MSDScript

MSDScript Documentation

0 Description

MSDScript is a programming language supporting mathematical operations and programming functions as well as if determinations with simple syntax in macOS system. As an interpreter, MSDScript allows local variables, optimizer and will be memory leak free after execution. Generally, users will run MSDScript in terminal with command line.

---

1. Getting Started

1.1 Setting Up

1. Create one empty directory.
2. Download source files into the directory.
3. Navigate to the directory and run `clang++ -std=c++17 -o filename *.cpp` in terminal, users can type in the filename by their preference and terminal will generate an executable file with this filename.
4. Run this executable file by run `./filename` in terminal.
5. Two more modes by run `./filename --step` or `./filename --opt` in terminal for different kinds of operations, we will talk about them later.
6. When user has typed in all content, press control+D to exit and let MSDScript run the input content. MSDScript will generate the result before exiting.

1.2 Building Library

1. Navigate to the source code directory.
2. Run
   ```
   clang++ -std=c++17 -c cont.cpp expr.cpp step.cpp Env.cpp exec.cpp parse.cpp value.cpp
   ```
   in the terminal and user will get a bunch of `.o` files generated by the `.cpp` files.
3. Run `ar rsv filename.a *.o` in the terminal, user can type in the filename by their preference. This command will generate a `filename.a` file.

4. Set user's project file path to the `Header.hpp` file and `filename.a` file or just simply let those two files in the same folder with user's `main.cpp` file.
5. Now the only thing user need to do to use MSDScript library is include `Header.hpp` file in user's C++ code.

```
#include "Header.hpp"
```

6. Use `clang++ -o file main.cpp filename.a` to create an executable file in terminal. The `file` in command is name of this executable file and user can type in any name by their preference.

---

2. User Manuals

2.1 Introduction

MSDScript has three different modes such as interpreter and step as well as optimizer.

- **Interpreter mode:** If user just run `./filename`, it will enter the interpreter mode which evaluates an MSDScript expression and returns the value. If there is any free variable, an error will be thrown.
- **Step mode:** If user run `./filename --step`, it will enter the step mode which uses heap to interpreter while it uses stack to interpreter in the interpreter mode. MSDScript will generate the same result as in interpreter mode.
- **Optimizer mode:** If user run `./filename --opt`, it will enter the optimizer mode. In this mode, MSDScript will optimize the input content instead of simply running it. If there is not any free variable, it will run it as another two modes. But if there is, it will run those runnable parts and leave the free variable unchanged.

2.2 Basic Expression

1. Numbers Expression, including both positive and negative number and 0.

```
7
-7
```

2. Variables Expression, which should be defined to allow only a-z and A-Z.

```
x
X
```

3. Boolean Expression, including two logic elements: true and false.

```
_true
_false
```

4. Add Expression, which is add two number expressions and return an expression.

```
1+2
3
```

But it is invalid to add a number expression with a boolean expression.

```
1+_true
//std::runtime_error:
 input is not a number
```

Also when there is one undefined variable expression, result depends on the mode.

```
//On Interpreter mode or Step mode
x+1
```

```
//std::runtime_error:
 free variable: x

//On Optimizer mode
x+1
(x + 1)
```

5. Multiply Expression, which is multiply two number expressions and return an expression.

```
2*3
6
```

But it is invalid to multiply a number expression with a boolean expression.

```
2+_false
//std::runtime_error:
 input is not a number
```

Also when there is one undefined variable expression, result depends on the mode.

```
//On Interpreter mode or Step mode
x*2
//std::runtime_error:
 free variable: x

//On Optimizer mode
x*2
(x * 2)
```

6. Equal Expression, which is compare right hand side and left hand side of `==` operation.

```
right_hand_side == left_hand_side
```

It will return a boolean expression as result.

```
1==1
_true

1==2
_false
```

Compare two number expressions or two boolean expressions are both valid.

```
_true==_true
_true

_true==_false
_false
```

Also when there is one undefined variable expression, result depends on the mode.

```
//On Interpreter mode or Step mode
x==2
//std::runtime_error:
 free variable: x

//On Optimizer mode
x==2
(x == 2)
```

2.3 Advanced Expression

1. Let Expression, which followed by `_in` syntax allows user to define a value for variable expression.

```
//Valid syntax
_let var_expr = right_hand_side _in body
```

The `right_hand_side` and `body` could be expressions. It means if there is any defined variable appears `_in` the `body` , MSDScript will automatically replace it by `right_hand_side` . It will return an expression.

```
_let x=3 _in x*2
6

_let x=3 _in x==3
_true
```

Remember the interpreter mode and step mode do not allow undefined variable.

```
//On Interpreter mode or Step mode
_let x=y _in y+2
//std::runtime_error:
 free variable: y

//On Optimizer mode
_let x=y _in y+2
(_let x = y _in (y + 2))
```

Nested let expression is valid, but user need to pay more attention since all variables should be defined correctly.

```
_let x=1 _in _let y=3 _in x+y
4
```

2. If Expression, which is a conditional statement and choose one branch to be executed.

```
```

```
//Valid syntax
_if boolean_expr _then then_expr _else else_expr
```

The `boolean_expr` is for determining to choose the `_then` part or the else part. If the `boolean_expr` is `_true`, it will execute `then_expr`, else it will execute the `else_expr` when the `boolean_expr` is `_false`.

```
_if _true _then 1 _else 2
1

_if _false _then 1 _else 2
2
```

Similarly, interpreter mode and step mode do not allow undefined variables.

```
//On Interpreter mode or Step mode
_if x == 1 _then 1+2 _else 2+3
//std::runtime_error:
 free variable: x
```

In optimizer mode, if there is any undefined variable in `boolean_expr` part, MSDScript will just optimize and return the whole input content. If the undefined variable is in `then_expr` or `else_expr` part, MSDScript will first choose which branch to execute, then optimize the chosen expression.

```
//On Optimizer mode
_if x == 1 _then 1+2 _else 2+3
(_if (x == 1) _then 3 _else 5)

_if _true _then x+1 _else y
(x + 1)

_if _false _then x _else y
y
```

3. Fun Expression, which allows user to define function. The syntax is

```
//Valid syntax
_fun(var_expr) body
```

The `var_expr` is an expression which will be executed in `body`, and result of this function expression will directly relate to this `var_expr`. But simply define a function can do nothing, it will return a `[FUNCTION]`. User need to define an `expr` after the body to tell fun expression what is the expression argument of `var_expr`, which is the CallFun Expression. We will talk about this below.

```
_fun(x) x+10
[FUNCTION]

(_fun(x) x+1)(1)
2
```

4. CallFun Expression, which allows user to call fun expression by passing an argument expression.

```
//Valid syntax
(_fun(var_expr) body)(argu_expr)
```

The `argu_expr` is an expression and it will substitute the `var_expr` in `body`.

```
(_fun(x) x*x)(3)
9
```

Similarly, interpreter mode and step mode do not allow undefined variables.

```
```

```
//On Interpreter mode or Step mode
(_fun(x) x*x)(y)
//std::runtime_error:
 free variable: y
```

In optimizer mode, MSDScript will optimize whole content if there is any undefined variable.

```
//On Optimizer mode
(_fun(x) x*x)(y)
(_fun(x) (x + 1))(y)

(_fun(x) x+y)(2)
(_fun(x) (x + y))(2)
```

Also, user can define a name for any anonymous fun expression.

```
_let f=_fun(x) x*x _in f(3)
9
```

What is more, user cannot do any loop in MSDScript. But it is valid for user to call fun expression recursively which can do some simple loop works instead.

```
//call countdown fun expression 100 times
_let countdown = _fun (countdown) _fun (n)
    _if n == 0 _then 0
    _else countdown (countdown) (n + -1)
_in countdown (countdown) (100)
```

Since any recursively calling has risks to cause stack overflow, user is recommended to do recursive work under step mode, especially when the recursive work is huge.

2.3 On Stdin/Exit Codes

If matches the grammar, then

- Exit 2 if interpreter and errors
- Exit 0 if interpreter without errors
- Exit 0 always for optimizer

If not matches the grammar, means failed to parse, then exit with code 1.

2.4 On Stdout

- For interpreter if no error: number (maybe "-" and then a nonempty sequence of digits) or boolean ("_true" or "_false") value, or "[function]" for function results –– always followed by a newline, no extra spaces anywhere. The number should be followed by the a new line character '\n'.
- For interpret, with 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).

2.5 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.

---

3 API Documentation

3.1 Modes

1. Interpreter Mode

Valid syntax is `std::string interp(std::istream &in)` . MSDScript will interpreter the input stream as an expression and return a string as output.

```
int main(int argc, char *argv[]) {
    std::string s = interp(std::cin);
    std::cout << s << std::endl;
    return 0;
}
```

2. Step Mode

Valid syntax is `std::string step_interp(std::istream &in)`. MSDScript will interpreter the input stream as an expression and return a string as output in step mode.

```
int main(int argc, char *argv[]) {
    std::string s = step_interp(std::cin);
    std::cout << s << std::endl;
    return 0;
}
```

3. Optimizer Mode

Valid syntax is `std::string optimize(std::istream &in)`. MSDScript will interpreter the input stream as an expression and return a string as output in optimizer mode.

```
int main(int argc, char *argv[]) {
    std::string s = optimize(std::cin);
    std::cout << s << std::endl;
    return 0;
}
```

3.2 Parse Class

Valid syntax is `parse(std::istream &in)`. Parse will take an input stream and start the parsing process to generate a interrelated programming expression chain. The expression chain is of type pointer and can return string in different modes. If the input stream cannot be parsed, it will throw an error.

```
PTR(Expr) parse(std::istream &in) {
    PTR(Expr) e = parse_expr(in);
    //Allow white space
    char c = peek_after_space(in);
    if(!in.eof())
        throw std::runtime_error((std::string)"expected end of file at " +
    return e;
}
```

3.3 Expr Class

1. Equals

Valid syntax is `virtual bool equals(PTR(Expr) e)` . Equals will check expressions on the right hand side and left hand side to see if they are equal to each other. It will return a boolean.

2. Interp

Valid syntax is `virtual PTR(Val) interp(PTR(ENV) env)` . Interp will interpreter the input expression and return a value object.

3. Step_interp

Valid syntax is `virtual void step_interp()` . Step_interp will interpreter the input expression on step class with `Step::interp_by_steps(PTR(Expr) e)` function.

4. Optimizer

Valid syntax is `virtual PTR(Expr) optimize()` . Optimize will interpreter the input expression on optimizer mode and return a new optimized expression.

3.4 Value Class

1. Equals

Valid syntax is `virtual bool equals(PTR(Val) val)`. Equals will check values on the right hand side and left hand side to see if they are equal to each other. It will return a boolean.

2. To_expr

Valid syntax is `virtual PTR(Expr) to_expr`. To_expr will take current value as input and return a new expression.

3. To_string

Valid syntax is `virtual std::string to_string()`. To_string will convert current value to a string and return it.

4. Call

Valid syntax is `virtual PTR(Val) call(PTR(Val) actrual_arg)`. Call will call the function with the input argument and return a value.

5. Call_step

Valid syntax is `virtual void call_step(PTR(Val) actual_arg_val, PTR(Cont) rest)`. Call_step will set the current argument by `actual_arg_val` and take `rest` as continuation for further function calls.

3.5 Env Class

1. Lookup

Valid syntax is `PTR(Val) lookup(std::string find_name)`. Lookup will find the input string in current value recursively.

2. Equals

Valid syntax is bool equals(PTR(Env) env). Equals will check environments on the right hand side and left hand side to see if they are equal to each other. It will return a boolean.

3.6 Cont Class

1. Step_continue

Valid syntax is `virtual void step_continue()` . Step_continue will set global variables to next step.

3.7 Step Class

1. Interp_by_steps

Valid syntax is `static PTR(Val) interp_by_steps(PTR(Expr) e)` . Interp_by_steps will interpreter an expression under the step mode and return an interpreted value.