with the <parameter>’s value.

(\_fun(<parameter>) <expression>)(<value to be passed in>)

This is not a lambda function; it has a name.

\_let <function name> = \_fun(<parameter>) <expression>

\_let f = \_fun(x) x + 2

It is not called yet, but can be called like so.

<function name> (<function parameter>)

f(3) => (3) + 2 => 5

**Interpret:**

The by default (no flags).

The interpret mode will either return a value (number or Boolean) , or an error.

1 + 2 => 3

10 \* 10 => 100

X + 5 => throws error: free variable

**Interpret By steps:**

The “-step” command line flag will turn the interpret by steps mode on.

Works mostly the same as Interpret. The distinction is how the memory is managed. It may run slower, but it won’t stack overflow on large recursive loops.

**Optimize:**

The “-opt” command line flag will turn the optimize mode on.

It will either return a syntax error, or the optimized parsed expression. For example, x + 5 with -int on will throw an error, but in -opt it will return x + 5. This is because it is not looking for a value, but is looking to simplify the expression then return it. It can handle unassigned variables.

5 + 5 will still return 10, but x + x will return x + x.

\_let x = 5 \_in x + x => 10 (substitutes and evaluates)

However

\_let x = y \_in x + x => \_let x = y \_in x + x (does not simplify)

The basic idea is that it will not optimize if it might make more work.