MSDscript documentation

I. Description

MSDscript is a simple implemented language that is able to perform some mathematical operations and parse. Even though it is lacking several kinds of operations, it still can be the kinds of languages that are embedded in applications besides can be run as a command-line program. For example, someone implementing a calendar program might want to have a language for advanced users to calculate dates for repeated meetings (say, more sophisticated than "every Tuesday"), and MSDscript could just about work for that.

II. Getting Started

1. Building the executable

The main language is used in MSDscript is C++, and then in order to build the executable, users need to:

Step 1: Put all the source codes in one directory.

Step 2: Go to that directory in terminal

Step 3: Run the following command line:

```
$ make                    //running Makefile to create the executable
```

After generating the executable, users are able to perform all the implemented operations by:

```
$ ./msdscript —help //listing of all the modes can be used inside of the program
$ ./msdscript —test  //running the test case
```

and so on…

2. Embedded in applications

In order to embedded MSDscript in applications, users need to:

Step 1: Put the application in the same directory with the source codes.

Step 2: Include suitable header files into to the program .cpp file.

Step 3: Go to that directory in terminal.

Step 4: Run the following command lines:

```
$c++ —std=c++14 -c cmdline.cpp cont.cpp env.cpp expr.cpp parse.cpp step.cpp tests.cpp val.cpp        //generating object files
$ar -ruv libmsdscript.a cmdline.o cont.o env.o expr.o parse.o step.o tests.o val.o        //generating the library
$c++ -o program_name program.cpp libmsdscript.a
```

After that, users can totally perform the —interp function that is implemented in MSDscript with their program.

For example, in order to embed MSDscipt in which_day program to take a week number (counting form 0) as a command-line argument and tell which day to meet that week (0 = Sunday, 1 = Monday, etc.), users need to:

Step 1: Put all the source codes and which_day.cpp in one directory.

Step 2: Modify the main function by adding #include “cmdline.h”, #include “env.h”, #include “parse.h”, #include “var.h”, and adding the following line of code:

```
std::cout << parse_str(expr)->interp(Env::empty)->to_string() << std::endl;
```

Step 3: Go to the directory in terminal.

Step 4: Run the following command lines:

```
$c++ —std=c++14 -c cmdline.cpp cont.cpp env.cpp expr.cpp parse.cpp step.cpp tests.cpp val.cpp
$ar -ruv libmsdscript.a cmdline.o cont.o env.o expr.o parse.o step.o tests.o val.o
$c++ -o which_day which_day.cpp libmsdscript.a
$./which_day 13
```

And the program will return the result is 4, meaning that you meet on Thursday in week 13.

III. User Guide

For now, these are modes using inside of the MSDscript:

—help;

—test;

—interp;

—step;

—print;

—pretty-print;

1. $./msdscript —help: listing all of the available command-line that are able to use.
2. $./msdscript —test: running all the test cases to make sure the program is correctly functioning.
3. $./msdscript —interp: performing mathematical operations. For example, if users want to interpret an addition of 17 and 24, users need to run the following command lines:

   ```
   $ ./msdscript —interp
   17 + 24
   ctrl D          //to notice that this is the end of input from the user
   ```

   Then the program will return the result is 41.
4. $./msdscript —step: performing mathematical operations and return the same result with —interp mode, but implements its own continuations instead of using the C++ stack. Therefore, it could prevent the program from being crash and result faster.
5. $./msdscript —print: printing the expressions in a regular way with all corresponding brackets. For example, if users want to print the expression of 17 multiply by x then the result multiply by 24, they need to run the following command lines:

   ```
   $ ./msdscript —print
   (17*x)*24
   ctrl D          //to notice that this is the end of input from the user
   ```

   Then the program will display the result is ((17*x)*24).
6. $./msdscript —pretty-print: printing the expressions in a pretty way with all necessary brackets and spaces. For example, if users want to print the expression of 17 multiply by x then the result multiply by 24, users need to run the following command lines:

   ```
   $ ./msdscript —pretty-print
   (17*x)*24
   ctrl D          //to notice that this is the end of input from the user
   ```

   Then the program will display the result is (17 * x) * 24.

IV. MSDscript API

1. Program a Backus-Naur Form (BNF) grammar

```
〈expr〉    = 〈number〉
           | 〈variable〉
           | 〈boolean〉
           | 〈expr〉 == 〈expr〉
           | 〈expr〉 + 〈expr〉
           | 〈expr〉 * 〈expr〉
           | 〈expr〉 ( 〈expr〉 )
           | _let 〈variable〉 = 〈expr〉 _in 〈expr〉
           | _if 〈expr〉 _then 〈expr〉 _else 〈expr〉
           | _fun( 〈variable〉 ) 〈expr〉
```

2. Built-in funtions

2.1. number: represents number values, can be optional hyphen followed by digits with no decimal point to perform either positive or negative numbers.

For example: 17 << result: 17

-24 << result: -24

2.2. variable: represents variable values by string that is performed by one or more alphabetic characters.

For example: x << result: x

abc << result: abc

2.3. boolean: represents true or false and must really be distinct from number values.

For example: true << result: _true

false << result: _false

2.4. ==: represents equality expressions, can work on any kinds of values, and it's even allowed to compare a boolean to a number. However, only booleans are equal to booleans, and only numbers are equal to numbers. It always produces a boolean value.

For example: 1 == 1 << result: _true

_true == _false << result: false

x == 1 << result: error "Free variable: x"

2.5. +: represents addition expressions of two or more number values.

For example: 17 + 24 << result: 41

*2.6. *:* represents multiplication expressions of two or more number values.

For example: 17 * 24 << result: 697

2.7. () function-call: is used to evaluate the function with a given variable. In order to perform the final result, the program will interpret the actual_arg expression first. Then, it will substitute that value into the body expression of the formal_arg expression if there is the same variable exists inside of the body.

For example: (_fun (x) x+2)(1) << result: 3

2.8. _let: binds a value to a variable if the variable exists in the body of the expression. In order to perform the final result, the program will interpret the right-hand side of the _let expression first, then it will look up in the body to see if there is the same variable exists in the body. If yes, substitute the variable by the right-hand side into the body, then interpret the body; otherwise, just interpret the body itself.

For example: _let x = 17 _in x + 24 << result: 41

_let x = 17 _in 24 << result: 24

2.9. _if: is used to represent the condition of two expressions. In order to form this expression, it requires its first subexpression to produce a boolean value. If the condition returns true, the function will interpret the expression in _then, otherwise interpret _else, not both.

For example: _if (24 == 17) _then 1 _else 0 << result: 0

2.10. _fun: is used to represent the function expressions with variable name

For example: _fun (x) (x+17) << result: _fun (x) (x+17)

3. Additional notes

- Whitespaces can appear between any two things in the grammar.
- Parentheses can be added around any expressions, and they should be put around the smallest expression needed to resolve an ambiguity.
- Operator precedence and associativity:
    - Only call function is left-associative, and the rest is right-associative

- () call function is higher precedence than *
- * is higher precedence than +
- _let expression is weaker than + and *, and should have parentheses only when needed to avoid ambiguity
- _let, _if, and _fun are the same level of precedence

V. Potential Error

When performing —interp mode, the program may crash due to stack overflow. In order to solve this, switching to —step mode instead of —interp, which implements its own continuations instead of using the C++ stack. Then interpreter in —step mode should never crash by exhausting the C stack.

VI. Report Bugs

For additional information or bugs reporting, please feel free to contact the author through email: ngahuynh1291@gmail.com

VII. License