A. General description

Implement an interpreter for a *stack-based* programming language called Lang which is defined below. The programming language chosen for your implementation is *Dr. Scheme*.

Programming language Lang heavily uses stack data structure and reverse Polish notation, which is also called postfix notation because the operator is placed after its operands, as opposed to the more common infix notation where the operator is placed between its operands.

The rationale for postfix notation is that it is closer to the machine language a computer will eventually use, and should therefore be faster to execute. For example, one could get the result of expression *25 * 10 + 50* using this notation. Suppose that you have a stack for you to use. Then you want to do the following.

```
25 10 * 50 +
```

This notation first pushes the numbers 25 and 10 on the stack. The operator * pops the top two numbers from the stack, multiplies them (25 * 10), and pushes the result 250 on the stack. After this, the number 50 is placed on the stack. The operator + pops off the top two numbers, i.e., 50 and the previous product, 250, adds them up, and pushes the sum 300 on the stack.

From this, you find that stack and the stack operations have a close relationship to the reverse Polish notation and calculate it in a natural way.

As another example, to calculate ((12 + 13) * 9 + (7 * 8)) / 3, we would have the following equivalence in the reverse Polish notation. Please try it using stack as we did above and see what happens.

```
12 13 + 9 * 7 8 * + 3 /
```

B. A stack-based programming language Lang

(1) Programming language Lang's interpreter always displays a prompt Lang>, at which you can type in a command as defined below. The command will be interpreted by the interpreter. For example, you can instruct Lang to do the some calculation as the above example.

```
Lang> 25 10 * 50 +
```

(2) You need to implement the arithmetic operations, i.e., +, -, *, and /, in the same meaning as their original intended arithmetic operations. As an example,

```
Lang> 25 10 *
```

25 is pushed into the stack. Next so is 10. Then * is applied to the top two elements by first popping them off as 10 (the top) and 25 (the second to the top), where the 25 is used as the first operand and 10 is the second, i.e., 25 * 10, and then calculates them. The result is pushed into the stack. All the operations, including +, -, *, /, and later <, >, <=, >=, follow this convention. However, <, >, <=, and >= result in Boolean values, where 1 is true and 0 is false.

Operator . is to display the top element in the stack. So after the above operation, if you type **.** at the prompt as follows,

```
Lang> .
```

```
250
```

is displayed. . message will display the message on the screen. For instance, the following Lang> . "Hello world!" displays the string Hello world! on the screen. You can assume that the only numeric data type is integer and we only have integers and strings in

Lang.

(3) Define operation

You can define a variable using the keyword define, as follows.

```
Lang> define a 10
```

Every variable must be defined and be initialized. Variable a defined this way is a global variable, in contrast to the local variables below defined in functions.

(4) Stack operations

In programming language Lang, all calculations depend on the current content of the stack. Your interpreter needs to maintain a stack internally. We therefore need to design a set of commands for moving the elements around the stack.

DROP will pop off the top element of the stack and throw it away.

POP will pop off the top element of the stack but keep it in a temporary location. (There is only just one such temporary location, which means that only the element from the last POP is saved there.)

Another way to get the top element is through

```
Lang> POP a
```

which pops the top element and put it into variable a, if you want to use it. Of course, variable a should have been defined before. Otherwise, you need to report an error message.

SAVE will push the item in the temporary location to the top of the stack.

DUP will duplicate the top element of the stack and pushes the duplicate into stack.

SWAP will swap the top two elements on the top of the stack.

REV will reverse all the elements in the stack, i.e., the top becomes the bottom while the bottom becomes the top.

STACK will print out the current content of the stack, from the top to the bottom.

CLEAR will clear and throw away all the elements in the current stack.

A push operation is implied by typing a number at Lang> prompt. For instance the following pushes 40 onto the stack,

```
Lang> 40
```

while the following pushes the value in variable a onto the stack. (Again, variable a should have been defined before.)

```
Lang> a
```

As another example, Lang> a + means that variable a is pushed onto the stack. It is then popped off and the next element is popped off. These two elements are added together and the result is returned back to the stack again.

(5) Conditionals

Lang has all the usual IF/THEN/ELSE structures that you get in any other programming language, but with a strange syntax. For example, the syntax of IF is:

```
<condition> IF

<action-if-true>

ELSE

<action-if-false>

THEN
```

Note that the condition comes before the IF, rather than after it as in most other languages. Also, THEN marks the end of an if statement, rather than the action to be carried out if the condition is true. An example of an if statement is as follows:

```
0 > IF

. "Top of stack is bigger than 0."

ELSE

. "Top of stack is not bigger than 0."

THEN
```

Suppose that the value of the top element is 10. Here, 0 is pushed at the top of the stack. Then the top two elements are popped off and compared with each other (i.e., 10 > 0, according to the above convention). The result is either true (1) or false (0) and is pushed onto the stack. If the result is true, message "Top of stack is bigger than 0." is displayed; otherwise message "Top of stack is not bigger than 0." is displayed.

An if statement without else can be described as follows.

```
<condition> IF <action-if-true> THEN
```

You need to implement the relational operations >, >=, <, and <=, whose returned values are either true (1) or false (0).

(6) Loops

Loops in Lang are Boolean controlled. A loop starts with a conditional and followed by the loop body. The conditional is checked for each iteration of the loop.

As an example, the following loop does 1 + 2 + 3 + 4 … + 10.

```
Lang> 0 1

Lang> DUP

Lang> 10 <= LOOP DROP DUP POP + SAVE 1 + DUP POOL

Lang> DROP DROP

Lang> .
```

As can be seen, after the required conditional, the loop body is enclosed by LOOP and POOL.

Initially, 0 is the sum while 1 is the counter.

In the above example, in the loop body, DROP drops the conditional result from the stack. DUP duplicates the counter. POP pops the counter and saves it in a temporary location. + takes the top two elements from the stack, i.e., the sum and the current counter, adds them, and pushes them back to the stack. SAVE saves the current counter from the temporary location and pushes it to the stack. Next, 1 + will add one to the current counter and save it back to the stack. DUP duplicates the counter and pushes it into the stack, preparing for the next iteration.

Finally the two DROPs drop the conditional result and then the counter, and . prints out the sum. Execute the loop by working around the stack and see what happens.

(7) Function definitions

There are two types of function you can define in Lang. We first show an example as how to define a function in Lang which will use static scoping. The keyword is FUNC$.

```
Lang> FUNC$ Square DUP * CNUF
```

FUNC$ marks the start of a function definition. Square is the name we have chosen to give the function. CNUF marks the end of the definition. The rest is just the sequence of actions that should be carried out by the function.

We can then use the function.

```
Lang> 5 Square
```

Here 5 is pushed into the stack. Then function Square is called, in which the top element is duplicated and push into the stack. * is then applied, which pops off the top two elements, multiply them, and pushes the result back to the stack.

```
Lang> .
```

This will display the result 25. As another example,

```
Lang> FUNC$ FLOOR5 DUP 6 < IF DROP 5 ELSE DROP 1 - THEN CNUF
```

this is a piece of code which defines a new function called FLOOR5 using the following commands: DUP simply duplicates the top number on the stack and puts the duplicate on the stack; 6 is then pushed onto the stack.; < pops the two numbers and compares them. It then pushes a true-or-false value onto the stack; IF takes a true-or-false value and chooses to execute commands immediately after it or to skip to the ELSE; DROP discards the true (1) on the stack while 5 pushes 5 on the stack; and THEN ends the conditional. If you further analyze the code, the net result is a function that performs similarly to this function written in the C/C++:

```
int floor5(int v) {
        return v < 6 ? 5 : v - 1; }
```

You can also define local variables in a function using define, as shown in the following example. Such a define should be in ( ). To define multiple variables, each one needs to be defined this way.

```
Lang> FUNC$ SWAP (define a) … CNUF
```

If a variable cannot be found locally, we use static scoping rule to find its definition.

The second type of functions is defined by the keyword FUNC%, which uses dynamic scoping, such as the following one.

```
Lang> FUNC% Calc a + … CNUF
```

This means that if there is a variable which is not defined locally, we need to follow dynamic rules to find its definition. For instance, the variable a in function Calc.

You assume that the functions defined to use static scoping can only call the functions defined to use static scoping. The same applies to the functions defined to use dynamic scoping.

Your functions only consist of commands/operations from (1), (2), (3), (4), and (5) shown above.

(C) Your interpreter

Your interpreter prints the screen prompt Lang>. A user interacts directly with your interpreter, typing sequences of strings which are then read and executed by the interpreter.

When the interpreter finds a function name, it tries to look it up in the internal *dictionary* and execute the function's code, if possible. If it fails, then it tries to print it out, followed by an error message

(e.g. "not in dictionary"), and awaits further user input.

(1) Here is an example of the interpreter at work:

```
Lang> 2 3 + Lang> .
```

5

```
Lang> 2 3 4 + * . 14

Lang> = 1 if 2 else 3 then . 3

Lang> FUNC$ square dup * CNUF (square)

Lang> 4 square . 16
```

(2) Some comments

(*) Whenever a new function is defined, your interpreter prints a list of the defined function names,

including the ones using static scoping and the ones using dynamic scoping. (*) You can create as many stacks as necessary. (*) You need separate tables to hold the defined global variables, the functions using static scoping,

and the functions using dynamic scoping, respectively.

(*) When a function calling sequence is generated, you may need to have a separate table for each function on the sequence, such that these tables can be used to find definitions of non-local variables.

(*) No recursive functions are allowed.

(*) In order to let you focus on the implementation, rather than getting involved in error checking heavily, you only need to deal with some obvious errors in user inputs. Some example could be: user tries to invoke a function name which has not been defined yet.