MSDScript Manual

Jonathan Sullivan
16APR20

Description:

MSDScript is intended to be a command-line or XCode program that can parse and then interpret or optimize provided strings or other string-like input streams comprised of numbers, variables, booleans, addition operations (and subtraction through the use of negative numbers), multiplication operations, equality operations, and defined functions. At its most basic level, this program can function as a calculator capable of processing Fibonacci numbers and factorials, but it can also be used as part of a larger program, including, for example, calculating dates for a calendar program.

“Parsing” converts the provided string into expressions comprehensible by the program. The program can then use these expressions for interpretation or optimization.

“Interpreting” is providing the solution - if any is available - of those expressions. For example, “`2 + 2`” would interpret to “`4`”. However, if the expression is not interpretable, the program will exit with an error condition.

“Optimizing” is ideally reducing the expressions to a more-concise form, though not all expressions can be shortened. For example, if “x” is not defined, “`x + 2 + 3`” would optimize to “`( x + 5 )`”; however “`x + 5`” would optimize to “`( x + 5 )`”.

The concepts of “interpretation” and “optimization” can output their results as strings, which can then be printed in the terminal or used in wrapper programs.

Build and Installation Instructions:

If you have XCode available:

Please extract the full archive to your intended location, load the “MSDScript.xcodeproj” file in this directory into your XCode program, and then Run (the play icon or CMD+R) or Test (the wrench icon or CMD+U) as you desire.

Warning: The SeparateTestingSuite directory must be present in this directory for either running or testing the program in XCode.

Note: Input optimization is not available in the XCode interface.

From the Command Line:

If your system has CMake or a similar program installed, please extract the full archive to your intended location, navigate to ./MSDScript/, and enter the command: "`make`". This will produce an executable file at ./MSDScript/MSDScript.

Alternatively, if you do not have CMake or just want to build manually, please navigate to ./MSDScript/ , and run the following command: "`g++ -std=c++17 -o MSDScript *.cpp`" . This will produce an executable file at ./MSDScript/MSDScript .

Note: The SeparateTestingSuite directory is not required for command-line use.

User Guide:

Grammar:

"Expressions", or "Expr", provide the programmatic building blocks for MSDScript, and they are built out of the following grammar, using the syntax as described after the grammar:

```
[Expr]     =     [Comp]
           =     [Comp] == [Expr]

[Comp]     =     [Additive]
           =     [Additive] + [Comp]

[Additive] =     [Mult]
           =     [Mult] * [Additive]

[Mult]     =     [Inner]
           =     [Mult] ( [Expr] )

[Inner]    =     [Number]
           =     ( [Expr] )
           =     [Variable]
           =     _let [Variable] = [Expr] _in [Expr]
```

```
=     _true
=     _false
=     _if [Expr] _then [Expr] _else [Expr]
=     _fun ( [variable] ) [Expr]
```

Syntax:

The MSDScript can handle the following syntax:

- Integers or `[Number]`: “1”, “42”, “-13”, etc. **Note:** decimals are not permitted, though negative numbers are allowed.
- Addition operator: “+”. Subtraction can be accomplished by adding a negative number.
- Multiplication operator: “*”.
- Equality test operator: “==”.
- Parentheses: “(“ and “)”. **Note:** parenthetical expressions can be nested, but all opening parentheses must be matched by a closing parenthesis.
- Booleans: “`_true`” and “`_false`”.
- `[Variable]`. Variable names are exclusively English alphabet letters, and must be declared in the following fashion: “`_let [variablename] = [expression describing variable’s value] _in [expression where variable occurs]`”. Variable declarations can be nested (i.e. “`_let x = (_let y = 1 + 7 _in y+2) _in x+4`” interprets to “`14`”); however, the lowest-level declaration takes priority over all others (i.e. “`_let x = 5 _in _let x = 6 _in x`” interprets to “6”). **Note:** The variable will exclusively be available in the expression where the variable occurs, and cannot be referenced inside of the expression describing the variable’s value. **Warning:** It is possible to use variables in an input where they have not been bound or defined - such as “`_let x = 5 _in y + 7`”. This input can be optimized, but cannot be interpreted, and will error with exit code 2.
- If-then-else statements. These are declared in the following fashion: “`_if [expression to be tested] _then [input to parse and interpret/optimize if true] _else [input to parse and interpret/optimize if false]`”.
- Functions. Functions are described by: “`_fun ([variablename]) [expression including that variable]`”. Please note that in order to have the function body interpreted, the full declaration is: “`( _fun ([variablename]) [expression including that variable] ) ( [value for variable] )`”. Alternatively, functions may be assigned to variable names themselves, but those variable names must still obey the above requirements.
- Spaces. Any number of spaces may be used at any point in the input, with the exception of inside variable names. **Note:** numbers separated by a space will be treated as separate numbers.

There are these additional details:

- The order of mathematical precedence is as follows: contents of a parenthetical phrase, then multiplication, then addition.
- The program is right-associative for mathematical operators, and left-associative for function calls.
- Multiplication can only be written as “[item1] * [item2]”, not “([item1])([item2])” and not “[firstvariable][secondvariable]”.
- If MSDScript cannot parse an input, it will exit with a code of 1.
- If MSDScript can parse the input, but cannot interpret it, it will exit with a code of 2.
- Optimization will never error, though may not necessarily shorten the input.

In order to provide an input to the program from either the command line or IDE interface, please type: “`[intended input] + RETURN + CTRL+D`”.

To Interpret the Input:

In Xcode:

Please load the project in XCode, press the play / arrow button or CMD+R, and type in the desired input in the Debug area of the program, following the above format.

Example input and output:

```
_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
```

Note: Executing the program in XCode only allows for interpretation, not optimization.

In the Command Line:

Once the program has been compiled as above, please execute “`./MSDScript/MSDScript`” and type in the desired input at the waiting prompt, following the above format.

Example input and output:

```
_let f = (_fun (x) (_fun (y) x*x + y*y)) _in (f(2))(3)
13
```

Note: For exceptionally recursive inputs or if your desired input is resulting in segmentation faults, the option of "`./MSDScript/MSDScript --step`" is available. For example, a function that counts down from 1,000,000 to 0 could be written as "`_let countdown = _fun(countdown) _fun(n) _if n == 0 _then 0 _else countdown(countdown)(n + -1) _in countdown(countdown)(1000000)`", and can only be successfully interpreted with the --step option.

To Optimize the Input:

In Xcode:

This option is not available.

In the Command Line:

Please execute "`./MSDScript/MSDScript --opt`" and type the desired input at the waiting prompt following the above format. Please note that the output will have each final expression identified by the program separated by parentheses, so the result may not match your original syntax, and may not be smaller than the original input.

Example input and output:

```
x + 1 == x + 1
( x + 1 ) == ( x + 1 )
```

To Run the Included Tests:

In XCode:

Please load the project into XCode as above, long-press on the play/right-arrow button and select "Test" or press CMD+U, and then allow the tests to proceed. Their results will be visible in the Debug window at the bottom, though be advised that the tests can take up to half a minute to complete.

Example output:

```
Test Suite 'All tests' started at 2020-03-31 12:49:55.114
Test Suite 'SeparateTestingSuite.xctest' started at 2020-03-31 12:49:55.114
Test Suite 'test' started at 2020-03-31 12:49:55.114
Test Case '-[test testAll]' started.
================================================================================
All tests passed (167 assertions in 24 test cases)

Test Case '-[test testAll]' passed (19.278 seconds).
Test Suite 'test' passed at 2020-03-31 12:50:14.393.
     Executed 1 test, with 0 failures (0 unexpected) in 19.278 (19.279) seconds
Test Suite 'SeparateTestingSuite.xctest' passed at 2020-03-31 12:50:14.394.
     Executed 1 test, with 0 failures (0 unexpected) in 19.278 (19.280) seconds
Test Suite 'All tests' passed at 2020-03-31 12:50:14.394.
     Executed 1 test, with 0 failures (0 unexpected) in 19.278 (19.280) seconds
Program ended with exit code: 0
```

In the Command Line:

This option is not available.

Reading from Files:

When using MSDScript from the command line, it is also possible to interpret and optimize MSDScript-acceptable syntax stored as plaintext in files. The files "./MSDScript/fact.msd" and "./MSDScript/fib.msd" are provided as examples of how to structure new files. In order to interpret a file, please execute "`./MSDScript/MSDScript [filename, including path]`"; for example, "`./MSDScript/MSDScript ./MSDScript/fact.msd`". Flags for optimizing the input or allowing the interpreter to handle deeper recursion can be added between the executable and the file name; for example, "`./MSDScript/MSDScript --opt ./MSDScript/fact.msd`" yields, "`( ( _fun (factrl) _fun (x) _if x == 1 _then 1 _else ( x * ( ( factrl ) ( factrl ) ) ( ( x + -1 ) ) ) ) ( _fun (factrl) _fun (x) _if x == 1 _then 1 _else ( x * ( ( factrl ) ( factrl ) ) ( ( x + -1 ) ) ) ) ( 10 )`".

Examples of Valid Inputs and Their Stringified Interpretations:

- "`_let y = 7 _in (y + 3) * 2`" interprets to "`20`"
- "`_let x = (_let y = 1 + 7 _in y+2) _in x+4`" interprets to "`14`"
- "`_if 1 + 2 == 3 _then 100 _else 0`" interprets to "`100`"
- "`_let f = _fun (x) x + 1 _in f(10)`" interprets to "`11`"
- "`_let f = (_fun (x) (_fun (y) x*x + y*y)) _in (f(2))(3)`" interprets to "`13`"
- "`(_true == _true) == 2`" interprets to "`_false`"

- "`_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)`" interprets to "`89`"
- "`_fun (x) x + 1`" interprets to "`[function]`"

Examples of Valid Inputs and Their Optimized Versions:

- "`_let x = 5 _in x + x`" optimizes to "`10`". Interpreting will yield the same result.
- "`_let z = 17 _in x * 2`" optimizes to "`( x + 2 )`". This input cannot be successfully interpreted due to "x" being unbound, and will error with exit code 2.
- "`_let y = 8 _in _let x = 5 _in y`" optimizes to "`8`". Interpreting will yield the same result.
- "`x + 1 == x + 1`" optimizes to "(x + 1) == (x + 1)". This input cannot be successfully interpreted due to "x" being unbound, and will error with exit code 2.
- "`_if _true _then x _else y`" optimizes to "`_if _true _then x _else y`". This input cannot be successfully interpreted due to "x" and "y" being unbound, and will error with exit code 2.
- "`_fun (x) x + 1`" optimizes to "`[function]`". Interpreting will yield the same result.

API Documentation:

In order to use this program without its included main.cpp file, please navigate to ./MSDScript/ and run the command "`make libmsdscript.a`". This will create a static library at ./MSDScript/libmsdscript.a which you can include and compile with your main.cpp or equivalent other file. **Warning:** C++17 is required for the MSDScript.

Alternatively, please have your Main file or other appropriate file include "parse.hpp". **Note:** all other files (with the exception of main.cpp) must be present in the MSDScript directory for command line use, and all files in this archive must be present in their current structure for XCode use..

Return Variables:

In order to use the parsing functionality of MSDScript, an expression variable to receive the return from parse() will need to be created. The easiest way to accomplish this is "`PTR(Expr) [variableName] = parse([incomingStream]);`".

If you desire to use the Expression output from optimize() rather than simply convert it straight to a string, another Expr variable will be necessary.

Likewise, if you desire to use the Value output from interp() or interp_by_steps(), a Var variable will be necessary; for example, "`PTR(Value) test = outputExpression->interp(env);`".

The five primary methods to interact with the program are as follows:

parse(std::istream &in);

Parse() takes an input stream - which can be converted from a string or other appropriate source - and starts the parsing process to generate a nested, interrelated programmatic expression chain. This expression chain is of type PTR(Expr), and can later be interpreted and rendered as a string, but cannot be output as a string itself. If the input stream cannot be parsed, parse() will produce an error with an exit code of 1.

Note: parse() throws exceptions if it encounters unparseable input.

Example parse-handling function:

```
static PTR(Expr) parse_str(std::string s)
{
    std::istringstream in(s);
    return parse(in);
}
```

[programmatic expression]->interp([environment]);

Interp() takes an expression, either created by parse() or by hand, and attempts to "solve" the mathematical or programmatic construct defined by that expression. Note that it must be passed an empty Environment, which can be generated with the following code: "`PTR(EmptyEnv) env = NEW(EmptyEnv)();`". This function returns a programmatic value of the interpreted expression of type PTR(Value) which can be converted a string, or an error with an exit code 2 if the expression cannot be interpreted.

Note: interp() throws exceptions if it encounters uninterpretable expressions.

Example interpretation-handling function:

```
static PTR(Value) interp_str(std::string s)
{
    PTR(EmptyEnv) env = NEW(EmptyEnv)();
    std::istringstream in(s);
    return parse(in)->interp(env);
}
```

Step::interp_by_steps([parsed expression]);

Stepping through a given input allows greater depths of recursion, and also avoids the need to define an empty Environment before attempting the interpretation. The easiest way to generate the [parsed expression] is by way of the parse() command itself. This function returns a programmatic value of the interpreted expression of type PTR(Value) which can be converted to a string, or an error with an exit code 2 if the expression cannot be interpreted.

Note: interp_by_steps() throws exceptions if it encounters uninterpretable expressions.

Example interpretation-by-steps-handling function:

```
static PTR(Value) interp_step_str(std::string s)
{
    std::istringstream in(s);
    PTR(Expr) parsedExpression = parse(in);
    return Step::interp_by_steps(parsedExpression);
}
```

[programmatic expression]->optimize();

This takes a programmatic expression, generally produced by the parse() function, and attempts to reduce it to its shortest form. For example, optimizing “x + 2 * 3 + y” without having associated _let declarations for x or y would optimize to “(x + (6 + y))”. This program returns a programmatic expression that can be converted to a string, and never errors, though may not necessarily reduce the size of the input.

Example optimization-handling function:

```
static PTR(Expr) optimize_str(std::string s)
{
    std::istringstream in(s);
    return parse(in)->optimize();
}
```

[programmatic expression]->to_string(); and [programmatic value]->to_string();

The outputs of interp(), interp_by_steps(), and optimize() can all be rendered in human-readable strings. The to_string() functions for both programmatic values and expressions will produce a concatenated string of either the answer (in the case of both interpretation methods) or the optimized input.

Example of how to take input from the command line, parse, optimize / interpret / interpret with deeper recursion support, and print to the command line:

```
PTR(EmptyEnv) env = NEW(EmptyEnv)();
PTR(Expr) outputExpression = parse(std::cin);
std::cout << outputExpression->optimize()->to_string() + "\n";
std::cout << outputExpression->interp(env)->to_string() + "\n";
std::cout << Step::interp_by_steps(outputExpression)->to_string() + "\n";
```