MSD Script
Documentation

Description

<u>What it does</u>
MSD Script is an expression evaluating software where the user chooses an evaluation method, inputs an expression that follows the MSD Script expression format, and the expression is then evaluated accordingly.

<u>Problem it solves</u>
MSDScript is capable of rapidly solving and simplifying complex mathematical expressions.

<u>How it is meant to be used</u>
MSD Script provides three evaluation modes:
- Interpreter - evaluates an MSDscript expression (defined below) and returns the value.
- Optimizer - simplifies an MSDscript expression and returns a simplified MSDscript expression.
- Step interpreter - an improved interpreter allowing for larger computations.

Terms and Grammar

<u>Terms</u>
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.

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

<u>Grammar</u>
```
<expr>          =           <comparg>
                      |     <comparg> == <expr>

<comparg>             =     <addend>
                      |     <addend> + <comparg>

<addend>              =     <multicand>
                      |     <multicand> * <addend>

<mutlicand>           =     <inner>
                      |     <multicand> (<expr>)

<inner>               =     <number> | (<expr>) | <variable>
                      |     _let<variable> = <expr> _in <expr>
```

| _true | _false

Grammar continued…

| _if <expr> _then <expr> _else <expr>

| (_fun (<variable>) <expr>)

Getting Started

Installation - Mac OS

1. Download and install CMake from https://cmake.org/download/ .
2. Unzip the "MSDScipt.zip" file.
3. Open terminal in the unzipped directory by right clicking on the build directory and choosing "New Terminal At Folder".
4. Type "cmake .." and press enter.
5. Type "cmake --build .."

Basic Usage

1. Access the ""Parser" Unix executable inside the build folder by right clicking on the build directory and choosing "New Terminal At Folder".
2. In terminal type "./Parser" and press enter.
   i. Including no arguments after "./Parser" runs the parser in interpretation mode. To learn more about running the parser in other modes see the User Guide below.
3. Expressions can now be typed into the terminal for evaluation. For more on evaluating expression see the User Guide below.

Library Usage

1. Copy all .h and .hpp files into your project folder.
2. Copy the libMSDLib.a into your project folder.
3. Include the "MSDScriptLib.hpp" file in the project (e.g. #include MSDScriptLib.hpp).
4. Compile your project in terminal using the libMSDLib.a file (e.g. g++ -std=c++17 -o "name of executable" main.cpp libMSDLib.a)

User Guide

Modes of Evaluation

The default method of expression evaluation is interpretation mode. If no arguments are provided following the command to run MSDScript, this mode is automatically chosen. Otherwise, arguments can be included to change the mode of evaluation after the command to run the program as shown below.

- ./Parser - Providing no arguments will execute the program in interpretation mode.

- ./Parser _interp - Including the "_interp" argument will execute the program in interpretation mode.

- ./Parser _optimize - Including the "_optimize" argument will execute the program in optimization mode.

- ./Parser _step - Including the "_step" argument will execute the program in step mode.

Expressions

After choosing an evaluation mode an expression can be entered into the terminal for evaluation. Expressions follow the grammar, as illustrated in the Terms and Grammar section of the documentation. Some examples of common expressions are described below.

Integers

Expressions can include integers in conjunction with the "+" and "*" operators, as well as parenthesis. Subtraction and division are not supported, however negative integers are supported. Although not required, white space is supported in these expressions. No whitespace is allowed within an integer, nor can commas be included in an integer. The maximum integer value supported is 2147483647, and values larger than this will wrap around. Floating point values are currently unsupported.

Examples:

Evaluation Mode	Input Expression	Result
Interpretation/Step	2 + 2 or 2+2	4
	2 + -2	0
	2 * 2 or 2*2	4
	(2+2) * 2	8
Optimization	2 + 2 or 2+2	4
	2 + -2	0
	2 * 2 or 2*2	4
	(2+2) * 2	8

Variables

Expressions can contain variables. Variables are made up of characters which are case sensitive and can range from ASCII letters a-z and A-Z. No white space is allowed within variable names. Free variables are not supported in interpretation and step modes.

Evaluation Mode	Input Expression	Result
Interpretation/Step	(2 + 2) + x	Error: Free Variable x
	2 + (2 + x)	Error: Free Variable x
	(2 + 2 + x)	Error: Free Variable x
Optimization	(2 + 2) + x	4 + x
	2 + (2 + x)	2 + 2 + x
	(2 + 2 + x)	2 + 2 + x

Boolean Expressions

Boolean expressions are expressions that include “_true” or “_false” keywords. Boolean expressions are often used in conjunction with If expressions and comparison expressions (described below). Arithmetic cannot be performed on booleans. Booleans can be compared using the double equals operator (==).

Evaluation Mode	Input Expression	Result
Interpretation/Step	_if _true _then 5 _else 6	5
	_if _false _then 5 _else 6	6
	_true == _true	_true
Optimization	_if _true _then 5 _else 6	5
	_if _false _then 5 _else 6	6
	_true == _true	_true

Comparison Expressions

Comparison expressions allow the user to evaluate the equality of expressions and values. Comparison expressions return a _true or _false keyword.

Evaluation Mode	Input Expression	Result
Interpretation/Step	5 == 5	_true
	5 == 6	_false
	5 + 6 == 5 + 6	_true
Optimization	5 == 5	_true
	5 == 6	_false
	5 + 6 == 5 + 6	_true

If Expressions

If expressions allow the user to evaluate the condition and make decisions based on the condition. If expressions follow the format of _if <expression> _then <expression> _else <expression>. If the _if condition evaluates to _true the _then expression is evaluated. Otherwise, if the _if condition evaluates to _false the _else condition is evaluated.

Evaluation Mode	Input Expression	Result
Interpretation/Step	_if _true _then 5 * 5 _else 6 * 6	25
	_if _false _then 5 * 5 _else 6 * 6	36
	_if (5*5) == (6*6) _then _true _else _false	_false

Evaluation Mode	Input Expression	Result
Optimization	_if _true _then 5 * 5 _else 6 * 6	25
Optimization	_if _false _then 5 * 5 _else 6 * 6	36
	_if (5*5) == (6*6) _then _true _else _false	_false

Let Expressions
Let expressions allow a user to use substitution to assign a value or expression to a variable within an expression. Let expressions follow the format of _let <string> = <expression>_in <expression> . A variable bound in a let expression is visible only in the body (the expression following _in) of the let, and not in the expression following the '='.

Evaluation Mode	Input Expression	Result
Interpretation/Step	_let x = 5 _in x * 5	25
	_let x = 5 * 5 _in x * 5	125
Optimization	_let x = 5 _in y * 5	y * 5
	_let x = y * 5 _in x * 5	_let x = y * 5 _in x * 5

Function Expressions
Function expressions allow users to create expressions that represent functions. Function expressions can be used to evaluate complex expressions and make recursive calls. Function expressions follow the format _fun <string> <expression> .

Evaluation Mode	Input Expression	Result
Interpretation/Step	(_fun(x) 2 + x) (3)	5
	(_fun(y) y * (y + 3)) (2)	10
Optimization	(_fun(x) 2 + x) (3)	(_fun(x) 2 + x) (3)
	(_fun(y) y * (y + 3)) (2)	(_fun(y) y * (y + 3)) (2)

Call Expressions
Call expressions are used in conjunction with function expressions. They are used to implement function calls through substitution. Call expressions follow the format _fun <string> <expression> <call expression>. CallExpr examples are are shown in bold below. Note that the call expression is bound to the <expression> of a function expression.

Evaluation Mode	Input Expression	Result
Interpretation/Step	_if _false _then _let f = _fun(x) x + 1 _in **f(10)** _else _let f = _fun(x) x + 1 _in **f(100)**	101

Evaluation Mode	Input Expression	Result
Interpretation/Step	_if _true _then _let f = _fun(x) x + 1 _in **f(10)** _else _let f = _fun(x) x + 1 _in **f(100)**	11
Optimization	_if _false _then _let f = _fun(x) x + 1 _in **f(10)** _else _let f = _fun(x) x + 1 _in **f(100)**	_let f = _fun(x) x + 1 _in f(10)
	_if _true _then _let f = _fun(x) x + 1 _in **f(10)** _else _let f = _fun(x) x + 1 _in **f(100)**	_let f = _fun(x) x + 1 _in f(100)

Complex Expressions
Below are some example of complex instructions and their results in interpretation/step mode. Note the the count and countdown examples will run out of memory if not in step mode.

Evaluation Mode	Input Expression	Result
Interpretation/Step	_let count = _fun(count) _fun(n) _if n == 0 _then 0 _else 1 + count(count)(n+-1) _in count(count)(100000)	100000
	_let countdown = _fun(countdown) _fun(n) _if n == 0 _then 0 _else countdown(countdown)(n+-1) _in countdown(countdown)(100000)	0
	_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)"))	89

Common Errors

- Invalid command line argument
  - An invalid command was passed to the program. Please ensure that the argument is one of '_interp', '_optimize', or '_step'.
- Expected end of file
  - End of file not encountered. Check format of expression.
- Expected equals sign
  - Expected '=' but none was found. Follow the correct expression format.
- Expected closed parenthesis
  - Expected an ')' but none was found. Follow the correct expression format.
- Expected underscore

    - Expected an ‘_’ but none was found. Follow the correct expression format. Check keywords are entered correctly.

- Expected the word ‘in’
    - Expected the word ‘in’ but none was found. Follow the correct expression format. Check keywords are entered correctly in IfExpr.

- Expected the word ‘then’
    - Expected the word ‘then’ but none was found. Follow the correct expression format. Check keywords are entered correctly in IfExpr.

- Expected the word ‘else’
    - Expected the word ‘else’ but none was found. Follow the correct expression format. Check keywords are entered correctly in IfExpr.

- Expected open parenthesis
    - Expected an ‘(’ but none was found. Follow the correct expression format.

- Unexpected keyword
    - Keyword does not match grammar. Follow the correct expression format. Check keywords are entered correctly.

- Expected a digit or open parenthesis
    - Expected an integer or an ‘(’ but none was found. Follow the correct expression format.

- No adding functions
    - Functions cannot be added together.

- No multiplying functions
    - Functions cannot be multiplied together.

- Functions are not true or false
    - is_true method cannot be called on functions.

- No BoolVal call step
    - Method call_step cannot be called on BoolVal’s.

- Invalid BoolVal call
    - Method call cannot be called on BoolVal’s.

- No adding booleans
    - Booleans cannot be added together.

- No multiplying booleans
    - Booleans cannot be multiplied together.

- No NumVal call step
    - Method call_step cannot be called on NumVal’s.

- Invalid NumVal call
    - Method call cannot be called on NumVal’s.

- Numbers are not true or false

  - is_true method cannot be called on NumVal's.

- Did not return true or false
  - IfExpr did not evaluate to _true or _false.

- Incorrect else
  - Check the _else expression in IfExpr to make sure it fits the grammar.

- Free variable
  - There is a free variable in an empty environment.
  - You are trying to interpret a free variable.

API Documentation

Basic Usage

To interact with the MSDScript API a user should first create an input stream containing the expression that they would like to evaluate (e.g. inputStream = _let x = 5 _in _let y = z + 2 _in x + y + (2 * 3)). The user then inputs the input stream into the parse method (e.g. parse(inputStream)). Next, the user can indicate which mode of operation they would like to use for evaluation (value, optimize, step). Last, if the user desires an expression to be printed they can use the to_string method.

Classes

Expr is the parent class, which classes *NumberExpr, AddExpr, MultExpr, VariableExpr, LetExpr, BoolExpr, CompExpr, IfExpr, FunExpr, CallExpr* inherit from.

Description:
The Expr class represents various types of expressions useful for evaluation common mathematical syntax. Expressions are combinations of symbols that are allowed in the MSDScript grammar, and contain operations to be evaluated.

Methods:
The virtual methods (below) of the Expr class are implemented in all child classes.

Method	Implementation	Returns
equals	virtual bool equals(PTR(Expr) e) = 0;	bool
	Evaluates equality of expressions and returns _true or _false.	
value	virtual PTR(Val) value(PTR(Env) env) = 0;	PTR(Val)
	Evaluates an expression and returns a pointer to a value.	
subst	virtual PTR(Expr) subst(std::string var, PTR(val) val) = 0;	PTR(Expr)
	Tests for equivalence between input variable and member variable. Substitutes if equivalent.	
containsVar	virtual bool containsVar() = 0;	bool

Method	Implementation	Returns
containsVar	Determines whether an expression contains a variable and returns _true or _false.	
optimize	virtual PTR(Expr) optimize() = 0;	PTR(Expr)
	Simplifies expressions and returns a pointer to an expression that is no longer than the original expression.	
to_string	virtual std::string to_string() = 0;	std::string
	Converts the result of a method to a string type.	
step_interp	virtual void step_interp() = 0;	void
	Evaluates an expression, just as value(), but solves issues with memory overflow. Only accessible from the interp_by_steps method in the Step class.	

NumberExpr

Description:

Represents an integer as an expression.

Member Variables:

Int val;

Constructor:

NumberExpr(int val);

Methods:

No additional methods.

AddExpr

Description:

An expression that performs addition on two expressions.

Member Variables:

PTR(Expr) lhs;
PTR(Expr) rhs;

Constructor:

AddExpr(PTR(Expr) lhs, PTR(Expr) rhs);

Methods:

No additional methods.

MultExpr

Description:

An expression that performs multiplication on two expressions.

Member Variables:

PTR(Expr) lhs;
PTR(Expr) rhs;

Constructor:

    MultExpr(PTR(Expr) lhs, PTR(Expr) rhs);

Methods:

    No additional methods.

VariableExpr

Description:

    Represents a variable as an expression.

Member Variables:

    std::string val;

Constructor:

    VariableExpr(std::string val);

Methods:

    No additional methods.

LetExpr

Description:

    An expression used for assigning variables within an expression.

Member Variables:

    std::string var;
    PTR(Expr) expr1;
    PTR(Expr) expr2;

Constructor:

    LetExpr(std::string var, PTR(Expr) expr1, PTR(Expr) expr2)

Methods:

    No additional methods.

BoolExpr

Description:

    An expression used to represent Boolean values.

Member Variables:

    bool rep;

Constructor:

    BoolExpr(bool rep);

Methods:

    No additional methods.

CompExpr

Description:

    An expression used to test the equivalence of expressions.

Member Variables:

PTR(Expr) lhs;
PTR(Expr) rhs;

Constructor:
CompExpr(PTR(Expr) lhs, PTR(Expr) rhs);

Methods:
No additional methods.

<u>IfExpr</u>
Description:
An expression used to evaluate the condition of an expression and provides branches of execution based on evaluation.

Member Variables:
PTR(Expr) ifExpr;
PTR(Expr) thenExpr;
PTR(Expr) elseExpr;

Constructor:
IfExpr(PTR(Expr) ifExpr, PTR(Expr) thenExpr, PTR(Expr) elseExpr);

Methods:
No additional methods.

<u>funExpr</u>
Description:
An expression used to represent a function.

Member Variables:
PTR(Expr) arg;
PTR(Expr) body;

Constructor:
funExpr(PTR(Expr) arg, PTR(Expr) body);

Methods:
No additional methods.

<u>callExpr</u>
Description:
An expression used to implement function calls through substitution.

Member Variables:
PTR(Expr) toBeCalled;
PTR(Expr) actualArg;

Constructor:
callExpr(PTR(Expr) toBeCalled, PTR(Expr) actualArg);

Methods:

No additional methods.

Val is the parent class, which classes *NumVal, BoolVal, FunVal* inherit from.

Description:
The Val class represents values such as integers, Booleans, and functions. These values are often returned as the result of expression evaluation.

Methods:
The virtual methods (below) of the Val class are implemented in all child classes.

Method	Implementation	Returns
equals	virtual bool equals(PTR(Val) val) = 0;	bool
	Evaluates equality of Vals and returns _true or _false.	
add_to	virtual PTR(Val) add_to(PTR(Val) other_val) = 0;	PTR(Val)
	Adds two Vals and returns the result as a pointer to a Vals.	
mult_with	virtual PTR(Val) mult_with(PTR(Val) other_val) = 0;	PTR(Val)
	Multiplies two Vals and returns the result as a pointer to a Vals.	
to_expr	virtual PTR(Expr) to_expr() = 0;	PTR(Expr)
	An expression that returns an expression form of a Val.	
to_string	virtual std::string to_string() = 0;	std::string
	Simplifies expressions and returns a pointer to an expression that is no longer than the original expression.	
call	virtual PTR(Val) call(PTR(Val) actualArg) = 0;	PTR(Val)
	Carries out a function call through the use of substitution.	
is_true	virtual void is_true() = 0;	bool
	Returns whether BoolVals are _true or _false.	

NumVal
Description:
A Val used to represent an integer value.

Member Variables:
int rep;

Constructor:
NumVal(int rep);

Methods:

No additional methods.

BoolVal
Description:
A Val used to represent a Boolean value.

Member Variables:
int rep;

Constructor:
BoolVal(int rep);

Methods:
No additional methods.

FunVal
Description:
A Val representation of a function.

Member Variables:
std:string formalArg;
PTR(Expr) body;
PTR(Env) env;

Constructor:
FunVal(std:string formalArg, PTR(Expr) body, PTR(Env) env);

Methods:
No additional methods.

Env is the parent class, which classes *EmptyEnv and ExtendedEnv* inherit from.

Description:
An Env is an Expr paired with a dictionary and is used to pair a Val with a variable to imitate substitution. Env's keep track of the substitution history, and different Env's represent different substitution histories. Envs can be empty, or they can be extended where an object value pair is stored along with a pointer to another shared Env.

Methods:

Method	Implementation	Returns
lookup	virtual PTR(Val) lookup(std:string find_name) = 0;	PTR(Val)
	Looks up the Val associated with string name inside an environment.	

EmptyEnv
Description:
An Env with no mapping between a variable and a value.

Member Variables:

None

Constructor:
New(EmptyEnv)();

Methods:
No additional methods.

ExtendedEnv
Description:
An Env with a mapping between a variable and a Val, and a pointer to a shared environment.

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

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

Methods:
No additional methods.

<u>**Parse** - *methods only*</u>
Description:
Parse allows a user to translate a valid input stream into an expression to be evaluated. The parse method, PTR(Expr) parse(std::istream &in), makes recursive calls to the methods described below. These methods are called exclusively by parse and are not intended to be interacted with by the user.

Constructor:
None

Methods:
static PTR(Expr) parse_expr(std::istream &in);

static PTR(Expr) parse_comparg(std::istream &in);

static PTR(Expr) parse_addend(std::istream &in);

static PTR(Expr) parse_number_or_paren(std::istream &in);

static PTR(Expr) parse_number(std::istream &in);

static PTR(Expr) parse_variable(std::istream &in);

static char peek_next(std::istream &in);

static PTR(Expr) parse_let(std::istream &in);

static std::string parse_strings(std::istream &in);

```
static std::string parse_str_error(std::string s);

static PTR(Expr) parse_str(std::string s);
```