MSDScript Documentation

1. Overview

MSDScript is scripting language written in C++ capable of evaluating basic mathematical expressions, setting variables with values, evaluating boolean statements and basic functions. MSDScript interprets from the command line, but can also take any C++ iostream class objects as input. The program is portable in that it is able to run on any platform capable of compiling and running C++.

2. Getting Started

2.1 Installation

1. Download msdscript_package.zip and unzip to directory of choice.
2. Navigate into msdscript_package directory in terminal or shell or choice. NOTE: shell should have 'make' or equivalent installed. Comes standard installed on Unix systems.
3. Enter "make" while in directory with Makefile.
4. Makefile will compile and link source files with an output of msdscript executable.
5. "./msdscript --help" will output a list of possible msdscript modes and their flags, discussed in detail in the User Guide section of this document.

6. To uninstall or rebuild, running "make clean" in the directory with the Makefile will remove the executables and object code files generated, allowing "make" to be rerun.

3. User Manual

3.1 Command Line Flags

MSDScript does not accept multiple flag values. Including multiple flags, such as "./msdscript --help --interp" will run the first and terminate.

3.1.1: --help

"./msdscript --help" will list possible flags

3.1.2: --test

"./msdscript --test" will run unit test code using C++ catch.h. This can be used to verify that the build was completed and runs as expected. "Tests failed" will be output if tests do not complete.

3.1.3: --interp

"./msdscript --interp" will allow the user to enter an expression to be interpreted in as many or as few lines as desired. An end of file signal on the shell (CTRL + D) will end the input and msdscript will output the value of the entered expression.

When interpreted, expressions associate to the right. For example, the input "2 + 2 + -2" will result in the "2 + -2" being evaluated before the "2 + 2", as in standard expression interpretation. If the

intention was for the "2 + 2" to be interpreted first, the expression can be input as "(2 + 2) + -2".

For example, if "2 + 2" is input and an EOF is input, mscsript will output "4", the value of the expression given.

3.1.4: --step

"./msdscript --step" also interprets the user inputted expression until end of file signal. Expressions are interpreted in 'steps', allowing larger recursive functions and programs without segmentation faults. Use this flag is "--interp" is outputting segmentation fault errors.

3.1.5: --print

"./msdscript --print" takes user inputted expressions and prints out the expression formatted with parentheses showing precedence around all expressions.

For example, "2 + 2 + -2" will result in the output "(2+(2+-2))".

3.1.6: --pretty-print

"./msdscript --pretty-print" takes user inputted expressions and prints out the expression with a readable, indented formatting regardless of input formatting.

For example, "2+2+-2" will result in output "2 + 2 + -2".

"_let x = _true _in _if x _then 1 _else 2" will result in ouptut:

```
_let x = _true
 _in _if x
   _then 1
```

_else 2

3.2 Expressions

Expressions make up building block of msdscript statements. Valid expressions all have an associated value when interpreted, whether a boolean result or integer. Variables do not have a value when interpreted, unless assigned one in a "_let" expression (see below).

Many expressions are preempted with a keyword, which is a word with an underscore appended to the beginning. Expressions can be nested inside of each other, for example "(2+(2+2))", with an addition expression nested within another addition expression.

Expressions with keywords do not necessarily need spacing or line breaks to be interpreted correctly, if using parentheses. Spaces are required between keywords and values.

Expressions are interpreted with a precendence to the right. See entry for "--interp" flag.

3.2.1 Integers

The basic building blocks. Examples include "2", "-2", or "13", with their values being their associated integer values.

3.2.2 Booleans

Boolean values are true or false statements. Input to the interpreter as "_true" or "_false". "true" will be treated as a variable, and is not a keyword.

3.2.3 Variables

Any character or combination of characters or integers without spacing is considered a variable. Variables do not have an intrinsic value, resulting in a runtime error if interpreted. Variables can be assigned a value with a "_let" expression (see below).
Examples of variables: test, 2test, test2, a, abricidabra

3.2.4 Equals Expression

An expression representing an evaluation of whether two expressions are equal or not. When interpreted, results in "_true" or "_false". Boolean expressions, variable expressions and integer expressions, if compared to each other, will result in a runtime error.
Examples of equals expression:
"2 == 2" results in "_true"
"2 == 1" results in "_false".
"_true == _true" results in "_true".
"_true == 2" results in a runtime error.
"a == a" results in "_true".
"a == 2" results in a runtime error.

3.2.5 Addition and Multiplication Expressions

Addition and multiplication expressions interpret to their values as added or multiplied together. Only integers or bound variables within a "_let" expression can be added or multiplied.
Examples of addition and multiplication expressions:
"2 + 2" results in "4"

"2 + -2" results in "0"

"2 * 4" results in "8"

"-2 * 4" results in "-8"

3.2.6 Let Expressions

Let expressions assign a integer or boolean value to a variable, and then give an 'body' expression whether that variable, if included, would have the assigned value. Let expressions have the format of:

_let 'variable name' = 'variable value'

_in 'body expression'

For example, interpreted as "4":

```
_let x = 2
_in x + 2
```

In the example above, the "x" doesn't necessarily need to be in the "_in" body statement.

"_let" expressions can be nested within each other. For example:

```
_let x = 5
_in  (_let y = 3
     _in  y + 2) + x
```

If there are multiple nested _let expressions using the same variable name, the last '_let' value bound to the variable takes precedence. For example, the interpreted value of the example below is 10, not 12, because 'x' is '3' in the nested _let expression:

```
_let x = 5
_in  (_let x = 3
     _in  x + 2) + x
```

3.2.7 If Expressions

'_if' expressions check a condition and depending on the boolean result, return the interpretation of a given set of expressions. Let expressions have the format of:

```
_if 'expression interpreted as boolean'
_then 'if _if statement true, return interpretation of this expression
_else 'if _if statement is false, return interpretation of this expression
```

For example, this expression is interpreted as "2"

```
_if 2 == 2
_then 2
_else 4
```

'_if' and '_let' expressions can be nested within each other. For example, interpreted as '8':

```
_let x = 5
_in  _if x == 4
     _then 2
     _else 8
```

3.2.8 Function Expressions

Function expressions allow the substitution of a value into a predefined expression, typically paired with a '_let' expression. Functions have the '_fun' keyword followed by a variable expression and a 'body' expression where any variable in the 'body' expression is replaced with the value if called in a '_let' expression.

Function expressions have the format of:

```
_fun ('variable name to be replaced in body') 'body expression
```

For example, a valid function that returns no value if interpreted because the variable has no intrinsic value:

```
_fun (x) x + 1
```

An example of a function that interprets as 11:

```
_let f  = _fun (x) x + 1
          _in f(10)
```

Functions can also be nested. For example, this expression interprets as 7:

```
_let f = _fun (g) g(5)
_in _let g = _fun (y) y + 2
    _in f(g)
```

4. API Usage

4.1 Setup

Included in the .zip folder is 'libmsdscript.a', a static C++ msdscript library. To use it in a program, include all '.h' files in the desired program and link using g++ with the command:

'g++ "object files that included all .h files and use msdscript" libmsdscript.a'

The output executable will be able to call msdscript functions.

4.2 MSDScript API Usage

The most common useage of MSDScript, after including all header files, is to call the function: 'parse_str_tostring(std::string)', which takes a given string in valid expression syntax, parses it and returns a string representation of the interpreted value of the given expression string.

For example:

```
std::cout << parse_str_tostring("2 + 2") << std::endl;
```

Would result in output of the string "4".