Table of Contents

Description

What does MSDScript do?

MSDScript is a language built to take in, interpret, and solve various mathematical expressions. Although basic mathematical expressions can be easily solved, pieces of MSDScript can be cobbled together in order to create more complex math problems, as well as performing more complex operations. Examples will follow throughout the documentation which will make it more obvious what kinds of simple and complex expressions MSDScript can solve.

What problems does MSDScript solve?

MSDScript provides a simple yet robust way to calculate the value of longer mathematical expressions without the need to break them down like you would have to by using a calculator. For example, the expression 2 * (3 + 2) would have to be entered into a calculator by typing 3, then +, then 2, then *, then 2, then =. Using MSDScript (from a full construction) you can simply type in the expression and MSDScript will evaluate it for you.

Additionally, MSDScript will allow you to evaluate more complex expressions and perform even more exciting operations. There are functionalities to consider mathematical functions, conditional statements, equivalence statements, Boolean expressions, and others. This will all be covered in greater detail in the User Guide portion.

How is MSDScript meant to be used?

MSDScript was originally built as a way to interpret and evaluate basic mathematical expressions. It has since grown dramatically to encompass a number of different mathematical expressions, functions, and operations, allowing it to be used as an extremely versatile and powerful mathematical interpreter (assuming you don't want to subtract or divide).

Getting Started

Installation

We'll be using CMake to build the executable for MSDScript. It's powerful and simple to use, which makes it ideal for this build.

These are the steps necessary to build your MSDScript executable:

1. Download the appropriate version of CMake from https://cmake.org/download/.
2. Run the file you just downloaded to execute the installer.
3. Unzip the file, "MSDScript.zip".
4. Using the terminal, navigate inside the now-unzipped file "MSDScriptApplication".
5. Enter the command: 'cmake .'.
    a. Note the period on the end of the command. This is important because that period is what tells CMake to build the executable using the information in your current directory.
6. Enter the command: 'cmake --build .'.
    a. Again, making sure to put the period at the end.
7. You've built your executable! Congratulations! The executable is called **MSDScript**.
8. These instructions also generate a library called **libMSDScript.a**. The instructions for how to use and implement this library are included below.

Basic usage - Application

Using the MSDScript application is super simple:

1. Navigate to the directory that contains your executable in the terminal.

2. Run your executable using the command './MSDScript'.
3. Once your executable is running, type in the mathematical expression you want to solve.
    a. Please note, MSDScript currently only supports addition and multiplication.
4. Press Enter/Return.
5. Then press Ctrl+D
6. See the result
    a. For results that are a single digit (for example 2+3 = 5), note that a 'D' will follow the answer, but for other calculations, this isn't a problem

Basic usage - Library

libMSDScript.a can be used the same way as other library files. You can compile and run your own main C++ file using this library using this command in your terminal:

```
clang++ -o <name of your new executable> <your main file> filepath/libMSDScript.a
```

The filepath before lib/MSDScript.a is optional if you have already navigated to that directory where your libMSDScript.a library file is located.

Note that in the command above, you do not need to include the angle brackets (<>) around the file names. Those are simply meant as placeholders. An example of running this command with an executable of **MyExec** and a main file called **MyMain.cpp** would be as follows (assuming you were in the same directory as libMSDScript.a):

```
clang++ -o MyExec MyMain.cpp libMSDScript.a
```

Basic User Guide for the Application

Using MSDScript in Optimize Mode

You can run the executable in optimize mode by using the --opt flag. In order to include the flag add it to your command when you run the executable:

```
./MSDScript --opt
```

Optimize is used to return a (potentially) simpler form of the input expression. For example, optimizing the expression 2*3+z would return 6*z. In the absence of variables, optimize and normal interpret mode will return the same answer.

The process for running and returning the executable is that same in optimize mode as it is in interpret mode:

1. After entering the executable command enter the expression you want to optimize
2. Press Enter/Return

3. Press Ctrl+D
4. See the result
    a. Again, if the result is a single digit, it will be appended by a D

Using MSDScript in Interpret By Steps Mode

You can run the executable in interpret by steps mode by using the --step flag. Similar to using the --opt flag, the --step flag is included in the following way:

```
./MSDScript --step
```

Interpret by steps mode is used to interpret expressions that are too big to be interpreted on the stack alone. Examples would be calculating particularly large Fibonacci sequences or performing computations that require a significant number of steps (i.e. 100 or more). If MSDScript closes because of a segmentation fault when running in normal interpret mode, interpret by steps mode will likely find the solution.

Expected Syntax - What MSDScript can do

Although MSDScript is currently limited to addition and multiplication in terms of mathematical operations, many other functionalities are possible if the proper syntax is given. The following is a comprehensive list of the basic syntax that will allow the interpreter to function and not throw errors. This is not meant to be a comprehensive list of all the possible combinations of syntax. There are many ways to expand the functionality of MSDScript through syntactical combinations and mixings that go beyond the scope of this user guide though this user guide will not be going into the details of how to mix all the syntax. It should be noted that whitespace is never a problem between pieces of syntax. However, all numbers, keywords, variables, etc. cannot have any whitespace in them. Keywords must begin with an (_) underscore.

Basic Mathematical Operators

MSDScript currently only supports the following mathematical operators: + for addition and * for multiplication. It does also have support for negative numbers, so there is the possibility to do subtraction by adding a negative number.
No special syntax is required to enter an operator as long as it would make mathematical sense to do so.
Examples:

2 * 3 = OK	5 + _false = NOT OK
5 + x = OK	5 * * 2 = NOT OK
x * y = OK	7 + * 4 = NOT OK

Numerical Expressions

MSDScript can handle any integers ranging from -2147483648 to 2147483647 (the INT_MIN and INT_MAX of C++). No special syntax is required to enter a number as long as there are no spaces within a single number. Because only integers values are supported, decimal numbers are not supported.
Examples:

2 = OK
133 = OK
-27 = OK

2   7 = NOT OK
1.7 = NOT OK
2x = NOT OK

Please note that using integers around the upper and lower limits may produce results you may not expect. Integers wrap around the range in a two's compliment manner. Two's compliment means that if you add past the INT_MAX then the result will wrap around from the INT_MIN and vice-versa. For example: 214748364 + 1 = -2147483648. The opposite is also true: -2147483648 - 1 = 2147483647. Moving beyond that range will continue as well. For example: 214748364 + 5 = -2147483643. This wrapping behavior can be expected for all numerical operations.

Variable Expressions

Variable expressions can be any collection of alphabetic characters only. Both uppercase and lowercase letters are acceptable as long as there aren't any spaces within a single word. Syntax requires variables to be entered with 1) only alphabetic characters, 2) there are no spaces within a single variable name.
Examples:

x = OK
X = OK
variable = OK
true = OK

this is a variable = NOT OK
va r = NOT OK
j8 = NOT OK
_true = NOT OK

Boolean Expressions

Boolean expressions are considered a keyword within MSDScript. As such, Boolean expression must begin with an underscore (_) in order to be considered valid with no space between the underscore and the keyword. Otherwise it will be considered a variable (as noted above in Variable Expressions.
Examples:

_true = OK
_false = OK

true = NOT OK
false = NOT OK
_tr ue = NOT OK
_ false = NOT OK

Substitution Expressions

Substitution expressions are created to substitute a variable with a number when appropriate. The syntax for these kinds of expressions are very particular:

_let ***variable* = *expression*** _in ***expression***

The syntax is the keyword **_let**, a variable, an =, another expression, the keyword **_in**, and a final expression. A variable can be any valid variable expression (as defined above), and an expression can be any valid mathematical expression.

It should be noted, that if the first expression (the one between the = and _in) contains a variable, **it will not substitute into the expression**. In order to substitute an expression must be a valid interpretable expression. Expressions with free variables are not interpretable Examples:

_let x = 5 _in x * 2 = OK ⇒ will interpret to 10
_let jam = 5 + 2 _in jam + 5 = OK ⇒ will interpret to 12
_let z = 7 _in 2 + 3 = OK ⇒ will interpret to 5
_let y = 7 _in x + 2 = OK
In this case, x is a free variable because it won't be substituted by y. This means that this expression will not be interpretable. However, this expression can be optimized using the optimization flag. The optimized output would be x + 2.
_let x = z + 2 _in x * 17 = OK
Because a free variable exists in the first expression, it will not substitute into the expression x * 17. Additionally, because the free variable exists in the first expression, the expression won't collapse if optimized. The optimized output would be identical to the input.

As you can see, as we get to higher up expressions, the flexibility becomes much greater. Many different permutations of substitution expressions can be entered, but not all will produce meaningful results. There are substitution expressions that would crash MSDScript. Below are some examples:

_let 8 = 5 _in 8 + 2 = NOT OK ⇒ 8 is not a valid variable name
_let _true = _false _in _false = NOT OK ⇒ _true is not a valid variable name
_let x = _true _in x * 8 ⇒ Expressions with Booleans in them can't be interpreted (i.e. _true * 8)

Equivalence Expressions

Equivalence expressions are used to determine if values or expressions are equal to each other. Their syntax is relatively simple:

expression == ***expression***

Broken down, the syntax is an expression, two =, and another expression.

On their own, equivalence expressions don't do much. It is generally when equivalence expressions are combined with other that they become more useful. They are particularly helpful when evaluating conditional expressions (see below) as they allow a finer degree of control over when certain outputs are produced, rather than simply being forced to preemptively choose _true or _false.

In interpret mode and interpret by steps mode, equivalence expressions will always be evaluated as _true or _false. In optimize mode, equivalence expressions containing variables will not be optimized.

Examples:

5 == 5 = OK ⇒ interprets to _true
2+3 == 5 = OK ⇒ interprets to _true
2+3 == 4+1 = OK ⇒ interprets to _true
1+4+2 == 6+1 = OK ⇒ interprets to _true
5 == 6 = OK ⇒ interprets to _false
1+1+1+1 == 5 = OK ⇒ interprets to _false
x == 6 = OK ⇒ interprets to _false
    This could be expanded by: _let x = 6 _in x == 6 ⇒ interprets to _true
    OR: _let x = 4 _in x == 6 ⇒ interprets to _false
_false == _true = OK ⇒ interprets to _false
5 == _true = OK ⇒ interprets to _false

Much like substitution expressions, there are many viable options to use for equivalence expressions but not all of them are meaningful. Equivalence statements are generally more useful when being substituted into or when being used in conditional expressions (see below).

Conditional Expressions

Conditional expressions are more commonly recognized among other programming languages as "if/else" statements. In MSDScript, they follow a similar pattern:

_if ***Expression that produces a Boolean*** _then ***expression**** _else ***expression***

The syntax is the keyword **_if**, an expression that produces a Boolean (i.e. a Boolean Expression, an Equivalence Expression, a Function Expression that returns a Boolean, etc.), the keyword **_then**, an expression, the keyword **_else**, and a final expression. When interpreted, if the Boolean or equivalence expression is interpreted as true, the expression following **_then** will be returned. If the Boolean or equivalence expression is interpreted as false, the expression following **_else** will be returned.

Examples:

_if _true _then 5 _else 6 = OK ⇒ interprets to 5
_if _false _then 5 _else 6 = OK ⇒ interprets to 6
_if x == 4 _then 5 _else 6 = OK ⇒ interprets to 5
Substitute via: _let x = 5 _in _if x == 4 _then 7 _else 9 = OK ⇒ interprets to 9
OR. _let x = 6 _in _if x == 6 _then 15 _else 0 = OK ⇒ interprets to 15
_if 1 _then 2 _else 3 = NOT OK ⇒ won't interpret because 1 is neither true nor false

Function Expressions

Function expressions are what you remember from middle school algebra. When you see f(x) = x + 10, you know that you're seeing a function dependent on x. Thus if you see the same function later but with f(5), you know you're supposed to plug 5 in for x. The difference is that here we're using the keyword **_fun** instead of f:

_fun (***dependent variable***) ***expression***

The syntax is the keyword **_fun**, an open parenthesis, a valid variable name (as described under Variable Expressions), and another expression. Returning to our example of f(x) = x+10, we would write that function like this in MSDScript syntax:

_fun (x) x+10

On their own, function expressions aren't particularly exciting, and really don't do much. In order to use them more effectively, they need to be combined with substitution expressions and call expressions.

Call Expressions

Call expression are what are used to "call" a function expression. Recalling our example from the Function Expressions of f(x) = x + 10, that function would be "called" when we see f(5). Calling a function means inserting the value in between the parenthesis for the dependent variable in the function. The syntax is really simple

Function call(***Numerical value to substitute into the function*)**

Function calls can by any valid alphabetic sequence. Using our example of f(x) = x + 10 and then f(5), our function call would be f and the value to substitute in would be 5. To match a function to the corresponding function call, we can combine some of our MSDScript syntax:

_let f = _fun (x) x + 10 _in f (5) ⇒ interprets to 15

The substitution expression is defining the function f and mapping that function to the function call. The call expression, f(5), then knows to call the function defined as f and to substitute in 5 for the dependent variable, in this case x . This is one example of how the individual pieces of MSDScript can be combined together to do more powerful things.

Examples:

One dependent variable:
_let function = _fun(y) y * 7 _in function(7) = OK ⇒ interprets to 49
Two dependent variables:
_let y = 8 _in _let xfun = _fun(x) x*y _in xfun(2) = OK ⇒ interprets to 16
_let doubleFun = _fun(x) _fun(y) x*y _in doubleFun(2)(8) = OK ⇒ interprets to 16

There are many different ways to combine substitutions, functions, and calls in order to do various computations. The last two examples show ways that you can do the same calculation with different syntax. Although on the surface MSDScript may seem limited, the possibilities are quite astounding when properly combined.

More Complex Combined Expressions

As you can see by reading the sections on Function Expressions and Call Expressions, there are many different ways to combine the syntax of MSDScript to get the desired outcome.

Fibonacci Series Example

A Fibonacci Series is a mathematical series where the next number is the sum of the previous two. The first two numbers are 0 and 1, and then it goes from there. Therefore, 0 + 1 = 1, then 1 + 1 = 2, then 1 + 2 = 3, then 2 + 3 = 5, and so on. The series would look like:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, etc.

In order to put the Fibonacci Series in via MSDScript, the input would be the following, followed by Ctrl+D:

```
	_let fib = _fun (fib) _fun (x) _if x == 0 _then 1 _else _if x == 2 + -1 _then 1 _else fib(fib)(x
+ -1) + fib(fib)(x + -2) _in fib (fib) (10)
```

In this case you can see that we're building something similar to the last example in Call Expressions. It's a function called fib that has two dependent variables, fib and x. You can see that at the end we're calling to find the 10th Fibonacci number, though you can find any Fibonacci number you want by replacing the 10 with another number. Look through the example and see if you can figure out how it works. Then see if you can build upon it.

MSDScript is powerful and flexible. Try all the combinations you can! Fibonacci is just one example of how the tools can be leveraged to accomplish cool things. Have fun with it!

API Documentation

Classes, Subclasses and their Methods

The MSDScript API is built out quite a bit to allow for a number of different classes, which allow it to have the functionality and flexibility that it does. The following will be a breakdown of each class, a list of methods available to that class, as well as a list of subclasses and their member variables. Each class method is written in JavaDoc format. Beyond each method, each subclass contains their own constructors and member variables.

<u>The Expr Class</u> (see expr.h & expr.cpp)

The Expr class is the base class for all expressions in MSDScript. Any time you see "expression" listed in a syntactical description; it's always referring to a member of the Expr class. This happens because MSDScript was designed to parse out the expression tree for a mathematical statement. Thus, everything is based upon the Expr class.

Expr contains the following methods:

equals

equals compares two Expr objects to see if the values contained in "this" and the input Expr are equal to one another.

param e - The *Expr to compare with "this"

return - Boolean true or false depending if the input Expr (e) is equal to "this" or not

interp

interp attempts to interpret the "this" Expr using the given environment and return a single value from an Expr.

param env - The environment that contains the relevant information for substituting variables

return a single *Val containing the interpreted form of the Expr

substitute

substitute attempts to substitute a value in for the given variable name if it is present in "this" Expr, assuming the variable is not bound inside a function expression. For example, in a function _fun (x), a substitution cannot be made for x.

param varName - String of the variable name to be substituted in

param subsVal - A *Val containing the value to be substituted in

return 'this' *Expr with subsVal substituted in where the variables are equal to varName

containsVariable

containsVariable looks through "this" and determines whether a variable (in the form of a VarExpr) is present within "this" Expr.

return Boolean true or false depending if a *VarExpr is present within the "this" or not *optimize*

optimize attempts to return a simpler form of an Expr by condensing the Expr where possible. Expressions that contain free variables cannot be optimized unless they are substituted.

return a (possibly) simpler “this” *Expr.

printToString

printToString returns a string representation of "this" Expr.

return a std::string representation of "this" *Expr.

The following are examples of how the various Expr classes are converted to strings:

NumExpr:	The number (ex: 4)
VarExpr:	The variable name (ex: x)
BoolExpr:	The Boolean (ex: true **OR** false)
AddExpr:	***Expr*** + ***Expr*** (ex: 5 + 4)
MultExpr:	***Expr*** * ***Expr*** (ex: 5 * 4)
LetExpr:	_let ***Expr*** = ***Expr*** _in ***Expr*** (i.e. _let x = 5 _in x + 10)
IfExpr:	_if ***Expr*** _then ***Expr*** _else ***Expr*** (i.e. _if true _then 5 _else 6)
EqualsExpr:	***Expr*** == ***Expr*** (ex: 5 == 6)
FunExpr:	_fun f(***variable***) ***Expr*** (ex: _fun f(x) x+10)
CallExpr:	***Function Call***(***Value***) (ex: f(5)

step_interp

step_interp works the same as interp() in that it attempts to condense the Expr down to a single value. step_interp returns void because step_interp() operates on a global Step variable in order to not overwhelm the stack. This allows larger computations to be performed without causing a segmentation fault

Subclasses (for better syntax examples, see **Expected Syntax – What MSDScript can do**)

<u>NumExpr</u> – Number Expressions

Member Variables:

int val

Constructor:

NumExpr(int val)

Syntax Example:

2

<u>AddExpr</u> – Addition Expressions

Member Variables:

*Expr lhs

Expr rhs

Constructor:

*AddExpr(*Expr lhs, *Expr rhs)

Syntax Example:

2 + 3

2 = lhs

3 = rhs

MultExpr – Multiplication Expressions
Member Variables:
Expr lhs
Expr rhs
Constructor:
MultExpr(*Expr lhs, *Expr rhs)
Syntax Example:
3*4
3 = lhs
4 = rhs

VarExpr – Variable Expressions
Member Variables:
std::string var
Constructor:
VarExpr(std::string var)
Example:
x

LetExpr – Substitution Expressions
Member Variables:
*VarExpr name
*Expr rhs
*Expr body
Constructor:
LetExpr(*VarExpr name, *Expr rhs, *Expr body)
Syntax Example:
_let x = 5 _in x + 15
x = name
5 = rhs
x + 15 = body

BoolExpr – Boolean Expressions
Member Variables:
bool rep
Constructor:
BoolExpr(bool rep)
Syntax Example:
_true
_false

IfExpr – Conditional Expressions
Member Variables:
*Expr testExpr
*Expr thenExpr
*Expr elseExpr
Constructor:
IfExpr(*Expr testExpr, *Expr thenExpr, *Expr elseExpr)
Syntax Example:
_if _true _then 5 _else 6
_true = testExpr
5 = thenExpr
6 = elseExpr

EqualsExpr – Equivalence Expressions
Member Variables:
*Expr lhs
*Expr rhs
Constructor:
EqualsExpr(*Expr lhs, *Expr rhs)
Syntax Example:
5 == 6
5 = lhs
6 = rhs

FunExpr – Function Expressions
Member Variables:
std::string formal_arg
*Expr body
Constructor
FunExpr(std::string formal_arg, *Expr body)
Syntax Example:
_fun (x) x+10
x = formal_arg
x+10 = body

<u>CallExpr</u> – Call Expressions
    Member Variables:
        *Expr to_be_called
        *Expr actual_arg
    Constructor
        CallExpr(*Expr to_be_called,
*Expr actual_arg)

Syntax Example:
    f(10)
    f = to_be_called
    10 = actual_arg

<u>The Val Class</u> (see value.h & value.cpp)

The Val class is the base class for interpretation of the input mathematical expression. When the parser builds out the expression tree, it is using the Expr class as a base. When we want the actual solution of a mathematical expression, we call interp() which traverses the tree and attempts to consolidate the expression tree down to a single value. To do that, interp condenses the Exprs down into Vals which is a much simpler class that is built to actually solve the problem, rather than simply represent it.

Val class contains the following methods:

equals
    equals compares to Val objects to tell if they're equal or not
    param val - The *Val to compare with "this"
    return Boolean true or false based on whether the two Val objects are equal or not

add_to
    add_to attempts to add two NumVal objects together to create a single NumVal object when possible
    param other_val - The *Val to add to "this" Val
    return a *NumVal with a value equal to the sum of "this" and other_val

mult_with
    mult_with attempts to multiply two NumVal object together to create a single NumVal object when possible
    param other_val - The *Val to multiply with "this" Val
    return a *NumVal with a value equal to the product of "this" and other_val

to_expr
    to_expr converts Vals to Exprs
    return an *Expr that represents "this" Val

printToString
    printToString returns a string representation of "this" Val object
    return a std::string representation of "this”
    The following are examples of how Val objects would be converted to strings:

NumVal:	the number (i.e. 7)
BoolVal:	the Boolean (i.e. false)
FunVal:	_fun f(***variable***) ***Expr*** (ex: _fun f(x) x+10)

is_true

is_true returns whether "this" evaluates to true or not (doesn't work if not a BoolVal)
return Boolean true or false based on the BoolVal rep

call

call is the method that actually allows us to call functions. It returns what the value of the function would be when called with the input value
param actual_arg - the input value to the function
return a *Val object that contains the value of the function with the input of the actual_arg

call_step

call_step is what allows us to call functions when in interpret by steps mode
param actual_arg_val - the value of the actual argument to be called by the function
param rest - the Cont that determines the continuation through the rest of the solution process
return - there is no return because answers are managed via the global Step variable

Subclasses

NumVal – Number Values
Member Variables:
int rep
Constructor:
NumVal(int rep)

BoolVal – Boolean Values
Member Variables:
bool rep
Constructor:
BoolVal(bool rep)

FunVal – Function Values
Member Variables:
std::string formal_arg
*Expr body
*Env env
Constructor:
FunVal(std::string formal_arg, *Expr body, *Env env)

The Env Class (see env.h & env.cpp)

The Env class was built to speed up the performance of MSDScript. Instead of requiring an expression tree to be completely traversed every time a substitution is to be made, environments (Env classes) store the values of the variable name and value in them so they're immediately available for lookup upon a call to interp. The Env class is super simple, but super powerful and increases the speed of running MSDScript dramatically.

The method available to the Env class is:

lookup

lookup find the variable name inside the Env and then returns the value to be associated with that variable

param find_name - the name of the variable to find

return a Val object with the value associated with that variable

Subclasses

EmptyEnv – Empty Environment

Member Variables:

None

Constructor:

EmptyEnv()

*as you can imagine, EmptyEnv is a special type of Env. Not all calls to interp require a lookup in an environment in order to work. Thus we always begin with an EmptyEnv and expand when necessary for substitution

ExtendedEnv – Extended Environment

Member Variables:

std::string name

*Val val

*Env rest

Constructor:

ExtendedEnv(std::string name, *Val val, *Env rest)

ExtendedEnv objects are made for when there are one (or more) variables to be substituted within a given expression. A call to *lookup* traverses the ExtendedEnv until it finds the appropriate variable to substitute.

The Step Class (see step.h & step.cpp)

The step class is a special class made specifically for memory management. For particularly large programs, MSDScript would have a tendency to crash programs because they would run out of stack space. The Step class allows all of this to be fixed by acting as a class holding global variables that change with each step of the interpretation. As such, Step has no subclasses. It also only contains a single method.

The method contained in Step is:

interp_by_steps

interp_by_steps returns the same value as a normal call to interp() would, except without overwhelming the stack space

param e - The parsed expression tree to be interpreted

return a Val object containing the interpreted value of the expression tree

The member variables of Step are:

mode_t mode

*Expr expr

*Env env
*Val val
*Cont cont

mode is the current mode that Step is operating in. These are defined in Step as being either **interp_mode** or **continue_mode**. Interp_mode means that Step needs to stop and interpret, while continue_mode tells Step to continue on to the next step of traversing the expression tree.

expr is the current expression on which Step is operating
env is the current environment on which Step is operating
val is the current Val object on which Step is operating
cont is the current Cont object (continuation) driving Step forward. It indicates how to move forward with a particular step depending on the given Expr, Env, and Val by resetting the global member variables

The Cont Class (see cont.h & cont.cpp)

The Cont class goes hand in hand with the Step class because the Cont class is what makes the Step class function. Cont class allows for Step member variables to change depending on how they need to operate for that particular Step of the interpretation of the expression tree.

The Cont class has a single method:

step_continue
step_continue resets each member variable of the global Step variable in order to let Step know how to proceed with the next step of interpreting the expression tree
return it is a void return, but it does make global changes behind the scenes

The Cont class has a single global member variable:
*Cont done
This only ever comes into play at the very end of interpretation when Cont wants to indicate to Step that it has completed the interpretation

Subclasses

DoneCont – Done with the Continuation
Member Variables:
None
Constructor:
DoneCont()
*DoneCont is a special form of Cont that indicates to Step that the interpretation is complete

AddCont – Left-hand side of Add
Member Variables:
*Val lhs_val
*Cont rest
Constructor:
AddCont(*Val lhs_val, *Cont rest)

RightThenAddCont – Right-hand side of Add
Member Variables:
*Expr rhs
*Env env
*Cont rest
Constructor:
RightThenAddCont(*Expr rhs, *Env env, *Cont rest)

MultCont – Left-hand side of Mult
Member Variables:
*Val lhs_val
*Cont rest
Constructor:
MultCont(*Val lhs_val, *Cont rest)

RightThenMultCont – Right-hand side of Mult
Member Variables:
*Expr rhs
*Env env
*Cont rest
Constructor:
RightThenMultCont(*Expr rhs, *Env env, *Cont rest)

CompCont – Left-hand side of Comp
Member Variables:
*Val lhs_val
*Cont rest
Constructor:
CompCont(*Val lhs_val, *Cont rest)

RightThenCompCont – Right-hand side of Comp
Member Variables:
*Expr rhs
*Env env
*Cont rest
Constructor:
RightThenCompCont(*Expr rhs, *Env env, *Cont rest)

CallCont – Call Expression
Member Variables:
*Val to_be_called_val
*Cont rest
Constructor:
CallCont(*Val to_be_called_val, *Cont rest)

ArgThenCallCont – Argument of Call Expression
Member Variables:
*Expr actual_arg
*Env env
*Cont rest
Constructor
ArgThenCallCont(*Expr actual_arg, *Env env, *Cont rest)

IfBranchCont – Conditional Expressions
Member Variables:
*Expr then_part
*Expr else_part
*Env env
*Cont rest
Constructor
IfBranchCont(*Expr then_part, *Expr else_part, *Env env, *Cont rest)

LetBodyCont – Substitution Expressions

Member Variables:

std::string var
*Expr body
*Env env
*Cont rest

Constructor

LetBodyCont(std::string var, *Expr body, *Env env, *Cont rest)

Parsing Functions (parse.cpp)

When MSDScript began, it was simply used to parse and interpret simple mathematical expressions. Since then it has grown far beyond that into a flexible and simple scripting language. However, at the core of MSDScript is the ability to parse apart mathematical expressions in order to build an expression tree to be evaluated.

The parsing functions are as follows:

parseInStream

parseInStream is the function for parsing the instream expression provided.
param in – instream of a complete (and syntactically correct) mathematical expression
return – an *Expr object that contains the expression tree of the instream expression

parseString

parseString runs parseInStream after converting the input string to an instream
param s - a string containing a complete mathematical expression
return – an *Expr object that contains the expression tree for the input string