MSD Script

A Comprehensive Guide and Documentation

By: David Scovel

**Description**

What it Does

MSDscript is an expression evaluating software where the user inputs in an expression that follows a particular grammar (see below) and evaluates it depending on the flags entered in by the user.

Problem it Solves

Have you ever had an overly complicated expression (that follows the below grammar)? If so, you’re in luck. This program will either simplify that expression or evaluate it into a value.

How it’s meant to be used

This program has three modes:

* Interpreter, which evaluates an MSDscript expression (defined below) and returns the value
* Optimizer, which simplifies an MSDscript expression and returns a simplified MSDscript expression
* Step Interpreter, an improved interpreter, which evaluates an MSDscript expression and returns the evaluated value using a step method

**Getting Started**

Installation

Adhere to the following instructions to install:

1. Download and install cmake from https://cmake.org/download/
2. Navigate to the folder containing the zipped msdScipt.zip files
3. Unzip msdScript.zip
4. Right click on the build folder and select: new Terminal at folder
5. Type the command: cmake ..
6. Type the command: cmake --build .
7. Run the created executable using: ./msdScript with any appropriate flags

Library Usage

To use the msdScript library and all the functions and classes below follow these steps:

1. Move msdScript.zip into the desired project’s directory
2. Unzip msdScript.zip
3. In any file you want to access the library use the line: #include “msdScriptLib.h”

**Terms and Grammar**

Terms

*Expressions* -- include addition, multiplication, local binding, and functions. See the grammar specification for details.

*Values* -- include numbers, booleans, and functions.

*Interpretation* -- Evaluates expressions and returns a value. If the evaluation reaches a free variable, an error will be thrown. Otherwise, the result is a semantically equivalent value.

*Optimize* -- Optimization produces a semantically equivalent expression that is no larger than the input expression.

Grammar

**<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>**

**User Guide**

Modes

After creating the executable using the steps in the installation section you have created an executable called msdScript. This executable has three modes of operation: interpret, optimize, and step-interpret. The program is run using the following flags in the terminal:

*./msdScript (no flags):* Interpreter Mode

Use Interpreter Mode when you want to evaluate an inputted expression down to a value.

Ex: 2 + 2 interprets to 4

*./msdScript --opt:* Optimize Mode

Use Interpreter Mode when you want to simplify an inputted expression to a simpler expression. Useful when the expression contains free variables.

Ex: (2 + 2) + x optimizes to 4 + x

*./msdScript --step:* Step Interpreter Mode

Use Step Interpreter Mode when evaluating functions that would overflow the stack in interpret mode.

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

Valid Inputs

Once you’ve run the program (with any appropriate flags), the program will wait for user input. This section will describe what types of inputs are valid.

* In order to be valid, the user must input an <expr> that **follows the grammar** defined above.
* Input does allow for whitespace (spaces or new lines) in between expressions
* Numbers
  + Must be an uninterrupted stream of integers.
    - No commas, decimals, spaces, or symbols
  + Negative integers are supported with a
  + Floating numbers are not supported
  + Max integer value is 2147483647
* Variables
  + Must be made up of ascii alphabetic characters
    - Upper and Lowercase are both valid
  + Cannot have spaces
* Booleans
  + Cannot perform arithmetic on booleans

Possible Errors

*Error: invalid command*

*Valid commands: --opt --step*

Use of unsupported flags

*Invalid Input*

Input didn’t follow the grammar specifications

*Error: requires ==*

Inputted comparg used a single equal sign instead of the valid double equal signs

*Expected a close parenthesis*

Inputted an open parenthesis without a matching closed parenthesis

*Expected a digit, character, or open parenthesis*

Encountered an input that doesn’t follow the grammar specifications in an <inner>

*improper \_let format: No =*

Missing an equal sign in a \_let expression

*improper \_let format: In*

Missing the \_in portion of a \_let expression

*improper \_let format: \_*

Missing an \_ in a \_let expression

*invalid if statement: missing \_*

Missing an \_ in an \_if expression

*invalid if statement: missing then*

Missing or errors with the then portion 0f an \_if expression

*invalid if statement: missing else*

Missing or errors with the else portion of an \_if expression

*invalid fun statement: missing (*

Missing the open parenthesis from a \_fun expression

*invalid fun statement: missing )*

Missing the closed parenthesis from a \_fun expression

*Cannot compute statements containing variables.*

Entered input is trying to interpret an expression with floating(undefined) variables

*not a number*

Cannot add or multiply non-number expressions

*Error: cannot call a number*

Invalid <multicand> input

*Error: cannot call a bool*

Invalid <multicand> input

*Numbers are neither true nor false.*

The <expr> following \_if in an \_if statement could not evaluate to a number instead of a boolean

*no adding booleans*

Cannot add booleans

*no multiplying booleans*

Cannot multiply booleans

*no adding functions*

Cannot add unevaluated functions

*no multiplying functions*

Cannot multiply unevaluated functions

*Functions are neither true nor false.*

The <expr> following \_if in an \_if statement could not evaluate to a function instead of a boolean

Input to Output Examples

*Numbers*

|  |  |  |
| --- | --- | --- |
| Evaluation Mode | Input | Output |
| --interp or --step | 2+2 or 2 + 2 | 4 |
| 2\*4 or 2 \* 4 | 8 |
| 2 + 4 \* 3 | 14 |
| --opt | 2+2 or 2 + 2 | 4 |
| 2\*4 or 2 \* 4 | 8 |
| 2 + 4 \* 3 | 14 |

*Variables*

|  |  |  |
| --- | --- | --- |
| Evaluation Mode | Input | Output |
| --interp or --step | 2 + x | Cannot compute statements containing variables. |
| 3 \* x |
| x + (2 \*3) |
| --opt | 2 + x | 2 + x |
| 3 \* x | 3 \* x |
| x + (2 \* 3) | x + 6 |

*Booleans*

|  |  |  |
| --- | --- | --- |
| Evaluation Mode | Input | Output |
| --interp or --step | \_true | \_true |
| \_false | \_false |
| --opt | \_true | \_true |
| \_false | \_false |

*Let*

|  |  |  |
| --- | --- | --- |
| Evaluation Mode | Input | Output |
| --interp or --step | \_let x = 2 \_in x + 3 | 5 |
| \_let y = 2 + 5 \_in y | 7 |
| \_let x = y \_in x + 1 | Cannot compute statements containing variables. |
| --opt | \_let x = 2 \_in x + 3 | 5 |
| \_let y = 2 + 5 \_in y | 7 |
| \_let x = y \_in x + 1 | \_let x = y \_in x + 1 |

*If*

|  |  |  |
| --- | --- | --- |
| Evaluation Mode | Input | Output |
| --interp or --step | \_if \_true \_then 5 \_else 7 | 5 |
| \_if \_false \_then 5 \_else 7 | 7 |
| \_if 3 == 3 \_then \_true \_else \_false | \_true |
| --opt | \_if \_true \_then 5 \_else 7 | 5 |
| \_if \_false \_then 5 \_else 7 | 7 |
| \_if 3 == 3 \_then \_true \_else \_false | \_true |

*Comparisons*

|  |  |  |
| --- | --- | --- |
| Evaluation Mode | Input | Output |
| --interp or --step | 2 + 2 == 3 + 1 | \_true |
| 5 \* 7 == 3 | \_false |
| 4 \* 2 == x | Cannot compute statements containing variables. |
| --opt | 2 + 2 == 3 + 1 | \_true |
| 5 \* 7 == 3 | \_false |
| 4 \* 2 == x | 8 == x |

*Functions/Calls*

|  |  |  |
| --- | --- | --- |
| Evaluation Mode | Input | Output |
| --interp or --step | \_fun (x) x + 2 | [function] |
| (\_fun (x) x + 5)(4) | 9 |
| \_let x = 7 \_in (\_fun (x) x + 3)(x) | 10 |
| --opt | \_fun (x) x + 2 | [function] |
| (\_fun (x) x + 5)(4) | 9 |
| \_let x = 7 \_in (\_fun (x) x + 3)(x) | 10 |

*Complex Example*

|  |  |  |
| --- | --- | --- |
| Evaluation Mode | Input | Output |
| --interp | \_let countdown = \_fun(countdown) \_fun(n) \_if n == 0 \_then 0 \_else countdown(countdown)(n + -1) \_in countdown(countdown)(100000) | Stack Overflow |
| --step | \_let countdown = \_fun(countdown) \_fun(n) \_if n == 0 \_then 0 \_else countdown(countdown)(n + -1) \_in countdown(countdown)(100000) | 0 |
| --opt | \_let countdown = \_fun(countdown) \_fun(n) \_if n == 0 \_then 0 \_else countdown(countdown)(n + -1) \_in countdown(countdown)(100000) | Stack Overflow |

**API Documentation**

**Macros**

This program uses the following macros for memory allocation and deallocation purposes.

# define NEW(T) std::make\_shared<T>  
# define PTR(T) std::shared\_ptr<T>

**class Expr**

Base Expr class made up of virtual methods that are implemented by its inheriting classes. An Expr cannot be instantiated.

*Constructor:*

None

*Methods:*

virtual bool equals(PTR(Expr)e)

-Checks for equality between two expressions

virtual PTR(Val) interp(PTR(Env) env)

-Evaluates a given expression into a value using an Env

virtual PTR(Expr)substitute(std::string var, PTR(Val)newVal)

-Returns an Expr where any instance of the Var class that is equivalent with the parameter var is replaced with the newVal

virtual bool containsVar()

-Returns whether or not the Expr contains a Var

virtual PTR(Expr)optimize()

-Returns the simplest form of the expression

virtual std::string stringify()

-Translates the expression into a String (used for printing)

virtual void step\_interp()

-Evaluates a given expression into a value using an Env, but solves overflow issues by not allocating as much memory (only used in the interp\_by\_steps method)

**class Number**

An Expression that contains a single integer value. Inherits from Expr.

*Member Variables:*

Int val

*Constructor:*

Number(int val)

*Methods:*

\*\*implements methods from Expr class\*\*

**class Add**

An Expression used for performing addition. Inherits from Expr.

*Member Variables:*

PTR(Expr)lhs  
PTR(Expr)rhs

*Constructor:*

Add(PTR(Expr)lhs, PTR(Expr)rhs)

*Methods:*

\*\*implements methods from Expr class\*\*

**class Mult**

An Expression used for performing multiplication. Inherits from Expr.

*Member Variables:*

PTR(Expr)lhs  
PTR(Expr)rhs

*Constructor:*

Mult(PTR(Expr)lhs, PTR(Expr)rhs)

*Methods:*

\*\*implements methods from Expr class\*\*

**class Var**

An Expression that contains string variable. Inherits from Expr.

*Member Variables:*

std::string val

*Constructor:*

*Var(std::string val)*

*Methods:*

\*\*implements methods from Expr class\*\*

**class Let**

An expression used for assigning variables in an Expr. Inherits from Expr.

*Member Variables:*

std::string toReplace;  
PTR(Expr)variableSub;  
PTR(Expr)body;

*Constructor:*

Let(std::string toRep, PTR(Expr)replaceW, PTR(Expr)replaceI)

*Methods:*

\*\*implements methods from Expr class\*\*

**class BoolExpr**

An expression that contains a bool. Inherits from Expr.

*Member Variables:*

bool rep

*Constructor:*

BoolExpr(bool rep)

*Methods:*

\*\*implements methods from Expr class\*\*

**class Comparison**

An expression used for performing comparisons. Inherits from Expr.

*Member Variables:*

PTR(Expr)lhs  
PTR(Expr)rhs

*Constructor:*

Comparison(PTR(Expr)lhs, PTR(Expr)rhs)

*Methods:*

\*\*implements methods from Expr class\*\*

**class If**

An expression used for evaluating conditions and entering various branches of execution. Inherits from Expr.

*Member Variables:*

PTR(Expr)condition  
PTR(Expr)thenExpr  
PTR(Expr)elseExpr

*Constructor:*

If(PTR(Expr)cond, PTR(Expr)thenE, PTR(Expr)elseE)

*Methods:*

\*\*implements methods from Expr class\*\*

**class Fun**

An expression used for representing functions. Inherits from Expr.

*Member Variables:*

std::string variable  
PTR(Expr) body

*Constructor:*

Fun(std::string var, PTR(Expr)bod)

*Methods:*

\*\*implements methods from Expr class\*\*

**class Call**

An expression used for carrying out functions through substitution. Inherits from Expr.

*Member Variables:*

PTR(Expr)toBeCalled;  
PTR(Expr)actualArg;

*Constructor:*

Call(PTR(Expr)lhs, PTR(Expr)rhs)

*Methods:*

\*\*implements methods from Expr class\*\*

**class Val**

Base Val class made up of virtual methods that are implemented by its inheriting classes. A Val cannot be instantiated.

*Member Variables:*

None

*Constructor:*

None

*Methods:*

virtual bool equals(PTR(Val)val)

-Checks for equality between two Vals

virtual PTR(Val)add\_to(PTR(Val)other\_val)

- Adds two NumVals together (returns errors for BoolVal and FunVal)

virtual PTR(Val)mult\_with(PTR(Val)other\_val)

-Multiplies two NumVals together (returns errors for BoolVal and FunVal)

virtual PTR(Expr)to\_expr()

-Returns the Expr form of any given Val

NumVal -> Number, BoolVal -> BoolExpr, FunVal -> Fun

virtual std::string stringify()

-Returns the string form of any given Val (used for printing)

virtual PTR(Val)call(PTR(Val) actual\_arg)

-Carries out a function through substitution

virtual bool is\_true()

-Returns whether a boolVal is true or false (returns errors for NumVal and FunVal)

**class NumVal**

A Value containing a single integer. Inherits from Val.

*Member Variables:*

Int rep

*Constructor:*

NumVal(int rep)

*Methods:*

\*\*implements methods from Val class\*\*

**class BoolVal**

A Value containing a bool. Inherits from Val.

*Member Variables:*

Bool rep

*Constructor:*

BoolVal(bool rep)

*Methods:*

\*\*implements methods from Val class\*\*

**class FunVal**

A Value representation of a function

*Member Variables:*

std::string variable  
PTR(Expr) body  
PTR(Env) env

*Constructor:*

FunVal(std::string var, PTR(Expr) bod, PTR(Env) env)

*Methods:*

\*\*implements methods from Val class\*\*

**class Env**

Base Env class made up of a virtual method that is implemented by its inheriting classes. Env is used to keep a history of variables and their associated values. An Env cannot be instantiated.

*Member Variables:*

None

*Constructor:*

None

*Methods:*

virtual PTR(Val) lookup(std::string find\_name)

-Looks to see if a given variable is in the Env and substitutes it with its associated value.

**class ExtendedEnv**

An Env that extends from the base in order to store more variable values. An Env can be made up of multiple ExtendedEnvs. Inherits from Env.

*Member Variables:*

std::string name;  
PTR(Val) val;  
PTR(Env) rest;

*Constructor:*

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

*Methods:*

\*\*implements methods from Env class\*\*

**class EmptyEnv**

An Env that marks the end of an Env chain. Inherits from Env.

*Member Variables:*

None

*Constructor:*

None

*Methods:*

\*\*implements methods from Env class\*\*