LIBRARY DESCRIPTION

MSDScript is an interpreter library built with C++. It is designed to operate a wide variety of functionalities through specific keywords in command line.

This library can perform the following:

- Addition
- Multiplication
- Functions
- Expression Comparison
- Print/Pretty-Print

The following modes are available:

- `--help` : lists available mode for library
- `--test` : runs a series of test to verify that built-in functions are working as expected
- `--interp` : interprets expression string for result
- `--step` : similar to interp mode but uses explicit continuation instead of relying on C++ stack
- `--print` : prints out a condensed expression string
- `--pretty-print` : prints out the expression string that is more legible with proper spacing and usages of parentheses

GETTING STARTED

To build and create the executable **msdscript** file, ensure that the directory in terminal contains the Makefile and the source files. The files will most likely be in the Downloads folder if you've downloaded it from GitHub. In terminal, type the command `make` . The file created is named `msdscript` .

To build a library file for MSDScript, run the following commands in terminal:

```
ar -ruv libmsdscript.a cont.o env.o expr.o parse.o step.o val.o cmdline.o test.o
```

The library file is named `libmsdscript.a` .

Note: You will need to run make first to generate .o files

USING MSDSCRIPT EXECUTABLE

To run the executable file in terminal, enter the command:

`./msdscript --mode` (the mode is specify with two hyphens and the keyword)

For example, interp mode can be accessed with the following command:
`./msdscript --interp`

Note: When executing command lines, it is necessary to hit `Ctrl+D` instead of return to begin program execution.

USING MSDSCRIPT LIBRARY FILE

The library file allows for MSDScript to be linked to different programs for specific needs. MSDScript has a built-in parsing functionality that takes string inputs and parses it to expressions that can be interpreted for results.

To use MSDScript with your project, you will have add these includes to the top of your project:

```
#include "parse.h"
#include "expr.h"
#include "val.h"
#include "step.h"
#include <sstream>
```

Parse function takes in an istream so you will have to turn your expression string into an istringstream by using the following code:

`std::istringstream iss(std::string string);` , where `std::string string` is your expression string.

Then, you can add the appropriate line below to perform specific MSDScript functionality. These functions will `return` strings that
can be outputted to stream using `std::cout` .

- To interpret your expression: `Step::interp_by_steps(parse(iss))->to_string();`

- To print your expression: `parse(iss)->to_string();`

- To pretty-print your expression: `parse(iss)->to_pretty_string();`

You can also replace the istringstream iss with `std::cin` if you want to take input from the command line.

Example

MSDScript library can be used to assign the day of the week to host a meeting. To link the library with the `which_day` program, the following command is used:

```
c++ -o which_day which_day.cpp libmsdscript.a
```

- `which_day` is the name of the executable file that we have chosen
- `which_day.cpp` is the C++ file for the program
- `libmsdscript.a` is the MSDScript library that we created beforehand

In this example, if we run the command:

```
./which_day 13
```

The result will be a 4. This means that on the 14th week (week starts from 0) of the year, the day 4 of that week is when the meeting will occur. Sunday is day 0 so day 4 will be a Thursday.

BUILT-IN KEYWORDS

When writing expression strings, the following keywords are available:

- `+` means Addition
- `*` means Multiplication
- `==` means Equality
- `_let` binds a value expression
  to a variable name in which the variable name can be used in the
  body of the let expression
- `_in` assigned the body of the let
  expression
- `_if` allows for conditional representation of two
  different expressions.
- `_then` assigns an Expr that will run if the if
  condition is met
- `_else` assigns and Expr that will run if the
  condition isn't met
- `_true` means "true" boolean
- `_false` means "false"
  boolean
- `_fun` allows for function expressions with variable name.
  Function calls will replace variable with designated variable
  expression.

MSDSCRIPT LANGUAGE

Notes Regarding usages of Whitespace and Parentheses:
Parser function has a `skip_whitespace` helper function that removes any excess or unneccessary spaces. MSDScript can handle any number of whitespaces between the languages, keywords, and grammars. The `parse_inner` function handles usages of parentheses around any expressions to set precedence or to clearly separate
expressions. Any open parenthese will have to have a closing parenthese associated to it otherwise MSDScript will throw an error. Excess parentheses will be resolved automatically. Pay attention to each languages below as some will have specific usages of whitespace and parentheses.

Numbers

Format: `⟨int⟩`

Number inputs should be integers with no decimals allowed. An optional minus sign in front of the digits can be used to signify negative numbers.
Be sure to have no spacing between the minus sign and the digits.

Examples:

- `24`
- `-81`

Variables

Format: `⟨string⟩`

Variables can be a single alphabet character or a continuous (no space) string of characters. Variables can either be lowercase, uppercase, or both.

Examples:

- `x` or `X`
- `myVar`

Booleans

Format: `⟨string⟩`

True or false booleans are recognized as `_true` or `_false` .

Addition

Format: `⟨Expr⟩ + ⟨Expr⟩`

Addition is designated by the `+` symbol that adds any two Exprs. No spacing needed between `+`.

Examples:

- `1+1` or `1 + 1` will be interpreted to a value of `2`
- `x+1` or `x + 1` is acceptable as well but will not interp because varible is not assigned and handled with.
- `3+(3+1)` or `3 + (3+1)` will interp to `7`. Multiple operations can be chained together and the usages of

  parentheses in this causes tells MSDScript to interp `3+1` first.
- `(3 + 3) + 1` is the same as `3 + 3 + 1` because the parentheses are not needed due to right-associative.

Multiplication

Format: `〈Expr〉 * 〈Expr〉`

Multiplication is designed by the `*` symbol that multiples any two Exprs. No spacing needed between `*`.

Examples:

- `2*3` or `2 * 3` will interp to a value of `6`
- `2 * x` will cause an `interp error` because the variable is not assigned.
- `2 + 4 * 2` will interp to `10` because multiplication has higher precedence than addition so interp will first do `4 * 2` and then add `2` to that `8`.
- `(2 + 4) * 2` will interp to `12` because parentheses takes higher precedence so the expression will be similar to `6 * 2`.

Equality

Format: `〈Expr〉 == 〈Expr〉`

Comparison for equality is designed by `==` between any two Exprs. No spacing is needed between `==`

Examples:

- `2 == 2` or `_true==_true` will interp to `true`
- `2==3` or `_false == 3` will interp to `false`
- `2 + (4 * 3) == 2 + 4 * 3` is also acceptable because we can compare if the interp result of the rhs and lhs Exprs
  will be the same value. In this case the comparison will be `true`

Let

Format: `_let ⟨String⟩ = ⟨Expr⟩ _in  ⟨Expr⟩`

Variable assignment with `Let` will require a specific format with spacing. The expression uses two keywords: `_let` and `_in`. The format starts with us using `_let` to declare a variable. There has to be a space between they keyword `_let` and the `variable`. After the `variable` you will need to use `=` to assign an `Expr` to the `variable`. Next, the keyword `_in` has to have a space behind it that separate it from the body `Expr` that received the variable assignment.

Examples:

- `_let x=1_in x+4` or `_let x=1 _in x+4` are both acceptable but the second is preferred because the space in front of `_in` makes it look more clean. Interping this will result in the value `1` replacing the variable `x`, leaving us with `1+4` which interps to `5`.
- `5 * _let x = 4 _in x + 2` does not need usages of parentheses because the addition is chained in front of the `LetExpr`. If chained behind without parentheses, MSDScript will treat the `5` as part of the body `Expr` in `_let`, such as `_let x=4 _in x+2*5`. The front-chain will interp to `30` while the back-chain will interp to `40`. To properly chain an operation behind a `LetExpr`, use parentheses around the `LetExpr` such as `(_let x = 4 _in x + 2) * 5`.

If

Format: `_if ⟨Expr⟩ _then ⟨Expr⟩ _else ⟨Expr⟩`

If conditional branching uses three keywords: `_if` , `_then` and `_else` ; Be sure to have a space after each of these keywords. `IfExpr` string format starts with *if followed by an Expr that serves as a test condition. For IfExpr to work, the test condition Expr has to return an interp boolean value. Next, *then branches to an Expr that gets called if the test condition is true while. The *else keyword branches to another Expr if the test condition is false.

Examples:

- `_if 3==3 _then 3*3 _else 3+3` will return an interped value of `9` because the test condition Expr `3=3` is `true` so the `IfExpr` follows the _then Expr of `3*3`
- `_if _false _then 2+4 _else _let x=1 _in x+4` will branch down to the `_else` clause when interped, resulting in a value of `5`
- `_if 1+3 _then 2+4 _else 4*5` will throw an `error` because the condition `Expr` is not a boolean expression
- `(_if 1+3 == 2+2 _then 2+4 _else 4*5) + 3` requires parentheses around the `IfExpr` because we are chaining an addition at the end, similar to the chaining example for `LetExpr`

Functions

Format: `_fun ( ⟨String⟩ ) ⟨Expr⟩`

To define a function, we start with `_fun` followed by a `variable` that is enclosed in parentheses. After the closing parenthese is a body `Expr` of any sort.

Example:

- `_fun (x) x + 24` or `_fun(x)x+24` are valid string expression of functions.

Function Calls

Format: `⟨Expr⟩ ( ⟨Expr⟩ )`

To call the function, use the format: `Expr(Expr)` . The value of the leading `to_be_called` Expr will have to be a function for function calls to work.
In another enclosed parentheses is the `actual_arg Expr` , however, for the function call to work without any error, the `actual_arg Expr` will have to be an Expr that can be interped to a number value.

Example:

- `(_fun (x) x + 24) (3)` or `(_fun(x)x+24)(3)` interps to `27`
  because the `actual_arg` of `3` gets passed onto the `formal_arg` variable
  `x` . In this case, we enclosed the function Expr in parentheses to set it as the `to_be_called` Expr.
- `_let f = _fun (x) x + 24 _in f(3)` also interps to `27` because we used Let expression to set the function as `f` . The function call format `f(3)`
  takes the `actual_arg` of `3` and passes that into the `to_be_called` Expr. In this case, the
  `to_be_called` Expr, `f` , does not need parentheses.
- `(_fun (x) x * x) (_if 3==3 _then _true _else _false)` will result in a `runtime error`
  because the
  `actual_arg Expr` resulted in a `boolean` value when interpreted
- `(_fun (x) x + 24) (3) + 5` does not need extra parentheses like `IfExpr` and
  `LetExpr` when chaining with other operations because parentheses are
  already used to define the Function and its call argument.

MSDSCRIPT GRAMMAR

Parse

MSDScript is designed to parse expression strings into Expr objects that are used to perform basic calculations, variable assignment, comparisons, and function calls.

Function: `PTR(Expr) parse(std::istream &in)`

- Takes an istream input and converts it to an Expr

Expr

Expr or expressions stores input information that MSDScript uses to perform specific operations. In the Expr Class, there are many Expr sub-types.

Expr Sub-Types:

- ## NUMBERS (NumExpr)

  NumExpr are the expression representation of integer values in MSDScript.

  Constructor: `NumExpr(int rep);`

- ## ADDITION (AddExpr)

  Script representation of addition (+). The left-hand side and right-hand side can be any sub-type Expr.

  Constructor: `AddExpr(PTR(Expr) lhs, PTR(Expr) rhs);`

- ## MULTIPLICATION (MultExpr)

  Script representation of multiplication (*). The left-hand side and right-hand side can be any sub-type Expr.

  Constructor: `MultExpr(PTR(Expr) lhs, PTR(Expr) rhs);`

- ## VARIABLE (VarExpr)

  Script representation of variables. Must be at least one character long and can only consist of lowercase or uppercase alphabet letters.

  Constructor: `VarExpr(std::string str);`

- ## VARIABLE ASSIGNMENT (LetExpr)

Script representation of variable assignment. The right-hand side Expr is assigned to a variable name. If the body Expr contains the variable name, then the variable gets replaced with the right-hand side Expr.

Constructor: `LetExpr(std::string variable, PTR(Expr) rhs, PTR(Expr) body);`

- ### BOOLEAN (BoolExpr)

  Script representation of booleans. The Expr can only store true or false values as its rep.

  Constructor: `BoolExpr(bool rep);`

- ### EQUALITY (EqExpr)

  Script representation of a comparison check (==). The left-hand and right-hand side can be any sub-type Expr.

  Constructor: `EqExpr(PTR(Expr) lhs, PTR(Expr) rhs);`

- ### IF CONDITIONS (IfExpr)

  Script representation of conditional branching. The _if parameter is an Expr that serves as a test condition. If the condition is met, the _then Expr is the path that is followed, otherwise it follows the _else Expr path.

  Constructor: `IfExpr(PTR(Expr) *if, PTR(Expr) *then, PTR(Expr) _else);`

- ### FUNCTION (FunExpr)

  Script representation of a function definition. The parameter formal_arg is the argument that can be applied to the body Expr.

  Constructor: `FunExpr(std::string formal_arg, PTR(Expr) body);`

- ### FUNCTION CALLS (CallExpr)

Script representation of a function call. The to_be_called parameter is the defined function and the actual_arg is the argument used to evaluate the function.

Constructor: `CallExpr(PTR(Expr) to_be_called, PTR(Expr) actual_arg);`

MSDSCRIPT API

expr.cpp and expr.h:

Main files for expression representations.

Methods:

- `bool equals(PTR(Expr) other)` : Checks if two expressions are the same.
- `PTR(Val) interp(PTR(Env) env)` : Evaluate the expression for the result.
- `std::ostream& print(std::ostream& argument)` : Prints out the Expr to the outstream.
- `std::ostream& pretty_print(std::ostream& argument)` : Similar to print method but prints the Expr that is more legible with proper spacing and usages of parentheses
- `std::string to_string()` : Converts the Expr to a string.
- `std::string to_pretty_string()` : Converts the Expr to a string that is more legible with proper spacing and usages of parentheses.
- `void pretty_print_at(print_mode_t mode, std::ostream& argument, int newLineLocation, bool alwaysRHS)` : Helper method that determines the correct spacing and usages of parentheses.
  * `void step_interp()` : Evaluate the expression using explicit continuation (interpreting by steps)

Also see MSDSCRIPT GRAMMER for more details.

val.cpp and val.h:

Stores interpreted values as a Val Class object instead of an Expr Class object.

Interpretting FunExpr returns FunVal while interpretting BoolExp returns BoolVal. Any other Expr interpretation returns NumVal.

Methods:

- `PTR(Val) add_to(PTR(Val) other_val)` : Helper method to add values together and return the result as a Val object.
- `PTR(Val) mult_by(PTR(Val) other_val)` : Helper method to multiples values together and return the result as a Val object.
- `bool equals(PTR(Val) other)` : Check if two values are the same.
- `std::string to_string()` : Converts value of the object to a string.
- `PTR(Val) call(PTR(Val) actual_arg)` : Call the function with the actual argument.
- `void call_step(PTR(Val) actual_arg, PTR(Cont) rest)` : Explicit continuation verson of call method. Call the function with the actual argument
- `bool is_true()` : Check if the value is true or false.

env.cpp and env.h:

Environment for referencing of values; similar to
substitution functionality

Properties:

- `static PTR(Env) empty` : Represents an empty environment.

Methods:

- `PTR(Val) lookup(std::string find_name)` : Find the value of a variable.

cont.cpp and cont.h:

Allows for step mode interpretation that avoids
using C++ stack to prevent segmentation faults for large recursive
expressions

Properties:

- `static PTR(Cont) done` : Finished. No more need for continue steps.

Methods:

- `step_continue()` : Sets the mode to either interp_mode or continue_mode in Step Class.

step.cpp and step.h:

Allows for step mode interpretation. Runs step by step until PTR(Cont) done is reached.

enum:

- mode_t consists of two modes, `interp_mode` or `continue_mode`

Properties:

- `static mode_t mode` : Represents the mode from enum mode_t. The `interp_mode` indicates that start expression interpretation while
- `continue_mode` deliver values to a continuation.
- `static PTR(Expr) expr` : Contains the current expression that needs to be interpreted.
- `static PTR(Env) env` : Keep track of the environment of the current step.
- `static PTR(Val) val` : Contains the value to be delivered to the continuation.
- `static PTR(Cont) cont` : Represents the next step in continuation.

Methods:

- `static PTR(Val) interp_by_steps(PTR(Expr) e)` : Interp expression by stepping. Avoids recursive calls at C++ stack level that causes stack overflow.

parse.cpp and parse.h:

Parses strings into expressions

Functions:

- `PTR(Expr) parse(std::istream &in)` : Parses command line expressions into Expr object.
- `PTR(Expr) parse_str(std::string s)` : Parses expression string to Expr object.

Helper Functions:

- `static void consume(std::istream &in, int expect)` : If the current in-stream character is equal to the expected char, then
  fetch that character from the in-stream.

- `static void skip_whitespace(std::istream &in)` : Removes/skip unneccessary spaces.

- `PTR(Expr) parse_num(std::istream &in)` : Parses numbers of the overall in-stream.

- `PTR(Expr) parse_var(std::istream &in)` : Parses the variables of the overall in-stream.

- `PTR(Expr) parse_let(std::istream &in)` : Parses the let expression of the overall in-stream.

- `std::string parse_keyword(std::istream &in)` : Detects the usages of keywords with an underscore then parses and return the entire keyword string.

- `PTR(Expr)parse_if(std::istream &in)` : Parses the If expression of the overall in-stream.

- `PTR(Expr) parse_fun(std::istream &in)` : Parses the function expression of the overall in-stream.

- `static PTR(Expr) parse_expr(std::istream &in)` : Initiator function that starts parsing expression in-stream. This helper function is called in
  the main parse and parse_str functions. Parses comparative expressions or returns parse_comparg.

- `PTR(Expr) parse_comparg(std::istream &in)` : Parses addition expressions or returns parse_addend.

- `PTR(Expr) parse_addend(std::istream &in` : Parses multiplication expressions or returns parse_multicand.

- `PTR(Expr) parse_multicand(std::istream &in)` : Parses Function Call expressions or returns parse_inner.

- `PTR(Expr) parse_inner(std::istream &in)` : Parses numbers, new Expr, variables, let expressions, boolean expressions, if expressions, and function expressions.

Parsing Grammar displayed visually:

```
⟨expr⟩ = ⟨comparg⟩
              | ⟨comparg⟩ == ⟨expr⟩

⟨comparg⟩ = ⟨addend⟩
              | ⟨addend⟩ + ⟨comparg⟩

⟨addend⟩ = ⟨multicand⟩
              | ⟨multicand⟩ * ⟨addend⟩

⟨multicand⟩ = ⟨inner⟩
              | ⟨multicand⟩ ( ⟨expr⟩ )

⟨inner⟩ = ⟨number⟩
              | ( ⟨expr⟩ )
              | ⟨variable⟩
              | _let ⟨variable⟩ = ⟨expr⟩ _in ⟨expr⟩
              | _true
              | _false
              | _if ⟨expr⟩ _then ⟨expr⟩ _else ⟨expr⟩
              | _fun( ⟨variable⟩ ) ⟨expr⟩
```

cmdline.cpp and cmdline.h:

Allows for specific calls of the different modes available for MSDScript

Functions:

`int use_arguments(int argc, char **argv)` : Takes in command line arguments and execute designated modes.

pointer.h:

Shared pointers used to help with memory leaks

The following macros are used to make code more readable:

Macro	C++
NEW(T)	std::make_shared<T>
PTR(T)	std::shared_ptr<T>
CAST(T)	std::dynamic_pointer_cast<T>
CLASS(T)	class T : public std::enable_shared_from_this<T>
THIS	shared_from_this()

catch.h:

Testing library used to test MSDScript functionalities. Tests can be found at the end of each .cpp files. Expr tests are in test.cpp

POTENTIAL ERRORS

`Segmentation faults` may occur in interp mode if the recursive expression is too large. Use step mode instead. Multiplication and addition cannot be used on other expressions besides number expression

REPORT A BUG OR QUESTIONS/CONCERNS

Please send a direct message to htruong17 on GitHub or email at harold.truong@gmail.com