Description:

This project is to implement an interpreter for a stack-based programming language called UofL, and it relies on postfix notation for calculation. Stack data structure is heavily used. For the programming part, it takes 10 hours to finish the prototype and extra 2 hours to polish the bugs.

Date Structure:

The interpreter uses several global variables as data structure:

```
#|
Global memories:
 Global stack
 Global variables
 Static functions
 Dynamic functions
 Helper for command lines
 Unique temporary
 Stack for static scoping
 String printing status
|#
```

global_stack : a running stack of the interpreter
running_vars: a running stack to record the global and visible local variables
static_funcs: a stack to record the defined static scoping functions
dynamic_funcs: a stack to record the defined dynamic functions
comd_list: a queue (or list) to memory the user-input commands
temp: a unique memory to store the value from POP
backup: an auxiliary stack to help static scoping functions
ss: an auxiliary flag used in . “MESSAGE” operation

Function structures:

The functions are built in groups:

- Given input helper: Given functions to read input from user
    - read-keyboard-as-list
    - read-keyboard-as-string
- General stack-based operations: Required commands
    - push: push a number into the running stack
    - drop: pop off and throw the top element of the stack
    - pop: pop off the top element and keep it in temporary location
    - save: push the item in temporary location into the stack again
    - dup: push a duplicated top element to the stack
    - swap: swap the top 2 elements
    - rev: reverse the stack
    - clear: free the stack

Basic math operations: including + - * / > < >= <=

Helpers for if-else and loop:

reach_else: used to reach the position of command ELSE

reach_then: used to reach the position of command THEN

reach_pool: used to reach the position of command POOL

find_index: used to find the position certain command in command list

double_loop: append the commands between LOOP & POOL to the end

Helpers to stack management

find_string_in_stack: used to find a string in a string stack

find_in_pair_stack: used as find for a map <key,list>

find_func_index: used to search a function in function stack

Helpers to define the variables

global_define: used to implement the keyword define in global scope

Helpers for function scoping

static_func: used to extract the execution commands in FUNC$ definition

static_var: used to extract the local variable definition in FUNC$ definition

static_build: used to build a FUNC$ function as (name,func,vcar) list

Helpers to process the command

read_comd: used read user-input commands to command list

print_comd: used print out the command list during test procedure

exec_comd: used to analysis and execute all the user-input commands one-by-one

Function: UofL

UofL: runner of the interpreter

Difficulties:

The main difficulty is the language of scheme itself, besides, some extra hours are spent to figure of the issues below:

1. Handling . operation

The difficulty lies in both distinguish the purpose of . (show top) and . “Message” (print message). Especially for . “Message”, it also needs to consider whether the message is a single word or a complete sentence.

2. LOOP-POOL

The difficulty is that when POOL is reached, then the operation needs to trace back to LOOP and execute it again. Finally we figure out a method which focuses on LOOP instead of POOL. The idea is that if the condition is #t, then we append the commands between LOOP-POOL again at the end of the POOL, and if the condition is #f, we simply skip to the POOL and execute the next commands.

3. Dynamic and static scoping

The dynamic and static scoping is the main focus of this project, which is very tricky.

For the static scoping, a method, as backing-up the running stack and recovery the running-stack as the static function finishes, is applied. Thus all the local variables would be freed from the stack.
For the dynamic scoping, local variables are put into the running stack directly, and the calling order would literally put the correct variables on the top of running-stack and trace down for binding.

Limitation of the interpreter:

The interpreter has been tested thoroughly, however limitation still exists.
The main issue is that certain commands must be input in one line instead of separately.
They are:

    IF ** ELSE ** THEN
    LOOP *** POOL
    FUNC$ *** CNUF
    FUNC% *** CNUF

This is mainly because we choose to treat both the starting tag and the ending tag as a complete command. Thus, if the user input the 2 tags separately, the interpreter would treat the operations between as undefined command.

Testing Procedure:

1. Testing the basic mathematical operations and stack operations

```
Welcome to DrRacket, version 6.1.1 [3m].
Language: scheme; memory limit: 128 MB.
UofL> 6 2 1 + . / .
3
2
UofL> STACK
(2)
UofL> DROP
UofL> 4 DUP 17 <=
UofL> . POP
#t
UofL> 15 SAVE STACK
(#t 15 4)
UofL> REV 4 / STACK
(1 15 #t)
UofL> DUP * .
1
```

2. Testing the different behavior of operator

```
Welcome to DrRacket, version 6.1.1 [3m].
Language: scheme; memory limit: 128 MB.
UofL> 5 9 <= . . "TEST STRING" STACK
#t
"TEST STRING"
(#t)
```

3. Testing definition of global variable

```
Welcome to DrRacket, version 6.1.1 [3m].
Language: scheme; memory limit: 128 MB.
UofL> define a 5
UofL> define b 16
UofL> a b c STACK
Command Undefined
(16 5)
```

Variable c is not defined, so the interpreter warns about this error.

4. Testing conditional IF-ELSE-THEN

```
Welcome to DrRacket, version 6.1.1 [3m].
Language: scheme; memory limit: 128 MB.
UofL> 1 0 >
UofL> STACK
(#t)
UofL> IF 1 ELSE 2 THEN .
1
UofL> 2 >
UofL> STACK
(#f)
UofL> IF 1 1 + ELSE 2 2 + THEN .
4
```

5. Testing loop by calculating factorial

```
Welcome to DrRacket, version 6.1.1 [3m].
Language: scheme; memory limit: 128 MB.
UofL> . "Factorial commands"
"Factorial commands"
UofL> 1 1 DUP 10 <=
UofL> LOOP DROP DUP POP * SAVE 1 + DUP 10 <= POOL
UofL> DROP DROP .
3628800
```

The UofL commands for factorial of 10 are:

1 1 DUP 10 <=

LOOP DROP DUP POP * SAVE 1 + DUP 10 <= POOL

DROP DROP .

6. Testing for simple function behavior

```
Welcome to DrRacket, version 6.1.1 [3m].
Language: scheme; memory limit: 128 MB.
UofL> FUNC$ Square DUP * CNUF
UofL> 5 Square .
25
UofL> FUNC$ Add1 ( define a 1 ) a + CNUF
UofL> Add1 .
26
```

7. Testing for static scoping

```
Welcome to DrRacket, version 6.1.1 [3m].
Language: scheme; memory limit: 128 MB.
UofL> . "Global a" define a 5 a .
"Global a"
5
UofL> FUNC$ Adda ( define a 7 ) a . "Local a" . + CNUF
UofL> 10 Adda .
"Local a"
7
17
UofL> . "Global a" a .
"Global a"
5
```

8. Testing for dynamic scoping

Testing commands are:

. "Define a in Func1 and b in Func2"

FUNC% Func1 (define a 10) . "a in Func 1" a . CNUF

FUNC% Func2 (define b 5) . "b in Func 2" b . CNUF

. "Define Func3 which uses a and b by calling order"

FUNC% Func3 . "a and b are dynamically binded in Func3" a b * . CNUF

. "Calling Func1, Func2, Func3 and Func3 calculates a*b"

Func1 Func2 Func3 .

. "Define new a in global area then call Func3"

define a 20 Func3 .

```
Language: scheme; memory limit: 128 MB.
UofL> . "Define a in Func1 and b in Func2"
"Define a in Func1 and b in Func2"
UofL> FUNC% Func1 ( define a 10 ) . "a in Func 1" a . CNUF
UofL> FUNC% Func2 ( define b 5 ) . "b in Func 2" b . CNUF
UofL> . "Define Func3 which uses a and b by calling order"
"Define Func3 which uses a and b by calling order"
UofL> FUNC% Func3 . "a and b are dynamically binded in Func3" a b * . CNUF
UofL> . "Calling Func1, Func2, Func3 and Func3 calculates a*b"
"Calling Func1, Func2, Func3 and Func3 calculates a*b"
UofL> Func1 Func2 Func3 .
"a in Func 1"
10
"b in Func 2"
5
"a and b are dynamically binded in Func3"
50
50
UofL> . "Define new a in global area then call Func3"
"Define new a in global area then call Func3"
UofL> define a 20 Func3
"a and b are dynamically binded in Func3"
100
```