COMS W4115 Fall 2020: Homework Assignment 3 Solutions

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

```
#include <stdio.h>

int x = 3, r = 5;

int foo(float r) {
    return x + 2 * r;
}

int bar(float l, float m) {
    return m - l - x;
}

int pltIsAwesome(int something) {
    int a = 2, b = 1, c = 3, x;
    for(x = 0; x < 2; x++) {
        b -= foo(x);
        float something = 2.5;
        if(x < 1) {
            float x = 2, r = x + 10;
            something--;
        }
        else {
            int x = 3;
            b += 1;
        }
        something -= r;
        int r = 2;
        a += bar(foo(something), x - r);
    }
    b += 2;
    return a * b - x - something;
}

int main() {
    printf("%d\n", foo(pltIsAwesome(5)));
    return 0;
}
```

Table 1: Global Symbol Table for Problem 1

Name	*Kind*	*Type*	*Line Number*
`x`	variable	`int`	3
`r`	variable	`int`	3
`foo`	function	`float` $\rightarrow$ `int`	5
`bar`	function	`float, float` $\rightarrow$ `int`	9
`pltIsAwesome`	function	`int` $\rightarrow$ `int`	13
`main`	function	`void` $\rightarrow$ `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`.

   Here are the four different scopes in function `pltIsAwesome`:

   Scope 1: [13, 32]
   Scope 2: [15, 29]
   Scope 3: [18, 21]
   Scope 4: [22, 25]

   (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 5. Finally, assume that all scopes are static. *Hint: only include variables defined in a scope as part of the symbol table for that scope.*

Table 2: Scope 1

Name	*Kind*	*Type*	*Line Number*
`something`	parameter	`int`	13
`a`	variable	`int`	14
`b`	variable	`int`	14
`c`	variable	`int`	14
`x`	variable	`int`	14

Table 3: Scope 2

Name	*Kind*	*Type*	*Line Number*
`something`	variable	`float`	17
`r`	variable	`int`	27

Table 4: Scope 3

Name	*Kind*	*Type*	*Line Number*
`x`	variable	`float`	19
`r`	variable	`float`	19

Table 5: Scope 4

Name	*Kind*	*Type*	*Line Number*
`x`	variable	`int`	23

(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.

The definition of `r` referenced at line 26 refers to the global variable `r` at **line 3**. To determine this from the symbol tables, we first identify what scope the `r` at line 26 is located in: scope 2.

We follow these steps to determine the correct definition of `r`:

i. Does `r` exist in scope 2? If yes, move on to the next step. Otherwise, skip the next step, and move on to step (iii).
ii. If `r` exists in scope 2, does the line number at which the symbol table says it has been defined precede line 26, *i.e.*, the line number at which `r` is being referenced? If yes, we have found our definition. If not, move on to the next step.
iii. Check the parent scope of scope 2, scope 1, and repeat steps (i) and (ii).
iv. If scope 1 does not contain `r`, then check the global symbol table. If the global symbol table does contain `r`, we have found our definition. If not, there is a compilation error in the code, since we were not able to resolve the definition of `r`.

By step (i), we see that `r` is defined in scope 2 at line 27. However, because line 27 comes after line 26, it is not our definition according to step (ii). This should make sense, since a variable cannot be used before it is defined. The `r` at line 27 corresponds to a *different* variable `r`. We then proceed to step (iii) and check scope 1, the parent scope of scope 2. Since scope 1 does not contain `r`, we need to check the global symbol table by step (iv) and hope that there is a global variable `r`. Indeed, we see that `r` is in the global symbol table, and it has been defined at line 3. This is the correct definition of `r`.

Note that there is an `r` defined in scope 3, but we do not consider it because that `r` is in an inner/child scope. We only consider outer scopes starting at scope 2, not inner scopes. Further, scope 3 has already been removed by the time we reach line 26, so all of the variables defined inside scope 3 no longer exist after line 25 has been scanned.

(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.

Here are the active identifiers:

(`a`, variable, `int`, 14)
(`b`, variable, `int`, 14)
(`c`, variable, `int`, 14)
(`something`, variable, `float`, 17)
(`x`, variable, `float`, 19)
(`r`, variable, `float`, 19)

For example, if `x` were called immediately after line 19 is observed, it would be a reference to the `x` at line 19 (which is active), not any other `x` (all others are inactive or not visible).

(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).

Here are the active identifiers:

(`r`, variable, `int`, 3)
(`a`, variable, `int`, 14)
(`b`, variable, `int`, 14)
(`c`, variable, `int`, 14)
(`x`, variable, `int`, 14)
(`something`, variable, `float`, 17)

For example, if `x` were called immediately after line 26 is observed, it would be a reference to the `x` at line 14 (which is active), not any other `x` (all others are inactive or not visible).

(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.

With static scoping, the value printed at line 35 is **-3**. Static scoping is what we typically see in most modern programming languages. Static scoping refers to the top-level environment of the variable. In other words, a variable defined inside a function is only visible/accessible to that function and not by any functions that are called by that function. Additionally, a variable defined inside a scope is only visible/accessible to that scope and any inner scopes created after that definition, but it is not visible/accessible by any other scopes.

Following the trace of the execution from the beginning, `main` calls `foo`(`pltIsAwesome`(5)) at line 35. `pltIsAwesome` calls `foo` at line 16 with the value of `x`, whose value is given by the current iteration of the `for` loop (`x` is defined at line 14, which can be seen from the symbol table for scope 1). Because of static scoping, the value of `x` in `foo` at line 6 is the value of the global variable `x` at line 3 (since there is no `x` defined inside `foo`), which is 3. Similarly, the value of the `r` in `foo` at line 6 is the value of the parameter `r` at line 5. The other statement in `pltIsAwesome` with function calls is the one at line 28, which has a function call to `bar`. `bar` has a function call to `foo` as its first argument, and this call to `foo` requires the value of the variable `something` defined at line 17 (determined by the symbol table for scope 2). `foo` returns, and `bar` can be executed. Note again with static scoping that the variable `x` at line 10 corresponds to the global variable `x` defined at line 3.

Executing two iterations of the `for` loop in `pltIsAwesome` yields -3 returned at line 31 (`a` is -1 after two modifications at line 28, `b` is -4 after two modifications at line 16 and a modification at line 30, `x` is 2 after incrementing `x` twice in the `for` loop condition and exiting after the second increment makes the loop condition false, and `something` is 5 after taking on the value of the function argument at line 13 once again upon exiting the `for` loop) and -3 as the value printed at line 35 with `foo` at line 35 passed an argument of -3.

(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.

With dynamic scoping, the value printed at line 35 is **7**. Unlike static scoping, dynamic scoping involves traverse the symbol table stack to find the last (correct) occurrence of a symbol. Variables refer to the closest enclosing bindings in the program execution, so identifiers have global bindings. That means symbol tables created in a caller function may be visible to the callee function if those symbol tables still exist, and all variables from those symbol tables are effectively inherited by the scopes of the callee function.

As with the static scoping case, we follow the trace of the execution. Notice that at line 16 when we call `foo`, we pass in the value of `x` based on the current iteration of the `for` loop, just as with static scoping. However, inside function `foo`, we need to be careful about the values of `x` and `r` at line 6. `x` does not take on the value of the global variable `x` defined at line 3 but instead takes on the value of the local variable `x` defined at line 14. Why? With dynamic scoping, you follow these steps:

i. First, check to see if any `x` is defined in (or in the symbol table of) `foo`. If this is the case, then the `x` at line 6 takes on the value of this definition of `x`. If not, move on to the next step.
ii. Check the scope of the caller function `pltIsAwesome` in which the function `foo` was called, scope 2 of `pltIsAwesome`, for any definition of `x`. If this is the case, then the `x` at line 6 takes on the value of this definition of `x`. If not, move on to the next step.
iii. Check the parent scope of scope 2, scope 1, to find the most recent definition of `x` that is visible to scope 2. If an `x` is found, then the `x` at line 6 takes on the value of this definition of `x`. If not, move on to the next step.
iv. Check the scope of the caller function `main` in which the function `pltIsAwesome` was called for any definition of `x`. If this is the case, then the `x` at line 6 takes on the value of this definition of `x`. If not, move on to the next step.
v. Check the global scope for any definition of `x`. If one is found, then the `x` at line 6 takes on the value of this definition of `x`. If not, there is a compilation error in the code, since we were not able to resolve the definition of the `x`.

Following these steps, you will note that step (i) does not result in a definition of `x`, so we proceed to step (ii). At step (ii), you have traversed the symbol table stack and arrived at the symbol table for scope 2. Scope 2 does not contain a definition of `x` from the symbol table, so we proceed to step (iii), at which we will find that `x` does exist in the symbol table for scope 1. Thus, the value of `x` at line 6 will be whatever the value of `x` at line 14 is prior to calling `foo` at line 16.

Note that applying this same algorithm for `r` at line 6 results in the `r` at line 5 (step (i)), since we find the function argument `r` at line 5 to be in the symbol table of `foo`'s scope.

At line 28 of `pltIsAwesome`, the function call to `foo` is made, and the value of `x` at line 6 still refers to the value of `x` defined at line 14 (the symbol tables for scopes 3 and 4 have already been popped from the stack upon exiting those scopes). For the function call to `bar`, the value of `x` at line 6 is, again, the value of `x` defined at line 14.

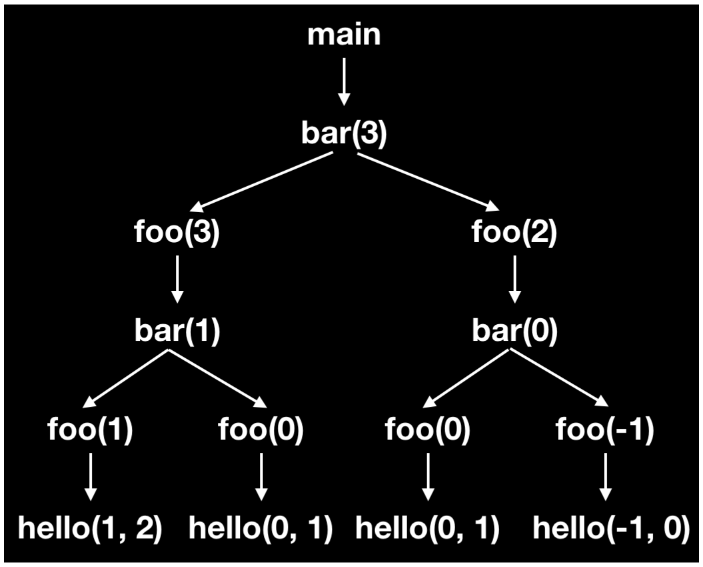
Executing two iterations of the `for` loop in `pltIsAwesome` yields 2 returned at line 31 (`a` is 9, `b` is 1, `x` is 2, and `something` is 5). Let us take a final look at the call to `foo` at line 35. What does the value of `x` correspond to at line 6? Now, `main` is the caller function of the `foo` at line 35 and not `pltIsAwesome`, so we would check the scope of `main` for any definition of `x`. None exists, so we then proceed to the global scope and end up with the global variable `x` defined at line 3. Note that all symbol tables from the function call to `pltIsAwesome` have already been popped from the stack, so that is a telltale sign that no definitions of `x` from `pltIsAwesome` should be considered anymore. Thus, we arrive at 7 as the value printed at line 35 with `foo` at line 35 passed an argument of 2.

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

```
Class PLT {

        int foo(int w) {
            if(w < 2)
                return hello(w, w + 1) (****) + 3;
            return bar(w - 2) (*****);
        }

        int bar(int x) {
            return foo(x) (**) * foo(x - 1) (***);
        }

        int hello(int y, int z) {
            return y * z;
        }

        int main() {
            bar(val) (*);
            return 0;
        }
}
```

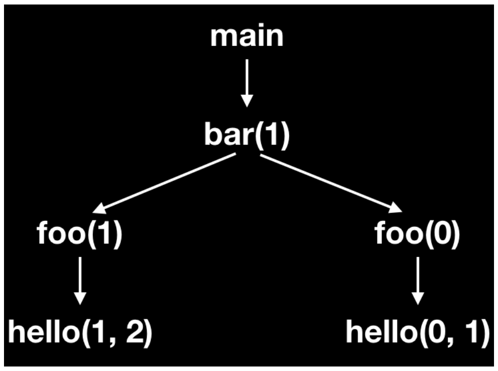
(a) **(5 Points)** Draw the activation tree generated by executing `main` when `val` is 3.

Here is the activation tree for part (a):



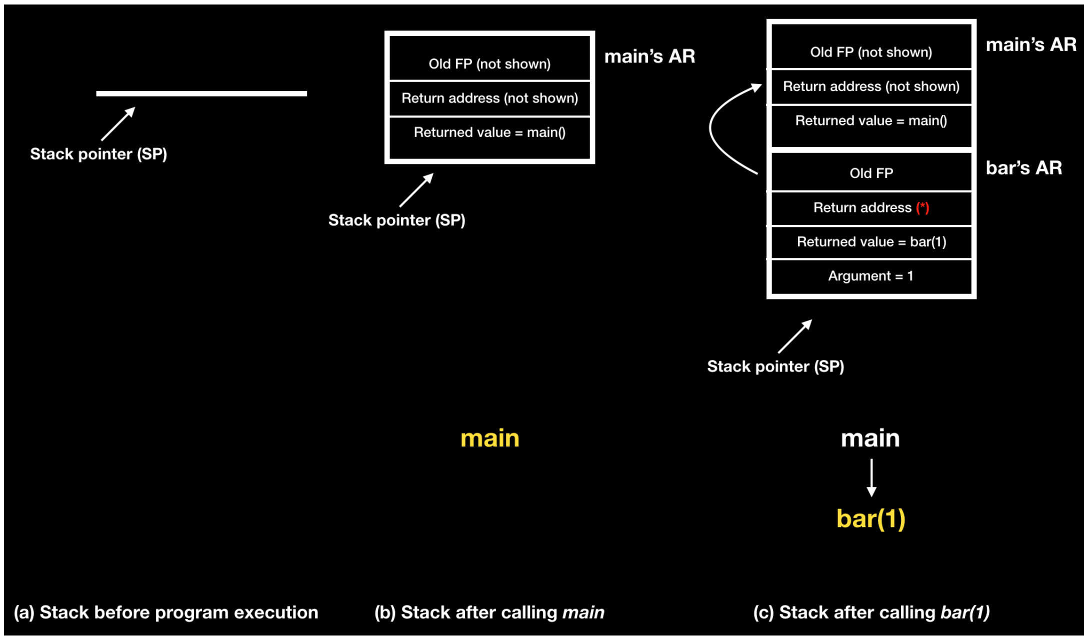
(b) **(10 Points)** Using a runtime stack, show the activation records generated by executing `main` when `val` is 1. You only need to include the old frame pointer, return address, returned value, and function argument(s); don't worry about temporary variables. You can expand the stack in either direction,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

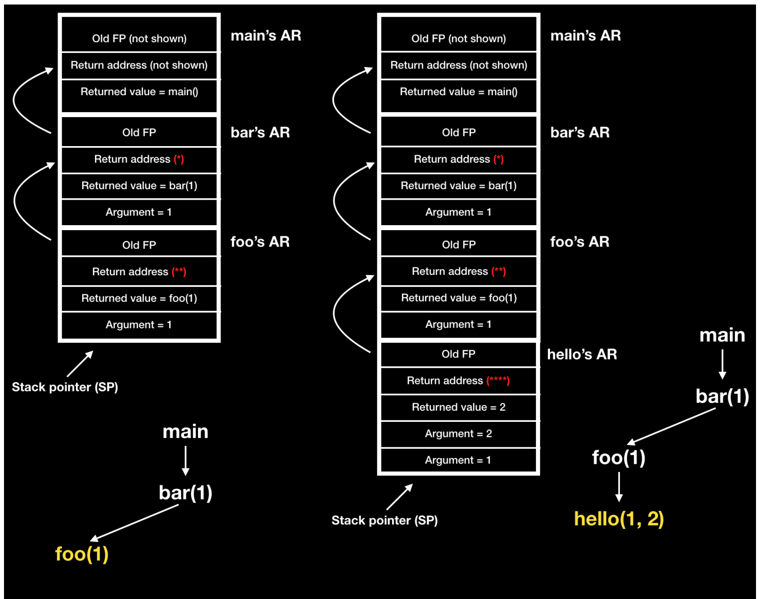
To identify the activation records, we first need to determine the activation tree:



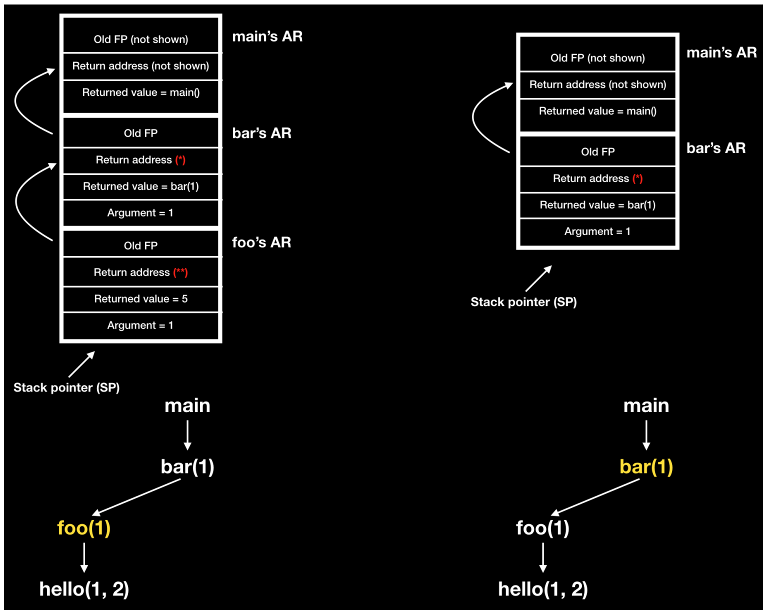
Note that other than `main`, there are 5 procedure calls: one call to `bar`, two calls to `foo`, and two calls to `hello`. Each of these procedure calls will require a separate activation record.

Here are the snapshots of the stack after records are pushed onto and popped off the stack (the direction of expansion chosen here is downward). The gold text indicates the function whose activation record is currently at the top of the stack. The stack pointer has been added in each snapshot, although it was not necessary to include in this problem:

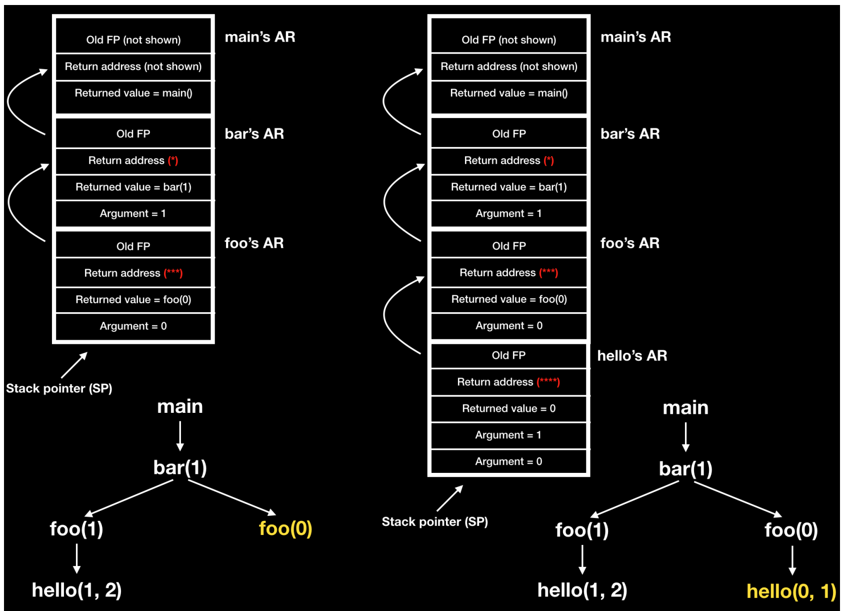




(d) Stack after calling *foo(1)* (e) Stack after calling *hello(1, 2)*

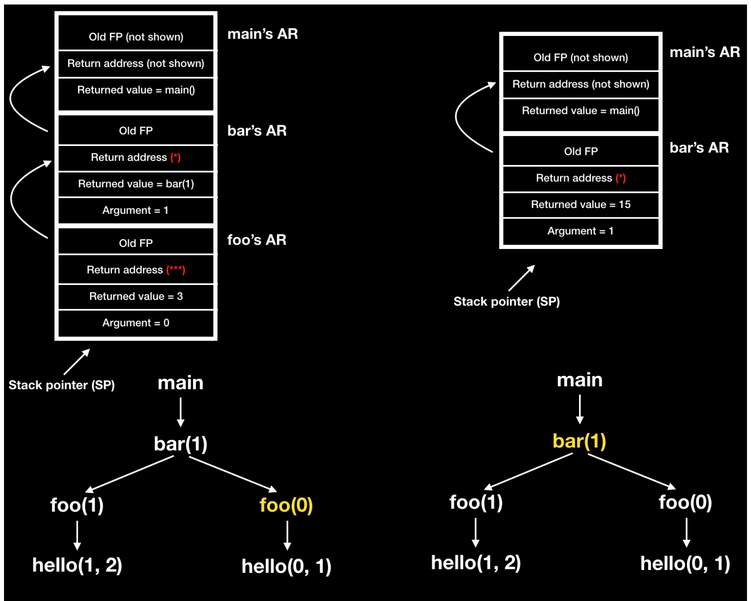


(f) Stack after returning from *hello(1, 2)* (g) Stack after returning from *foo(1)*



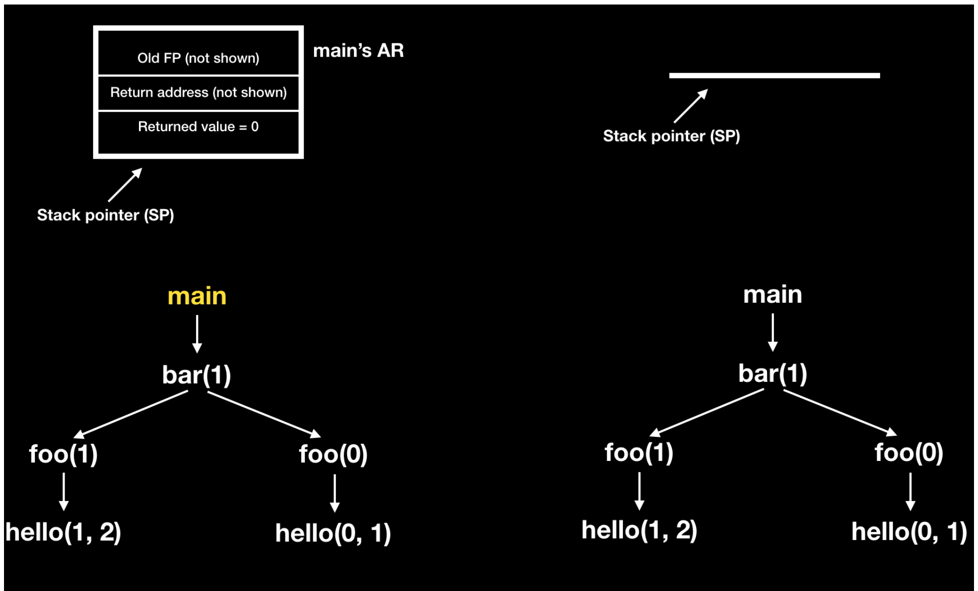
(h) Stack after calling *foo(0)*

(i) Stack after calling *hello(0, 1)*



(j) Stack after returning from *hello(0, 1)*

(k) Stack after returning from *foo(0)*



(l) Stack after returning from *bar(1)*   (m) Stack after returning from *main*

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.

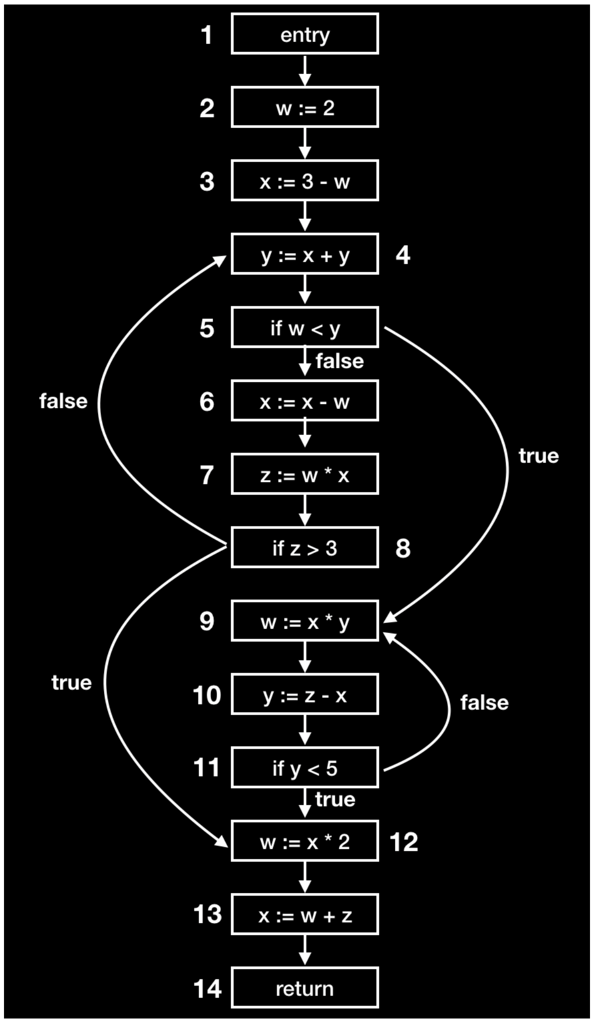
Here are the basic blocks generated from the code snippet: {1, 2, 3}, {4, 5}, {6, 7, 8}, {9}, {10, 11, 12}, {13}, and {14, 15, 16}.

It is acceptable if students decided to segregate the entry block (line 1) into a separate basic block, *i.e.*, the basic block {1, 2, 3} became {1} and {2, 3}.

It is NOT acceptable if students combined lines 8 and 9 and lines 12 and 13 into the same basic blocks, *i.e.*, students had basic blocks containing {8, 9} together and {12, 13} together. Why can we not create these combinations? Think about the definition of a basic block: *a sequence of straight-line code that can be* ***entered only before the first statement/instruction*** *and* ***exited only after the last statement/instruction***. Thus, each basic block must satisfy both conditions of the definition, entrance and exit. Let's take a candidate basic block {6, 7, 8, 9}. Is it possible to enter the basic block at any point other than the beginning of the basic block (before line 6)? No, so we still have a green light. Is it possible to exit the basic block at any point other than the end of the basic block (after line 9)? Well, there is a `goto` instruction at line 8, so it is possible to exit the basic block at line 8 (and jump to line 10, `L2`) before reaching the end of the basic block - this violates the definition of a basic block! So, we must separate lines 8 and 9 into separate basic blocks, making line 8 the terminator of the former basic block and line 9 the leader of the latter basic block. This same reasoning applies to lines 12 and 13 as well.

(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.*

Here is the CFG for part (b):



Note that there are 14 nodes in the above CFG, with node labels 1 to 14. The `goto` instructions in the `if` statements translate to `true` edges, whereas the `goto` instructions in the `else` statements translate to `false` edges. At line 6, there is no `else` statement; however, there is still a `false` edge that is implied for the branch instruction at line 5. In that scenario, the control flow simply "falls" to the next instruction in the code. While explicit `goto` nodes could have been created for all `goto` instructions in the code, they are not necessary.

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

(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.

Here are the dominator sets:

1: {1}
2: {1, 2}
3: {1, 2, 3}
4: {1, 2, 3, 4}
5: {1, 2, 3, 4, 5}
6: {1, 2, 3, 4, 5, 6}
7: {1, 2, 3, 4, 5, 6, 7}
8: {1, 2, 3, 4, 5, 6, 7, 8}
9: {1, 2, 3, 4, 5, 9}
10: {1, 2, 3, 4, 5, 9, 10}
11: {1, 2, 3, 4, 5, 9, 10, 11}
12: {1, 2, 3, 4, 5, 12}
13: {1, 2, 3, 4, 5, 12, 13}
14: {1, 2, 3, 4, 5, 12, 13, 14}

A few notes:

- A dominator set for each node in the CFG will, by definition, be non-empty, since every node always dominates itself.
- Node 1 (the entry node) dominates every node in the CFG (including itself), since all paths in the CFG start from the entry node; thus, all dominator sets must contain node 1. Node 1 will not be dominated by any node except itself. This should make sense.
- Node 14 (the exit node) will not dominate any node except itself, since all paths in the CFG must end at the exit node, and the exit node is not visited before visiting other nodes; thus, only the dominator set for node 14 contains node 14. Again, this should make sense.
- All paths from entry to exit in the CFG must go through nodes 1, 2, 3, 4, 5, 12, 13, and 14. This can be observed by verifying the dominator set of the exit node (node 14), which contains all nodes that are present in every path from entry to exit.
- Node 9 is not dominated by nodes 6, 7, and 8. This is because there is a path from entry to node 9 that does not go through each of those three nodes, namely the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 9$.
- By the same reasoning as the previous note, node 12 is not dominated by nodes 6, 7, 8, 9, 10, and 11. A path to node 12 that does not contain nodes 6, 7, and 8 is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12$. A path to node 12 that does not contain nodes 9, 10, and 11 is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 12$.

(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.

There are nodes in the CFG that have a non-empty dominance frontier. In other words, it is possible to find a node `x` in the CFG such that `x` dominates an immediate predecessor of a different node `y` but does not dominate `y` directly. Here are the dominance frontiers of all nodes in the CFG:

```
1: {}
2: {}
3: {}
4: {4}
5: {4}
6: {4, 12}
7: {4, 12}
8: {4, 12}
9: {9, 12}
10: {9, 12}
11: {9, 12}
12: {}
13: {}
14: {}
```

Since the problem required providing only one example of a node in the CFG with a non-empty dominance frontier, any of the non-empty dominance frontiers could have been listed:

```
4: {4}
5: {4}
6: {4, 12}
7: {4, 12}
8: {4, 12}
9: {9, 12}
10: {9, 12}
11: {9, 12}
```

(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` → `x`," where `y` represents the label of the source node, and `x` represents the label of the target node.

There are exactly two back edges in the CFG:

$$8 \rightarrow 4$$
$$11 \rightarrow 9$$

An edge is considered a back edge of a natural loop if it satisfies the following conditions:

- The source of the edge is the tail of the loop, and the target of the edge is the head of the loop.
- The head of the loop dominates the tail of the loop, *i.e.*, the target of the edge dominates the source of the edge.

Both of the above edges are back edges of natural loops. From the dominator sets computed in part (c), node 4 is in the dominator set of node 8, and node 9 is in the dominator set of node 11, *i.e.*, node 4 (the target of the first back edge) dominates node 8 (the source of the first back edge), and node 9 (the target of the second back edge) dominates node 11 (the source of the second back edge). Further, node 4 is the head of a loop in which node 8 is the tail, and node 9 is the head of a loop in which node 11 is the tail. Both loops are, thus, natural loops.

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

A CFG is reducible if the edges of the CFG can be partitioned into two disjoint sets, one set containing just the forward edges that form an acyclic graph in which every node can be reached via some path starting from the entry node, and another set containing back edges consisting only of edges whose target nodes dominate the source nodes. In other words, for a CFG to be reducible, any loops in the CFG must be natural loops, and all nodes must be reachable. If any loop is non-natural or any node is unreachable, then the CFG is irreducible.

The CFG from part (b) is reducible because you can remove the back edges of all natural loops in the CFG and form an acyclic graph. There are no non-natural loops in this CFG, and the two natural loops in the graph have the back edges identified in part (e), respectively.

Table 6: 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 6.

   In order to help Rachel, here is the table that should be drawn to compute E:

Step	*Accumulator*	*Stack*
`acc` $\leftarrow$ 3	3	`<init>`
`push`	3	`3, <init>`
`acc` $\leftarrow$ 12	12	`3, <init>`
`acc` $\leftarrow$ `acc` $-$ `top`	9	`3, <init>`
`pop`	9	`<init>`
`push`	9	`9, <init>`
`acc` $\leftarrow$ 8	8	`9, <init>`
`push`	8	`8, 9, <init>`
`acc` $\leftarrow$ 2	2	`8, 9, <init>`
`acc` $\leftarrow$ `acc` $+$ `top`	10	`8, 9, <init>`
`pop`	10	`9, <init>`
`acc` $\leftarrow$ `acc` $*$ `top`	90	`9, <init>`
`pop`	90	`<init>`

Here is another way you could have helped Rachel:

Step	*Accumulator*	*Stack*
`acc` $\leftarrow$ 12	12	`<init>`
`push`	12	`12, <init>`
`acc` $\leftarrow$ 3	3	`12, <init>`
`acc` $\leftarrow$ `top` $-$ `acc`	9	`12, <init>`
`pop`	9	`<init>`
`push`	9	`9, <init>`
`acc` $\leftarrow$ 8	8	`9, <init>`
`push`	8	`8, 9, <init>`
`acc` $\leftarrow$ 2	2	`8, 9, <init>`
`acc` $\leftarrow$ `acc` $+$ `top`	10	`8, 9, <init>`
`pop`	10	`9, <init>`
`acc` $\leftarrow$ `acc` $*$ `top`	90	`9, <init>`
`pop`	90	`<init>`