CS420: Compiler Design

2018 Fall

Term Project: Overall Description

- **Goal of term project**

  The students will construct a so-called mini-C interpreter through this project.

- **The construction of the interpreter**

  In this project, the students are encouraged to build a mini-C interpreter based on the given mini-c language production rule.

  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 beginning to the moment.

3. **[OPTIONAL] Recursive function call**: The feature is not mandatory and can be harsh to implement. But if you could implement the feature, you will get plenty of additional points that may cover a whole unsubmitted homework.

- **Expected Result**

The implemented interpreter is expected to be operated as below example.

Example input code

```
1  int avg(int count, int *value) {
2    int i, total;
3    total = 0;
4    for (i = 0; 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 }
```

Interpreter input commands and results

(In this example, the interpreter starts at the top of main function, line 12)

```
>> next
>> next 2
>> print count
 0
>> next
>> print count
 4
>> next 1000
 End of Program
>> trace i
 i = 0 at line 19
 i = 1 at line 19
 i = 2 at line 19
 i = 3 at line 19
>>
```

Sample code for recursive function call

```
1  int sum(int num) {
2    if (num > 0) {
3      num = num + sum(num - 1);
4     }
5
6    return num;
7  }
8
9
10 int main(void) {
11   int result;
12   result = sum(10);
13   printf("%d\n", result);
14 }
15
16
```

- **Submissions**

  The students should submit up to three (intermediate) result materials during this semester. Result materials are related and accumulative; second result would be done based on the first one, and the final submission would include the first and second intermediate results.

  1. A document about internal data structure that would be used in your implementation of the interpreter. It requires only paperwork; there is no need to submit the actual implementations or source codes at this moment.
     Due: 22nd Nov.

  2. A short document about the implemented interpreter and its source codes, and a revised document for the internal structure. The implementation should include three command features that are mentioned above.
     Due: 14th Dec.

  Submission deadlines are subject to change depending on overall course schedule, and late submission is allowed, but the points will be deducted.

- **Good-to-know details**

  1. Implicit type casting between int ↔ float should be implemented.

  2. There can be some type error such as between function name and variable name. For all cases of type error like this, you can choose one implementation alternative from below.

  1) Detect type error before interpretation, and do not start interpretation printing "Type error"

  2) if type error detected while interpretation, stop interpretation printing "Run-time error"

  3. 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 re-declarations those can be occur at the start of a new scope section made by bracket ”{” in the same function.

4. For pointer, there would be no problem with implementation in languages that exposures physical memory space like C language, there can be some problems in languages not like that such as Java and Python. In this case, you should implement a virtual structure to support referring by index. Address that should be printed when print command called with no indexing or referring, would show different values depending on environment and circumstance. So that will not a criterion for grading, just print an arbitrary value.

5. 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 can be added in usage of the function. printf function will only be used among the three cases below.

1) printf(char*); → The string doesn't contain any %.
2) printf("%d₩n", int);
3) printf("%f₩n", float);