CS314 Spring 2023
Homework 4

Sample Solution

March 30, 2023

1 Problem — Pointers (20 pts)

Given the following correct program in C:

```
int main() {int a, b, c;
    ??? ra; ??? rb; ??? rc; ??? raa; ??? rbb; ???rcc;
    a = 3; b = 2; c = 1;
    ra = &a;
    rb = &b;
    rc = &c;
    ra = rb;
    raa = &rb;
    rc = *raa;
    rcc = raa;
    rc = &a;
    rbb = &rc;
    rb = &c;
    *ra = 4;
    *rb = *ra + 4;
    /* (*) */
    printf("%d %d %d\n", a, b, c);
    printf("%d %d\n", *ra, *rb);
    printf("%d %d %d\n", **raa, **rbb, **rcc);
}
```

1. Give the correct type definitions for pointer variables `ra, rb, rc, raa, rbb, and rcc`.

   Answer:

   ```
   int *ra;
   int *rb;
   int *rc;
   int **rra;
   int **rrb;
   int **rrc;
   ```

2. Draw a picture that shows all of the variables and their contents similar to the picture as shown, for example, in lecture 12, page 12. Also indicate whether your variables live on the stack or on the heap. Your picture should show the variables and their values just before the first print statement (*).

   Answer: All variables live on the stack. See Figure 1.

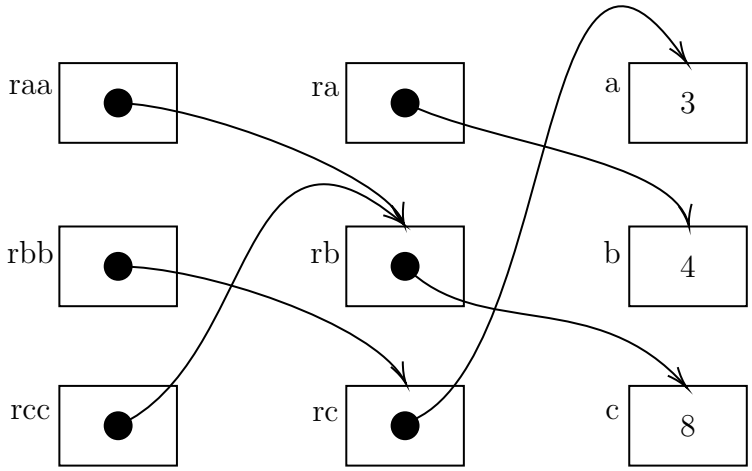

Figure 1: Answer for Problem 1.2. Diagram showing all of the variables and their contents, similar to the figure shown in Lecture 12, page 12, just before the first print statement (*).

3. Show the output from this program.

   Answer:

```
3 4 8
4 8
8 3 8
```

4. Write a statement involving a pointer expression using the variables in this program which is ILLEGAL given your declared types.

   Answer: One illegal statement would be "`ra` = ∗`a`;". Many other statements are possible.

2 Problem — Compiler Optimization and Aliasing (10 pts)

Assume the following program fragment without any control flow branches (straight line code). Your job is to implement a compiler optimization called "constant folding" for straight line code. This optimization identifies program variables with values that are known at compile time. Expressions that consist of only such variables can be evaluated at compile time.

```
begin
  int a, b, c;
  ... /* some other declarations */
  a = 4;
  b = 3;
  ... /* no statements that mention ‘‘a’’ or ‘‘b’’ */
  c = a - b; /* c == 1 ? */
  print c;
end.
```

Would it always be safe for the compiler optimization of constant folding to replace the assignment "`c = a - b`" by "`c = 1`" ? Note that there are no assignments to variables `a` or `b` between "`b = 3`" and "`c = a - b`". The control flow is linear, so there are no branches. Give an example where constant propagation would be not be safe (incorrect) in this situation, without violating any of the above assumptions about the code fragment. Note: You can add declarations of other variables and other statements that do not mention `a` or `b`.

Answer: Performing the constant folding operation would be unsafe. Here is an example by adding a pointer "p":

```
begin
  int a, b, c;
  int *p = NULL; ... /* some other declarations */

  p = &a; /* generating an alias for variable ‘‘a’’ */
  a = 4;
  b = 3;
  *p = 10; ... /* no statements that mention ‘‘a’’ or ‘‘b’’ */
  c = a - b; /* c == 1 ?  NO, since c == 7. */
  print c;
end.
```

We cannot replace "`c = a - b;`" by "`c = 1;`" since `c` now equals 7. *Aliasing* means that there are different names for the same memory location. Compilers typically use *alias analysis* to determine whether optimizations such as constant folding are safe.

3 Problem — Lexical/Dynamic Scoping (18 pts)

Assume variable names written as **capital** letters use **dynamic** scoping and variable names written as **lower case** letters use **static** (lexical) scoping. Assume that procedures return when execution reaches their last statement. Assume that all procedure names are resolved using static (lexical) scoping. Show the output of the entire program execution. Label the output with the location of the print statement (e.g.: (*2*): ...)).

```
program main()
{   int A, b;
    procedure f()
    {   int c;
        procedure g()
        {   int c;
            c = 33;
            ... = ...b...
            print A,b,c; //<<<------ (*1*)
            end g;
        }
        print A,b; //<<<------------ (*2*)
        A = 1; b = 2; c = 3;
        call g();
        print c; //<<<------------ (*3*)
        end f;
    }
    procedure g()
    {   int A,b;
        A = 4; b = 9;
        call f();
        print A,b; //<<<----------(*4*)
        end g;
    }
    A = 5; b = 3;
    print A,b; //<<<------------ (*5*)
    call g();
    print A,b; //<<<----------(*6*)
    end main;
}
```

Answer (2 pts each):

(*1*) 1, 2, 33

(*2*) 4, 3

(*3*) 3

(*4*) 1, 9

(*5*) 5, 3

(*6*) 5, 2

4 Problem — Lexical Scoping Code Generation (40 pts)

Assume that all variables are lexically scoped.

```
program main()
{   int a, b;
    procedure f()
    {   int c;
        procedure g()
        {
            ... = b + c;   //<<<-------- (*A*)
            print a,b,c;
            end g;
        }
        a = 0; c = 1;
        ... = b + c;      //<<<-------- (*B*)
        call g();
        print c;
        end f;
    }
    procedure g()
    {   int a,b;
        a = 3; b = 7;
        call f();
        print a,b;
        end g;
    }
    A = 2; b = 3;
    print a,b;
    call g();
    print a,b;
    end main;
}
```

1. Show the runtime stack with its stack frames, access and control links, and local variables when the execution reaches program point (*A*).

   Answer: See Figure 2.

2. Give the ILOC RISC code for the expressions at program points (*A*) and (*B*). The value of the expressions need to be loaded into a register. The particular register numbers are not important here. *Answer:* Please use answers that use loadAI.

```
(*A*)                       (comments)
loadAI r0, -4 => r1    ;  r1 = FP of f
loadAI r1, -4 => r2    ;  r2 = FP of main
loadAI r2,  8 => r3    ;  r3 = b
                       ;
loadAI r0, -4 => r4    ;  r4 = FP to f
loadAI r4,  4 => r5    ;  r5 = c
add r3, r5    => r6    ;  r6 = b + c
```

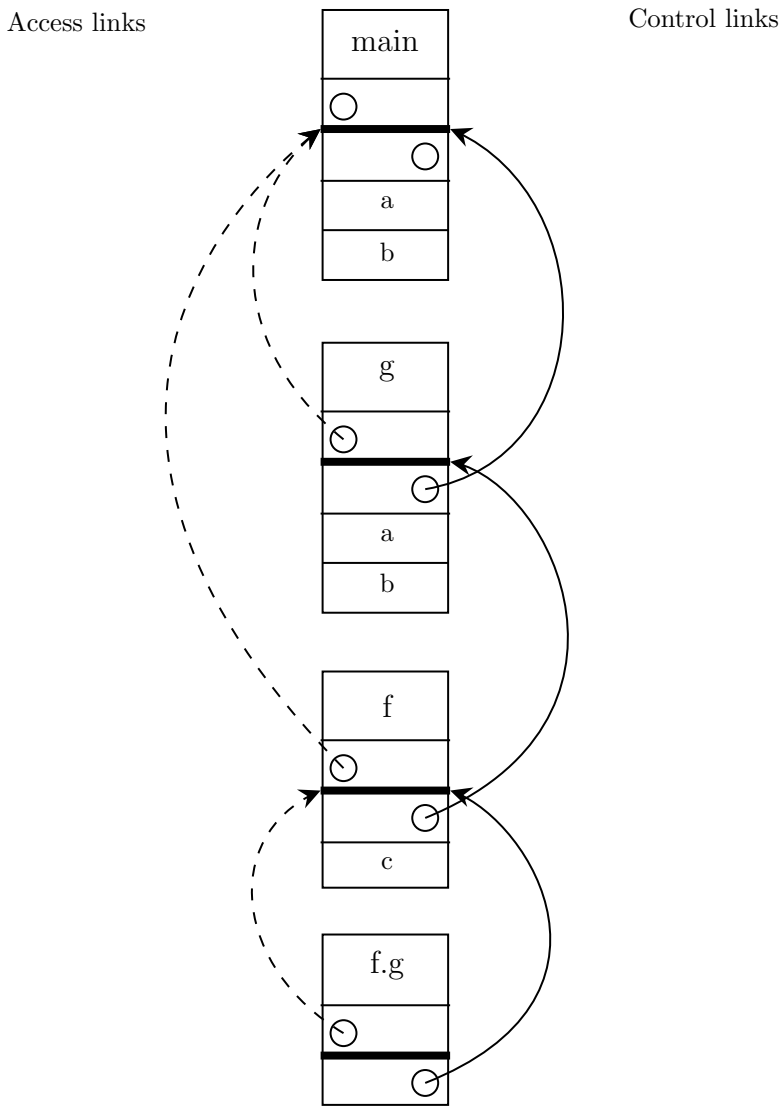

Figure 2: Answer for Problem 4.1. Runtime stack with stack frames, access/control links, and local variables when execution reaches program point (*A*).

```
(*B*)                        (comments)
loadAI r0, -4 => r1     ;  r1 = FP of main
loadAI r1,  8 => r2     ;  r2 = b
                        ;
loadAI r0,  4 => r3     ;  r3 = c
add r2, r3    => r4     ;  r4 = b + c
```

5 Problem — Parameter Passing (12 pts)

Assume that you don't know what particular parameter passing style a programming language is using. In order to find out, you are asked to write a short test program that will print a different output depending on whether a *call-by-value*, *call-by-reference*, or *call-by-value-result* parameter passing style is used. Your test program must have the following form:

```
program main()
{   x integer;
    procedure bar(integer a)
    {
        // statement body of bar
    }

    // statement body of main
    x = 1;
    call bar(x);
    print x;
}
```

The body of procedure bar must only contain assignment statements. For instance, you are not allowed to add any new variable declarations.

1. Write the body of procedure bar such that print x in the main program will print different values for the different parameter passing styles.

   Answer:

   ```
   program main()
   {   x : integer;
       procedure bar(integer a)
       {
           a = a + 2;
           a = x + 3;
       }

       // statement body of main
       x = 1;
       call bar(x);
       print x;
   }
   ```

2. Give the output of your test program and explain why your solution works.

   Answer:

call-by-value	call-by-reference	call-by-value-result
1	6	4

   call-by-value: The value 1 was copied to the variable "a" at the start of the call "call bar(x)". However, because of the *call-by-value* parameter passing mode, the variable "x" remains unchanged.

   call-by-reference: Because of the *call-by-reference* parameter passing mode, the reference of the memory location of "x" is the same as "a" at the start of the call "call bar(x)". We add 2 then 3 to the value stored at this memory location, which is 1, yielding the value of 6.

call-by-value-result: The value 1 was copied to "a" at the start of the call "call bar(x)". The variable "a" is first incremented by 2, and then set to "x + 3". Note, however, that at this point the variable "x" still has value 1. So the variable "a" now has value of 4 after the command "a = x + 3;", which is then copied back to the variable "x" as a result.