CS420: Compiler Design

Fall, 2019

Term Project : Interpreter Implementation

- **Interpreter Implementation**

  In this project, the students are encouraged to build a mini-C interpreter based on the given mini-c language production rule. C++, Java and Python are allowed to use in implementation. Using Lex and Yacc is highly recommended.

  1. **Basic structure as an interpreter**: The implemented interpreter should generate a flow graph and symbol table that reflect input source code. The flow graph contains the instructions of source code and represents the control flow of each instructions. The symbol table holds variables and their type and scope.

  2. **Command functions**: The interpreter should provide interactive command prompt like Python or some other interpret languages. The implemented interpreter is required to include three basic commands:

     - **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 the beginning to the moment.

- **Expected Result for the interpreter implementation**

  The implemented interpreter is expected to operate as an example below.

Example input code	Interpreter input commands and results

1 int avg(int count, int *value)
2 {
3   int i, total;
4   total = 0;
5   for(i = 0; i < count; i++)
6   {
7   total = total + value[i];
8   }
9
10   return (total / count);
11 }
12
13 int main(void)
14 {
15   int student_number, count, i, sum;
16   int mark[4];
17   float average;
18
19   count = 4;
20   sum = 0;
21
22   for(i = 0; i < count; i++)
23   {
24   mark[i] = i * 30;
25   sum = sum + mark[i];
26   average = avg(i + 1, mark);
27   if(average > 40)
28   {
29   printf("%f\n", average);
30   }
31   }
32 }
	(In this example, the interpreter starts at the top of main function, line 14)
	
	>> **next**
	>> **next** 4
	>> **print** count
	 N/A
	>> **next**
	>> **print** count
	 4
	>> **next** 1000
	 45.0000
	 End of program
	>> **trace** i
	 i = N/A at line 15
	 i = 0 at line 22
	 i = 1 at line 22
	 i = 2 at line 22
	 i = 3 at line 22
	 i = 4 at line 22
	>>

- **Good-to-know details**

1. Implicit type casting between int ↔ float must be implemented.
2. There will be no explicit type casting in the grading code.
3. Type error detection is out of scope in this project.
4. All variable declarations will be done before all assignments. There will be no statement that declaration and assignment occur together in the grading code, except for the case of re-declarations that can be occur at the start of a new scope section made by bracket '{', '}' in the same function.
5. Pointers will be used only for referring the correspondent values of memory spaces and printing the address with print command. There will be no calculation on memory address.
6. If you implement in C language, handling pointers will be easy because the language exposures physical memory space. However, there can be some problems in Java and Python, not like that. In these cases, you should implement a virtual structure to support referring by index. When print command called with no indexing or referring, the printed address will show different values depending on environment and circumstance. Reality of memory address will not criteria for grading, so just print an arbitrary value that is constant by each variable.
7. printf function in standard C language specification shows somewhat complicated operation. This can be another major problem if you implement in other languages than C, so a constraint is added in usage of the function. printf function will only be used among the three cases below.

Format	Example
printf(const char*);	printf("hello world");
printf("%d₩n", int);	printf("%d₩n", count);
printf("%f₩n", float);	printf("%f₩n", average);

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 32 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 105 lines in total)

    for the main() function : 41 lines = 9(ln14~21, 32) + 1(ln22) * 5(times) + 6(ln23~27, 31) * 4(times) + 3(ln28~30) * 1(once)

    for the avg() function : 64 lines = 5(ln2~4, 9~10) * 4(times) + 1(ln5) * (2+3+4+5)(times) + 3(ln6~8) * (1+2+3+4)(times)

    ※ When the file pointer of the interpreter is indicating a certain line, the being indicated execution line is not yet executed, but waiting for being executed.

  - ✓ 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 must be decoded as float. For a pointer type variable, its byte data must 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	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.	
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. Maximum code length : 1024 characters per line, 1024 lines per program	
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 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: not assigned

	"*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*"	