CS420: Compiler Design

Fall, 2017

Term Project #2: Interpreter Implementation

(Due date: Nov. 30, 2017)

- **Overview**

  The second term project submission should be an implementation of a semi-C language interpreter.

- **Interpreter Implementation**

  Based on the internal data structure specified in the first submission, students have to build their own interpreter for semi-C programs. The interpreter should parse input source program properly, translate input source code into some efficient intermediate representation and execute the stored program.

  The interpreter should have three interactive prompt commands as explained in the previous notice.

  - ✓ **next** command: This command executes a single or multiple line(s) of the source code. For example, "next" just executes current line of source code, and "next 10" will execute 10 lines including current line.
  - ✓ **print** command: This command prints the value contained in a variable at the moment. For example, if an integer variable a contains value 10, then "print a" will print "10"
  - ✓ **trace** command: This command shows the history of a variable from beginning to the moment.

  And the expected result is as follow.

Example input code	Interpreter input commands and results

1 int avg(int count, int *value) {
2 int i, total;
3 int sum = 0;
4 for (i = 1; i < count; i++) {
5 total = total + value[i];
6 }
7
8 return (total / count);
9 }
10
11 int main(void) {
12 int studentNumber, count, i, sum;
13 int mark[4];
14 float average;
15
16 count = 4;
17 sum = 0;
18
19 for (i = 0; i < count; i++) {
20 mark[i] = i * 30;
21 sum = sum + mark[i];
22 average = avg(i + 1, mark);
23 if (average > 40) {
24 printf("%f\n", average);
25 }
26 }
27
28 }
	(In this example, the interpreter starts at the top of main function, line 12)
	
	>> **next**
	>> **next** 3
	>> **print** count
	N/A
	>> **next**
	>> **print** count
	4
	>> **next** 1000
	45.0000
	End of Program
	>> **trace** i
	i = N/A at line 12
	i = 0 at line 19
	i = 1 at line 19
	i = 2 at line 19
	i = 3 at line 19
	>>

- **Submit form**

  ✓ A zip file of the source codes

  HW3_StudentID_Name.zip

  ex) HW3_20173283_Kyuho_Son.zip

If plagiarism is detected, zero-score will be given.

TA will check the details of source code of each student.

If any problem or question, feel free to ask TA with E-mail or face-to-face at office hour in every Friday.

TA (Kyuho Son) : ableman@kaist.ac.kr

Requirement Specification

- **Terminology**

  - ✓ Line:

    Meaning ①: When used in indicating the position in a code, means each line separated by line feed ('₩n') in the code

    (In this meaning, the sample code is 28 lines long in total)

    Meaning ②: When used as an argument of 'next' command, means each executed line including those of all stacks of function calls

    (In this meaning, the sample code executes 81 lines in total)

    81 = 9(ln12~18, 27~28) + 6(ln19~23, 26) * 4(times) + 2(ln24~25) * 1(once) + 4(ln2~3, 7~8) * 4(times) + 3(ln4~6) * (1+2+3+4)(times)

  - ✓ Value (of a variable): Means the real semantic value of a variable according to its data type. For example, for a float type variable, its byte data should be decoded as float. For a pointer type variable, its byte data should be decoded as memory address.

  - ✓ History (of a variable): Means all its value change logs due to declaration and assignment. Even if the actual value does not change, all of the above operations should be included in the history. Each log record consists of (line (Meaning ①), value) tuple.

  - ✓ Scope (of a variable): Means the visibility of a variable. For a variable, scope can be one of 'Invisible' or a distinct item in the symbol table. Its visibility is equivalent to the semantic visibility on the line in the code where 'print' or 'trace' command is called while interpreting.

- **Specification**

Non-functional		
Syntax error handling	When meets syntax error while interpreting a code, should stop interpreting and print "*Syntax error : line x*" for the line x (meaning ①) of the syntax.	
Run-time error handling	[OPTIONAL] When meets run-time error while interpreting a code, should stop interpreting and print "*Run-time error : line x*" for the line x (meaning ①) of the syntax.	
Register allocation	[OPTIONAL] The interpreter may do register allocation for every variable declaration.	
Functional		
Interpretation	The interpreter should be able to interpret C++ code with the equivalent to feature scope of the sample code. The interpreter should print "*End of Program*" when meets EOF. [OPTIONAL] The interpreter may contain other C++ features than those of the sample code.	
printf function	When meets 'printf()' function while interpreting a code, interpreter should print appropriate output to CLI according to the argument of the function, the symbol table and the values of the variables.	
next command	Interpreter should accept the command below via CLI. When used with no argument, should be equivalent to 'next 1'. When used with a natural number argument, should proceed lines(meaning ②) as the count of the number When other cases of argument, should print "*Incorrect command usage : try 'next [lines]'*"	
print command	Interpreter should accept the command below via CLI. When used with an argument (variable naming typing	N/A: not assigned

	rule satisfied), should find out its scope. If the scope is visible, should print the value of the item in the symbol table. If not assigned yet, should print "*N/A*". If the scope is 'Invisible' or not in the symbol table, should print "*Invisible variable".* If the argument is invalid typing, should print "*Invalid typing of the variable name*"	
trace command	Interpreter should accept the command below via CLI. When used with an argument (variable naming typing rule satisfied), should find out its scope. If the scope is visible, should print the history of the item in the symbol table. If not assigned yet, should print "*N/A*". Each log in history (line x, value) (meaning ①) of the variable should be printed as "*variable = value at line x*". If the scope is 'Invisible' or not in the symbol table, should print "*Invisible variable"* If the argument is invalid typing, should print "*Invalid typing of the variable name*"	

Optional items are not mandatory for the project. They would not have that major effect on grading, but if there are many people who gets the same scores, we can give additional scores based on satisfaction on the items. If your program satisfies the optional items, we recommend you to let us know that with *"readme.txt"* file.