MSDScript

by ***Xuefeng Xu***
Version: *1.0.1*

MSDScript is a programming language backed by an interpreter. MSDScript offers a range of tools and features that empower the programmers with more easily writing memory-leak free and robust code. MSDScript support addition, multiplication, local variables, functions and code optimizer.

The MSDScript interpreter can be used as a command line program. With the CLI mode, user can type in certain expressions and execute expressions. With the Script mode, user can feed in a script with suffix `.msd` and the interpreter will execute the script. The MSDScript CLI can also be used as a code optimizer with no interpreting.

THe MSDScript can also be used as an embedded language to support the programer build the application. 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.

Contents:

Getting started

MSDScript provides a makefile that the user can customize to meet the requirement.

MSDScript can be used in three ways:

To build MSDScript as a command line program, the user do not need to change anything in the makefile. run the following command directly in the terminal.

```
cd ./config
make
cd ../build
./msdscript
./msdscript --script ../test/test.msd
```

To embed MSDScript as part of an application, user can choose to provide their own main.cpp file or use MSDScript as a library. The build way is slightly different.

1. To provide a customized main function, change the main source variable `MAIN_SOURCES` in makefile (with a customized main.cpp file) and remake the application.

   ```
   cd ./config
   vim makefile
   # Change the makefile with a customized main.cpp file
   make
   cd ../build
   ./msdscript
   ```

2. To use MSDScript as a library, build the lib first and compile with other source code statically.

   ```
   cd ./config
   make msdscriptlib.a
   ```

The executable program will be in the `./build` folder with name:

- **msdscript** -- executable command line program
- **msdscriptlib** -- static library

User guide

User guide is mainly for users who use MSDScript as a command line program. But input and output types should be the same.

Introduction

MSDScript takes an expression and interpret it to a value.

Expression is a combination of one or more constants, variables, operators, and functions that the programming language interprets (according to its particular rules of precedence and of association) and computes to produce another value. This process, as for mathematical expressions, is called evaluation.

In MSDScript, expressions can be numbers, variables, additions, multiplications, booleans, functions, function calls, comparisons.

The parsed user input will be an expression and wait to be interpreted or optimized.

Value defines and constrains the data type interpreted from an expression.

In MSDScript, values can be numbers, booleans, and functions.

In particular, a variable does not have a value in nature. We need to bind a value to it. The value is associated with the variable in some specific scopes, which we will talk about at the implementation concepts section.

Interpretation will evaluate 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.

- Examples for interpretation:

Original Code	Result
_if _true _then 8 _else x	8
x + y	error
_let f = _fun (x) y _in f(2)	error
_let f = _fun (x) 2 _in f(y)	error
(_fun(x) 2+x)(3)	5

Optimization produces a semantically equivalent expression that is no larger than the input expression. The result of optimization does not have an expression that uses +, *, and == on two values. For a _let, if the right-hand side expression can be optimized to something without a variable, then the optimized form must have the substitution performed. Optimization does not call functions by substituting arguments for values.

- Examples for optimization:

Original Code	Optimized Code
_let y = 1 _in y + (2+x)	_let y = 1 _in y + (2+x)
(_fun (x) x)(3)	(_fun (x) x)(3)
(_fun (x) x+(2+3))(3)	(_fun (x) x+5)(3)
1+2+x	1 + 2 + x
(1+2)+x	3 + x

Command line arguments

1. Interpreter CLI: `./msdscript`
2. Interpreter with script: `./msdscript --script script.msd`
3. Optimizer CLI: `./msdscript --opt`

> Note: The interpreter and optimizer take exactly one expression.
> **Node: MSDScript support multiline inputs. When finished, type `Control-D` to execute.**

Input Type and Format

Values are numbers, booleans, and functions.

- **Only support positive/negative integers.** Number without prefix is a positive number. Number with a `-` prefix is a negative number.
- `+` means addition
- `*` means multiplication
- `==` means equality: same booleans or same numbers
- `_let` binds a expressionvalue (the right-hand side) to a name (the variable), where the name can be used in the body
  Example: `_let x = 10 _in x + y` optimized to means the same as `(10 + y)`
- `_if` is a conditional that lets you pick between two other expressions based on a boolean, the first if the boolean value is true, the second if the boolean value is false
- `_true` means boolean true
- `_false` means the boolean false
- `_fun (<var>) <expr>` is a function value, where `<var>` is meant to be replaced with a value in `<expr>` when the function is called

> **Specifying operations:**
> `+` `*` works on integer values only
> `==` works on integer values and boolean values only
> *"Works on" means interpreting to a value; otherwise errors*

Input requirements:

1. Variables only contain ASCII alphabetic characters (no numbers, special characters, or spaces). Variable names are case-sensitive.
2. All multiplication must be written as `Val1 * Val2`, not `(Val1)(Val2)` or `Val1Val2` (note: white space is ignored).
3. Multiplication has a higher precedence than addition, and the contents of parenthetical expressions have higher precedence than both (standard order of operations).
4. Functions of multiple arguments can be implemented by nesting `_fun` forms and `call` forms.
5. `=` by itself does not form an expression; it's only used with `_let`. For example use `_let x = 3` instead of `x = 3`.
6. Any number of spaces may be added anywhere other than the above limitations.

Output Information and Format

On stdin / exit codes:

- If matches the grammar, then
  - Exit 2 if interpret and error
  - Exit 0 if interpret not errors
  - Exit 0 always for optimize
- If not matches grammar (failed to parse), then
  - exits with code 1

On stdout:

- For interpret:
  - If no error:
    1. number (maybe "-" and then a nonempty sequence of digits)

2. boolean ("_true" or "_false") value
3. function "[function]"

> always followed by a newline, no extra spaces anywhere.
> The number should be followed by the a new line character '\n'.

- If error:
  - Error message (printed to stderr) should include the reason behind the failure of the interpret
- For optimize:
  - Parentheses are required around every expression except for variables, numbers, already parenthesized, and booleans;
  - parsing the output should produce the right expression (i.e., sending it back in to optimize would produce the same output)

> Examples of good output:
> `8, x, _true, _false, (1 + x), (2 * 8), (_true == _false),(_if _true _then 8 _else 9), (_fun (x) x), (f(10)), (_let x = 5 _in x)`

On stderr:

- For interpret:
  - nothing if no error, some type of identifying information regarding the cause of the error
  - if error
- For optimize: never provides output, as never errors

API documentation

Contents of API documentation

    2. lookup()
7. Class: Cont
    1. done
    2. void step_continue()
8. Class: Step
    1. mode_t
    2. mode
    3. expr
    4. env
    5. val
    6. cont
    7. interp_by_steps(PTR(Expr) e)

1. Implementation Concepts

Scope: Closures and Environment

To accurately imitate substitution, pair an expression and a dictionary The pair is called a **closure**. The dictionary is called an **environment**.

Each local variable's value reside in its own environment, just like other programming languages' terminology: scope. If a variable is out of the current scope, its value will be discarded. Trying to interpret a free variable will lead to an error.

Shared environments reflects shared substitution history. Propagate environments to sub expressions lazily. Different environments reflect different substitution histories.

Explicit Continuation

Explicit continuations are implemented in MSDScript for two reasons:

1. Make functions calls work for loops, like algebra-style simplification
2. Use available memory, instead of just available stack space

Interpret steps will replace the C++ stack for the interpreter. Specific classes will provide functions to interpret as we claimed. Global variables are used to make sure that continuation works.

Associative:

- Right-associative: `+` `*`.

    > Subtraction is handled by adding negative numbers.

    - Ex: `1+2+-3` parse() => `AddExpr(new NumExpr(1),new AddExpr(new NumExpr(2), new NumExpr(-3)));`
- Left-associative `function call Expr`
    - Ex: `f(1)(2)` parse() => `CallExpr(new CallExpr(new VarExpr(“f”),new NumExpr(1)),new NumExpr(2));`

Memory Leak

MSDScript uses reference count to eliminate memory leak (Garbage collector in the future). Some specific macros are used to replace original C++ style keyword to use the reference count. The macros used are described in the next section.

2. Macro

> `#include "pointer.hpp"`

MSDScript uses the following macro to make the code clear and simple:

Macro	**C++**
NEW(T)	`std::make_shared<T>`
PTR(T)	`std::shared_ptr<T>`
CAST(T)	`std::dynamic_pointer_cast<T>`
THIS	`shared_from_this()`
ENABLE_THIS(T)	`public std::enable_shared_from_this<T>`

3. Function: `Parse()`

> `#include "parse.hpp"`

- **`PTR(Expr) parse(std::istream &in)`**
  - Parse the user input and give back an expression.
  - Parameters:
    - `in` the input stream
  - Return:
    - `PTR(Expr)` an expression parsed from the user input
  - Example:

```
parse(NEW(std::istringstream)("_let add = _fun (x) _fun (y) x + y
_in _let addFive = add(5) _in addFive(10)"));

parse(NEW(std::istringstream)(
          "_let factrl = _fun(factrl)"
          "                _fun(x)"
          "                  _if x == 1"
          "                  _then 1"
          "                  _else x * factrl(factrl)(x + -1)"
          "_in factrl(factrl)(5)"));

parse(NEW(std::istringstream)(
          "_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)"));
```

4. Class: `Expr`

> `#include "expr.hpp"`

Represent an expression.

- **`bool equals(PTR(Expr) e)`**
    - Check if two expressions are the same.
    - Parameters:
        - `e` another expression to compare with
    - Return:
        - `bool` true if two expressions are equal, false otherwise.
    - Example:

```
NEW(NumExpr)(1))->equals(NEW(NumExpr)(1);
NEW(VarExpr)("hello"))->equals(NEW(VarExpr)("hello");
```

- **`PTR(Val) interp(PTR(Env) env)`**
    - Interpret the expression.
    - Parameters:
        - `env` current environment
    - Return:
        - `PTR(Val)` an interpreted value result
    - Example:

```
parse(NEW(std::istringstream)("_let add = _fun (x) _fun (y) x + y
_in _let addFive = add(5) _in addFive(10)"))-
>interp(Env::emptyenv);

parse(NEW(std::istringstream)(
                 "_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)"))->interp(Env::emptyenv);

parse(NEW(std::istringstream)(
                 "_let factrl = _fun(factrl)"
                 "                 _fun(x)"
                 "                   _if x == 1"
                 "                   _then 1"
                 "                   _else x * factrl(factrl)(x +
-1)"
                 "_in factrl(factrl)(5)"))-
>interp(Env::emptyenv);
```

- `void step_interp()`
  - Interpret using explicit continuation, ie. interpret with steps.
  - This function is called by class `Step` with `Step::interp_by_steps(PTR(Expr) e)` function.
  - Parameters:
    - `void`
  - Return:
    - `void`
  - Example:

```
Step::interp_by_steps(parse(NEW(std::istringstream)("_let add =
_fun (x) _fun (y) x + y _in _let addFive = add(5) _in
addFive(10)")));

Step::interp_by_steps(parse(NEW(std::istringstream)(
                 "_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)")));

Step::interp_by_steps(parse(NEW(std::istringstream)(
                 "_let factrl = _fun(factrl)"
                 "                 _fun(x)"
                 "                   _if x == 1"
                 "                   _then 1"
                 "                   _else x * factrl(factrl)(x +
-1)"
                 "_in factrl(factrl)(5)")));
```

- `PTR(Expr) optimize()`
  - Get the optimized expression
  - Parameters:
    - `void`
  - Return:
    - `PTR(Expr)` a new expression which is optimized

- Example:

```
parse(NEW(std::istringstream)("_let add = _fun (x) _fun (y) x + y
_in _let addFive = add(5) _in addFive(10)"))->optimize();

parse(NEW(std::istringstream)(
                 "_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)"))->optimize();

parse(NEW(std::istringstream)(
                 "_let factrl = _fun(factrl)"
                 "                _fun(x)"
                 "                  _if x == 1"
                 "                  _then 1"
                 "                  _else x * factrl(factrl)(x +
-1)"
                 "_in factrl(factrl)(5)"))->optimize();
```

5. Class: `Val`

> `#include "value.hpp"`

Represent the value of the interpreted result.

- **`bool equals(PTR(Val) other_val)`**
  - Check if two values are the same.
  - Parameters:
    - `other_val` another value to compare with
  - Return:
    - `bool` true if two values are equal, false otherwise.
  - Example:

```
NEW(NumVal)(5))->equals(NEW(NumVal)(5);
NEW(BoolVal)(true)->equals(NEW(BoolVal)(true);
```

- **`bool is_ture()`**
  - Check if the value can be interpreted as a boolean true value.
  - Parameters:
    - `void`

  - Return:
    - `bool` true if the value is a boolean value and true, false otherwise.
  - Example:

    ```
    NEW(NumVal)(5)->is_true();
    NEW(BoolVal)(true)->is_true();
    ```

- **`PTR(Expr) to_expr()`**
  - Build an expression according to current value and data type.
  - Parameters:
    - `void`
  - Return:
    - `PTR(Expr)` a new expression with corresponding value and data type.
  - Example:

    ```
    NEW(NumVal)(5)->to_expr();
    NEW(BoolVal)(true)->to_expr();
    ```

- **`std::string to_string()`**
  - Build a string representation according to current value and data type.
  - Parameters:
    - `void`
  - Return:
    - `PTR(Expr)` a new expression with corresponding value and data type.
  - Example:

    ```
    NEW(NumVal)(5)->to_string();
    NEW(BoolVal)(true)->to_string();
    ```

- **`PTR(Val) call(PTR(Val) actual_arg)`**
  - For a function value, call the function with the actual argument.
  - Parameters:
    - `PTR(Val) actual_arg` the actual argument to the function.
  - Return:
    - `PTR(Val)` the return value of the function execution.
  - Example:

    ```
    parse(NEW(std::istringstream)("((_fun(x) _fun(y) x * x + y * y)
    (2))(3)"))->interp(Env::emptyenv);

    parse(NEW(std::istringstream)(
    ```

```
                 "_let add = _fun(x)"
                 "              _fun(y)"
                 "                x + y"
                 "_in _let addFive = add(5)"
                 "_in addFive(10)"))
        ->interp(Env::emptyenv);
```

- **`void call_step(PTR(Val) actual_arg_val, PTR(Cont) rest)`**
    - Explicit continuation version of call. For a function value, call the function with the actual argument.
    - Parameters:
        - `PTR(Val) actual_arg_val` the actual argument to the function.
        - `PTR(Cont) rest` the next step.
    - Return:
        - `void`
    - Example:

```
Step::interp_by_steps(parse(NEW(std::istringstream)("((_fun(x)
_fun(y) x * x + y * y)(2))(3)")));

Step::interp_by_steps(parse(NEW(std::istringstream)(
                 "_let add = _fun(x)"
                 "              _fun(y)"
                 "                x + y"
                 "_in _let addFive = add(5)"
                 "_in addFive(10)")));
```

6. Class: `Env`

> `#include "env.hpp"`

Set the environment to interpret local variable.

- Property: **`static PTR(Env) emptyenv`**
    - Represent the empty environment.

- **`PTR(Val) lookup(std::string file_name)`**
    - Recursively look up for the value of a variable.
    - Parameters:
        - `PTR(Val) actual_arg_val` the actual argument to the function.
        - `PTR(Cont) rest` the next step.
    - Return:
        - `void`

7. Class: `Cont`

> `#include "cont.hpp"`

Provide explicit continuation for all kinds of expressions.

- Property: **`PTR(Cont) done`**
    - Represent no more continue step.
- **`void step_continue()`**
    - Set global variables to next step
    - Parameters:
        - `void`
    - Return:
        - `void`

8. Class: `Step`

> `#include "step.hpp"`

Execute the expression step by step, until a done continuation is hit. The Step class use global variables so those variables should be saved for the interpreter.

- enum: **`mode_t`**
    - Two mode for execute steps:
        - `interp_mode`
        - `continue_mode`
- Property: **`Step::mode_t Step::mode`**
    - Represent step mode with enum type: `mode_t`.
    - Mode indicates whether the next step is to start interpreting an expression or start delivering a value to a continuation.
- Property: **`PTR(Cont) Step::cont;`**
    - The continuation to receive a value, meaningful only when `mode` is `continue_mode`.
- Property: **`PTR(Expr) Step::expr`**
    - The expression to interpret, meaningful only when `mode` is `interp_mode`.
- Property: **`PTR(Env) Step::env`**
    - The environment of current step.
- Property: **`PTR(Val) Step::val`**
    - The value to be delivered to the continuation, meaningful only when `mode` is `continue_mode`.
- **`PTR(Val) interp_by_steps(PTR(Expr) e)`**

- Function to interpret an expression by stepping. The function should only be called once to start an expression's interpretation. It must not be called by `step_interp` or `step_continue` (i.e., it must not be called recursively, since the whole point is to avoid rcursive calls at the C++ level).
- Parameters:
  - `PTR(Expr) e` an expression need to interpret.
- Return:
  - `PTR(Val)` interpreted value of the expression.
- Example:

```
Step::interp_by_steps(parse(NEW(std::istringstream)("((_fun(x)
_fun(y) x * x + y * y)(2))(3)")));

Step::interp_by_steps(parse(NEW(std::istringstream)(
                 "_let add = _fun(x)"
                 "              _fun(y)"
                 "                x + y"
                 "_in _let addFive = add(5)"
                 "_in addFive(10)")));
    Step::interp_by_steps(parse(NEW(std::istringstream)("_let add
= _fun (x) _fun (y) x + y _in _let addFive = add(5) _in
addFive(10)")));

Step::interp_by_steps(parse(NEW(std::istringstream)(
                 "_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)")));

Step::interp_by_steps(parse(NEW(std::istringstream)(
                 "_let factrl = _fun(factrl)"
                 "                 _fun(x)"
                 "                   _if x == 1"
                 "                   _then 1"
                 "                   _else x * factrl(factrl)(x +
-1)"
                 "_in factrl(factrl)(5)")));
```