Programming Languages: Project #3 - Ginko interpreter

SWE3006 Programming Languages - Fall 2019

Due date: November 29 (Fri) 11:59pm

1 Objective

In this assignment, you will use `yacc` to implement an interpreter for a novel programming language *Ginko* that supports the set of statements listed below. The interpreter executes the statements one after the other in the order which they appear in the source code.

2 Grammar

The following grammar is for the programming language that your interpreter should handle.

```
program -> func_decl stmt_list
stmt_list -> stmt_list stmt
           | stmt
stmt -> assign_stmt
      | print_stmt
      | if_stmt
      | while_stmt
      | func_call
assign_stmt -> 'SET' ID 'to' expr ';'
             | 'SET' ID 'to' func_call ';'
print_stmt -> 'PRINT' expr ;
if_stmt -> 'IF' expr 'THEN' stmt_list 'ENDIF'
         | 'IF' expr 'THEN' stmt_list 'ELSE' stmt_list 'ENDIF'
while_stmt -> 'WHILE' expr 'DO' stmt_list 'ENDWHILE'
func_decl -> 'FUNC' ID '(' param_list ')' '{' fstmt_list 'RETURN' expr ';' '}' func_decl
           | 'FUNC' ID '(' param_list ')' '{' fstmt_list 'RETURN' func_call ';' '}' func_decl
           | epsilon
fstmt_list -> fstmt_list fstmt
           | fstmt
fstmt -> assign_stmt
      | print_stmt
      | if_stmt
      | while_stmt
param_list -> ID ID_list
ID_list -> ',' ID ID_list
            | epsilon
func_call -> ID '(' arg_list ')'
arg_list -> expr expr_list
expr_list -> ',' expr expr_list
       | epsilon
expr -> ( expr )
       | + expr expr
       | - expr expr
       | * expr expr
       | / expr expr
       | < expr expr
       | > expr expr
```

```
| <= expr expr
| >= expr expr
| == expr expr
| != expr expr
| ! expr
| NUMBER
| ID
```

3 Details

- Variable names consist of only alphabets, i.e., [a..zA..Z]
- Support negative integer numbers and real numbers
- Negative sign symbol is !, i.e., -100 is written as !100 in this language.
- Positive sign symbol is not supported.
- expr is always in double, i.e., / 5 2 is 2.500000 not 2
- Use %lf format for PRINT
- If an uninitialized variable is found, print `Unknown variable:  abc` and abort the program.
- If any other errors are encountered in line 12, print `Error:  12` and abort the program.
- Boolean condition expression in WHILE and IF statements treats non-zero as true.
- Gingko uses dynamic scoping. That is, the binding of a variable is determined based on calling sequence at runtime. Please check lecture slides for an example.
- According to the grammar, RETURN statement can be used only at the end of a function body.
- The scope of variables declared in IF-ELSE and WHILE block is the function, not the IF-ELSE and WHILE block. I.e., you should not create a runtime stack frame for IF-ELSE and WHILE block.
- A function can not call multiple functions, and a function can be called only with RETURN statement at the end, as in tail recursion.
- Function parameters are passed by value.
- Note that there can be some unclear requirements in this project since this is the first time that I designed this language. If you find any error or if you need clarification, please do not hesitate to post your questions or feedback in Piazza. Also, please make sure you check our class web page for any updates on this project.

4 Inputs and Outputs

Your program will read Ginko source code from a file. I.e., the first argument of your command is the input file. The output must be printed to `stdout` as follows.

```
$ cat input.gk
   FUNC max(a, b) {
        IF ( > a b ) THEN
            SET x to a ;
        ELSE
            SET x to b ;
        ENDIF
        RETURN x;
   }
```

```
   SET p to 10;
   SET q to 20;
   SET r to max(p,q);
   PRINT r;
   WHILE < p q DO
       SET p to * p 2;
   ENDWHILE
   SET r to max(p,q);
   PRINT r;
$ ./a.out input.gk
   20.000000
   20.000000
$
```

Another example:

```
$ cat input2.gk
FUNC g(a, b) {
    SET r to * a b;
    RETURN r;
}
FUNC f(p, q) {
    SET x to + p q;
    SET y to - p q;
    RETURN g(x, y);
}
SET x to 10;
SET y to f(3, 2);
PRINT x;
PRINT y;
$ ./a.out input2.gk
10.000000
5.000000
```

More example inputs will be announced later to clarify requirements if necessary.

5 How to Submit

You will need to submit multiple files that you write for this project. Therefore, you should create a sub-directory, for example proj3, in your home, and then run pl_submit command as follows.

```
$ cd /home/2012345678/proj3
$ pl_submit proj3 ./
```

Note that you can submit multiple times. But only the last submission will be graded. Using the following command, you can check whether your file has been correctly submitted.

```
$ pl_check_submission proj3
```

Note that these commands only work in swin, swui, swye, swji machines. If you implemented your codes in your desktop, you must upload the file to these machines before you submit. Otherwise, pl_submit command will fail to read your implementation.

For any questions, please post them in Piazza so that we can share your questions and answers with other students and TAs. Please feel free to raise any issues and post any questions. Also, if you can answer other students' questions, you are welcome to do so. You will get some credits for posting questions and answering other students' questions.