COMS W4115 Fall 2020: Homework Assignment 3

Programming Languages and Translators

Deadline: Wednesday, November 25, 2020 by 11:59 PM

Overview

This homework assignment will test your knowledge of semantic analysis, runtime environments, control flow analysis, and stack machines, all of which you have learned in class. Please submit your assignment via Gradescope by the deadline. This is an *individual* assignment, and all answers must be *typed*. However, you are allowed to hand-draw any control flow graphs/activation trees and scan and attach them in your final submission. Finally, you must adhere to the academic integrity policies of the course.

Total Points: 85
Number of Problems: 4

Problems

```
1  #include <stdio.h>
2
3  int x = 3, r = 5;
4
5  int foo(float r) {
6        return x + 2 * r;
7  }
8
9  int bar(float l, float m) {
10       return m - l - x;      8
11 }
12
13 int pltIsAwesome(int something) {
14       int a = 2, b = 1, c = 3, x;
15       for(x = 0; x < 2; x++) {
16             b -= foo(x); -2,-7
17             float something = 2.5;
18             if(x < 1) {
19                   float x = 2, r = x + 10;12
20                   something--;
21             }                  4
22             else {
23                   int x = 3;
24                   b += 1;
25             }
26             something -= r; -8,-13
27             int r = 2;
28             a += bar(foo(something), x - r);
29       }    29          -13,-23        -2,-1
30       b += 2;  -4
31       return a * b - x - something;
32 }
33
34 int main() {
35       printf("%d\n", foo(pltIsAwesome(5)));
36       return 0;
37 }
```

Table 1: Global Symbol Table for Problem 1

Name	*Kind*	*Type*	*Line Number*
`x`	variable	`int`	3
`r`	variable	`int`	3
`foo`	function	`float` → `int`	5
`bar`	function	`float, float` → `int`	9
`pltIsAwesome`	function	`int` → `int`	13
`main`	function	`void` → `int`	34

1. **(35 Points)** In Programming Assignment 3, you worked with scope-based semantic analysis to identify redefined variables. One way to accomplish this task is to use *symbol tables* to store information about variable definitions and the scopes that house them. Used by the compiler, symbol tables are map-like data structures that contain important information about variables, functions, and classes, allowing for powerful operations such as scope resolution and type checking of code. You will now have another opportunity to work with symbol tables in this problem.

   Suppose you are given the C code snippet on the previous page.

   (a) **(2 Points)** Identify the different scopes of function `pltIsAwesome` in the code snippet. Label scopes in the following form: "Scope X: [*Beginning Line Number*, *Ending Line Number*]." X is a numerical value (starting from 1) that labels your scope, *Beginning Line Number* is the line number of the first line in the scope, and *Ending Line Number* is the line number of the last line in the scope. For example, if the second scope you identified is defined from line 2 to line 4, inclusive, then you should include "Scope 2: [2, 4]" in your response. Assume that all scopes are static. Consider function `pltIsAwesome` only; don't worry about functions `main`, `foo`, or `bar`.

   (b) **(10 Points)** For each scope that you identified in part (a) for function `pltIsAwesome`, construct its symbol table. Each table must be titled, "Scope X" (where X is the scope label you created in part (a)), and have the following four columns: *Name*, *Kind*, *Type*, and *Line Number*; make sure to identify all of these attributes for each identifier you add to the symbol table. Note that the identifier *Kind* will be either "parameter" or "variable" for this part. Further, you do not need to worry about global identifiers (global variables and function names) when constructing your symbol table, but you do need to consider function parameters. We have already created the global symbol table for you in Table 1. Finally, assume that all scopes are static. *Hint: only include variables defined in a scope as part of the symbol table for that scope.*

   (c) **(3 Points)** Which definition of variable `r` does the one referenced at line 26 correspond to? Explain how you can use the symbol tables you constructed in part (b), as well as the global symbol table we have provided, to help you identify this definition.

   (d) **(5 Points)** Suppose we are still considering only function `pltIsAwesome` and are currently at line 19 in the code snippet (we have not scanned the rest of the function yet). Which identifiers (parameters and variables, including global variables if applicable) are *active* at the point of the program immediately after line 19 is scanned? Active identifiers are those that are currently being referenced at a particular point of the program. List the active identifiers as tuples of the form "(*Name*, *Kind*, *Type*, *Line Number*)." You do not need to explain the identifiers you selected.

   (e) **(5 Points)** Suppose we are instead at line 26. Which identifiers are *active* at the point of the program immediately after line 26 is scanned? List the active identifiers just as in part (d).

   (f) **(5 Points)** Now, consider all functions in the code snippet. Assuming that the entire code snippet is still *statically scoped*, what is the value printed at line 35 after executing `main`? Briefly explain your answer in one or two sentences.

   (g) **(5 Points)** Once more, consider all functions in the code snippet. Now, assume that the entire code snippet is *dynamically scoped*. What is the value printed at line 35 after executing `main`? Briefly explain your answer in one or two sentences.

2. **(15 Points)** Suppose you are given the following C code snippet:

```
1  Class PLT {
2
3          int foo(int w) {
4              if(w < 2)
5                  return hello(w, w + 1) + 3;
6              return bar(w - 2);
7          }
8
9          int bar(int x){
10             return foo(x) * foo(x - 1);
11         }
12
13         int hello(int y, int z) {
14             return y * z;
15         }
16
17         int main(){
18             bar(val);
19             return 0;
20         }
21 }
```

(a) **(5 Points)** Draw the activation tree generated by executing `main` when `val` is 3.

(b) **(10 Points)** Using a runtime stack, show the activation records generated by executing `main` when `val` is 1. You only need to include the old frame pointer, return address, returned value, and function argument(s); don't worry about temporary variables. You can expand the stack in either direction, as long as you are clear about the direction of expansion in your answer.

3. **(27 Points)** Suppose you are performing control flow analysis on the following code snippet:

```
1          entry
2          w := 2
3          x := 3 - w
4   L1:    y := x + y
5          if w < y goto L2
6          x := x - w
7          z := w * x
8          if z > 3 goto L3
9          else goto L1
10  L2:    w := x * y
11         y := z - x
12         if y < 5 goto L3
13         else goto L2
14  L3:    w := x * 2
15         x := w + z
16         return
```

(a) **(4 Points)** List the basic blocks generated from the code snippet, *using as few basic blocks as possible*. The basic blocks should be represented as a sequence of line numbers surrounded by curly braces. For example, if lines 20-24 form a basic block, then you should include {20, 21, 22, 23, 24} in your response. Similarly, if only line 36 forms a basic block, then you should include {36} in your response.

(b) **(8 Points)** Suppose instead that we treat each statement/instruction in the code as a separate node (*e.g.*, `w := 2` at line 2 and `x := 3 - w` at line 3 are part of separate nodes). Given these newly-created nodes, draw the control flow graph (CFG[1]) for this code snippet. Note once more that each node in the CFG should represent exactly one statement/instruction in the code snippet. Make sure to also label your nodes, as you will need to reference them for the next few parts. You can assume that the `return` block is the exit block. *Hint: don't forget to handle the "else" statements appropriately.*

(c) **(6 Points)** In a CFG, a node `x` *dominates* node `y` if all paths from the entry node to node `y` include node `x`. Node `x` is also referred to as the *dominator* of node `y`. List the *dominator sets* for each node `y` in your CFG from part (b), where a dominator set is the set of nodes that dominate node `y`. A dominator set should be represented as a set of node labels (which you added in part (b)) surrounded by curly braces. For example, if nodes `r`, `t`, and `x` dominate node `y`, then you should include "`y`: {`r`, `t`, `x`}" in your response.

(d) **(5 Points)** In a CFG, a node `x` *strictly dominates* node `y` if all paths from the entry node to node `y` include node `x`, and $x \neq y$. We define the *dominance frontier* of a node `x` as the set of all nodes `y` in the CFG such that node `x` dominates an immediate predecessor of node `y`, but node `x` does not strictly dominate node `y`. In other words, the dominance of node `x` is said to be terminated at its dominance frontier. Given your CFG from part (b), is there any node that has a non-empty dominance frontier? If there is, provide one example of such a node in your CFG by referencing its label, and identify its dominance frontier. The dominance frontier should be represented as a set of node labels surrounded by curly braces. For example, if nodes `r`, `t`, and `x` comprise the dominance frontier of node `y`, then you should include "`y`: {`r`, `t`, `x`}" in your response. If there is no such node, explain why all nodes have an empty dominance frontier.

(e) **(2 Points)** In a CFG, a *back edge* is an edge such that the target node dominates the source node. List all back edges in your CFG from part (b). Back edges should be represented as "`y` $\rightarrow$ `x`," where `y` represents the label of the source node, and `x` represents the label of the target node.

(f) **(2 Points)** Is the CFG from part (b) reducible? Briefly explain why or why not.

[1]Not to be confused with the acronym for a context-free grammar, which is also CFG.

Table 2: Stack Machine Example for Problem 4

Step	*Accumulator*	*Stack*
`acc` $\leftarrow$ 2	2	`<init>`
`push`	2	2, `<init>`
`acc` $\leftarrow$ 3	3	2, `<init>`
`acc` $\leftarrow$ `acc` $*$ `top`	6	2, `<init>`
`pop`	6	`<init>`

4. **(8 Points)** Rachel has a stack machine with an accumulator. She wants to know if it is possible to compute the expression $E = (12 - 3) * (8 + 2)$. Demonstrate to Rachel that it is possible to use the accumulator to compute E by drawing a table that shows the steps of the evaluation, including the contents of the stack and the accumulator. Your table should contain the following columns: *Step*, *Accumulator*, and *Stack*. For *Step*, you can only use `acc` $\leftarrow$ `int` (`acc` represents the accumulator, and `int` represents an integer), `push` (pushes a node from `acc` to the top of the stack), `pop` (pops off a node from the top of the stack), `top` (represents the top of the stack), and the mathematical operators ($+$, $-$, and $*$) in the expression.

   For example, suppose you were computing the expression $F = 2 * 3$. You would then create a table similar to Table 2.