# End-of-Year Quiz

Compiler Construction, Fall 2013

Wednesday, December 18th, 2013

Last Name : ______________________

First Name : ______________________

Camipro Number : ______________________

| **Problem** | **Points** | **Rating** |
|---:|---:|:---:|
| 1 | /20 | ☆☆☆☆☆ |
| 2 | /10 | ☆☆☆☆☆ |
| 3 | /20 | ☆☆☆☆☆ |
| 4 | /20 | ☆☆☆☆☆ |
| **Total** | /70 | |

# General Instructions for This Quiz

- You have in total **3 hours 45 minutes**.
- Have your CAMIPRO card ready on the desk.
- You are allowed to use any printed material that you brought yourself to the exam. You are not allowed to use any notes that were not typed-up. Also, you are not allowed to exchange the notes with other students taking the quiz.
- Write the answer of each question on a **separate sheet**, and not on the quiz question sheets. Write your name on each sheet containing your answers.
- Return the printed question sheets back to us. Please rate each problem according to your interest (0 stars: nothing to say. 1-star: Uninteresting problem. 5 stars: Truly awesome problem)
- Use a permanent pen.
- We advise you to do the questions you know best first.
- You will perhaps discover that questions about type systems take longer to understand, but not necessarily longer to solve.

# Problem 1: Type Rules for Collections (20 points)

Consider the following typed language on immutable identifiers:

| | | |
|---|---|---|
| $\mathit{expr} ::= \textbf{val}\ \mathit{ident}\ \texttt{=}\ \mathit{expr};\ \mathit{expr}$ | Variable binding | (1) |
| $\mathit{expr} ::= i$ | where i is an integer constant | (2) |
| $\mathit{expr} ::= (\mathit{ident} : T) \Rightarrow \mathit{expr}$ | Creates an anonymous function | (3) |
| $\mathit{expr} ::= \mathit{expr}(\mathit{expr})$ | Applies a function | (4) |
| $T ::= \mathit{Int}$ | Int type | (5) |
| $T ::= T \Rightarrow T$ | Function type | (6) |
| $T ::= \mathit{Collection}[T]$ | Collection type | (7) |

For the purpose of this exercise, we add a polymorphic type $\mathit{Collection}[T]$ to the language, where $T$ can be any existing type. We extend the expression syntax with existing customized function symbols:

| | | |
|---|---|---|
| $\mathit{expr} ::= \mathrm{EmptyCol}[\alpha]$ | Creates an empty collection of elements of type $\alpha$ | (8) |
| $\mathit{expr} ::= \mathrm{add}(\mathit{expr}, \mathit{expr})$ | Adds an element to a collection | (9) |
| $\mathit{expr} ::= \mathrm{permute}(\mathit{expr})$ | Returns a function from a collection | (10) |
| $\mathit{expr} ::= \mathrm{map}(\mathit{expr})$ | Returns a function from a collection | (11) |
| $\mathit{expr} ::= \mathrm{flatMap}(\mathit{expr})$ | Returns a function from a collection | (12) |

Informally, the semantics of the extensions is the following:

- $\mathrm{EmptyCol}[\alpha]$ (line 8) takes a type parameter and returns an empty collection of this type.
- $\mathrm{add}(e_1, e_2)$ (line 9) takes a collection $e_1$ and an element $e_2$ and returns the collection $e_1$ to which the element has been added.
- $\mathrm{permute}(\mathit{expr})$ (line 10) takes a collection, and returns a function which is a permutation mapping of the elements of the collection. See example below.
- $\mathrm{map}(\mathit{expr})$ (line 11) takes a collection and returns a function. This function accepts a mapping from an element to another element and returns the mapped original collection.
- $\mathrm{flatMap}(\mathit{expr})$ (line 12) takes a collection and returns a function. This function accepts a mapping from an element to a collection of elements and returns the union of all images of elements of the original collection.

For example, if $x = \mathrm{add}(\mathrm{add}(\mathrm{add}(\mathrm{EmptyCol}[\mathit{Int}], 1), 2), 3)$, it could be that:

$$\mathrm{permute}(x)(1) == 2$$

$$\mathrm{permute}(x)(2) == 3$$

$$\mathrm{permute}(x)(3) == 1$$

For any other integer, $i$, permute$(x)(i)$ could be either 1, 2 or 3.
If collections were sets, we would also have the following (to illustrate):

$$\text{map}(x)(a \Rightarrow 1) = \{1\}$$

$$\text{flatMap}(x)(a \Rightarrow \text{add}(x, a+3)) = \{1,2,3,4,5,6\}$$

$$\text{map}(x)(\text{permute}(x)) = \text{map}(x)((i : \mathit{Int}) \Rightarrow \text{permute}(x)(i)) = x$$

The type rules for expressions are the following:

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash e : T \;\; \Gamma \vdash f : T \Rightarrow U}{\Gamma \vdash f(e) : U} \qquad \frac{\Gamma \vdash e_1 : T \qquad \Gamma, (x, T) \vdash e_2 : U}{\Gamma \vdash (\text{val } x = e_1;\, e_2) : U}$$

$$\frac{\Gamma \vdash x : T \qquad T <: U}{\Gamma \vdash x : U} \qquad \frac{\Gamma \vdash x : T \Rightarrow U \qquad U <: V \qquad W <: T}{\Gamma \vdash x : W \Rightarrow V}$$

We also allow standard subtyping rules, with function results covariant and function arguments contravariant.

**a)** **[5 pts]** Complete the following type rule templates (replace all ??? occurences) so that they are consistent with the described meaning of the extensions and sufficient to type check the extension if we exclude subtyping. You can abbreviate *Collection*$[T]$ as $C[T]$. Here is a example rule for flatMap:

$$\frac{\Gamma \vdash e : \mathit{Collection}[T]}{\Gamma \vdash \text{flatMap}(e) : (T \Rightarrow \mathit{Collection}[U]) \Rightarrow \mathit{Collection}[U]}$$

$$\frac{???}{\Gamma \vdash \text{EmptyCol}[\alpha] :???} \qquad \frac{???}{\Gamma \vdash \text{add}(e_1, e_2) :???} \qquad \frac{???}{\Gamma \vdash \text{permute}(e) :???} \qquad \frac{???}{\Gamma \vdash \text{map}(e) :???}$$

---

## solution

$$\frac{}{\Gamma \vdash \text{EmptyCol}[\alpha] : \mathit{Collection}[\alpha]} \qquad \frac{\Gamma \vdash e_1 : \mathit{Collection}[T] \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \text{add}(e_1, e_2) : \mathit{Collection}[T]} \qquad \frac{\Gamma \vdash e_1 : \mathit{Collection}[T]}{\Gamma \vdash \text{permute}(e) : T \Rightarrow T}$$

$$\frac{\Gamma \vdash e : \mathit{Collection}[T]}{\Gamma \vdash \text{map}(e) : (T \Rightarrow U) \Rightarrow \mathit{Collection}[U]}$$

---

**b)** **[10 pts]** Prove that the following program type checks by writing the type derivation tree. You can write a separate tree for each right-hand side of variable definition.

```
val x = add(add(EmptyCol[Int], 1), 2)
val z = add(add(x, 3), 4)
val y = add(EmptyCol[Int ⇒ Int], permute(z))
flatMap(y)(map(x))
```

---

## solution

x typechecks to $\mathit{Collection}[\mathit{Int}]$, z to $\mathit{Collection}[\mathit{Int}]$ and y to $\mathit{Collection}[\mathit{Int} \Rightarrow \mathit{Int}]$ Therefore map(x) typechecks to $(\mathit{Int} \Rightarrow \mathit{Int}) \Rightarrow \mathit{Collection}[\mathit{Int}]$ and flatMap(y)(map(x)) typechecks to $\mathit{Collection}[\mathit{Int}]$

---

Let us consider the following extended type system used to prevent at compilation time the calling of permute on empty sets. We introduce for each regular collection type $\mathit{Collection}[T]$ the empty-annotated types $\mathit{Collection}^{+}[T]$ and $\mathit{Collection}^{-}[T]$.

- $\mathit{Collection}^{+}[T]$ meaning that the collection may be empty,
- $\mathit{Collection}^{-}[T]$ meaning that the collection may not be empty.

It follows that $\mathit{Collection}^{-}[T]$ <: $\mathit{Collection}^{+}[T]$. This is the only subtyping relation available between collections here.

**c) [10 pts]** Give sound type rules for this language. Remember, that the types in your rules have to be $\mathit{Collection}^{-}[T]$ or $\mathit{Collection}^{+}[T]$, $\mathit{Collection}[T]$ only appears in the code.

---

## solution

$$\frac{}{\Gamma \vdash \mathrm{EmptyCol}[\alpha] : \mathit{Collection}^{+}[\alpha]} \quad \frac{\Gamma \vdash e_1 : \mathit{Collection}^{+}[T] \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \mathrm{add}(e_1, e_2) : \mathit{Collection}^{-}[T]} \quad \frac{\Gamma \vdash e_1 : \mathit{Collection}^{-}[T]}{\Gamma \vdash \mathrm{permute}(e) : T \Rightarrow T}$$

$$\frac{\Gamma \vdash e : \mathit{Collection}^{-}[T]}{\Gamma \vdash \mathrm{map}(e) : (T \Rightarrow U) \Rightarrow \mathit{Collection}^{-}[U]} \quad \frac{\Gamma \vdash e : \mathit{Collection}^{+}[T]}{\Gamma \vdash \mathrm{map}(e) : (T \Rightarrow U) \Rightarrow \mathit{Collection}^{+}[U]}$$

$$\frac{\Gamma \vdash e : \mathit{Collection}^{+}[T]}{\Gamma \vdash \mathrm{flatMap}(e) : (T \Rightarrow \mathit{Collection}^{+}[U]) \Rightarrow \mathit{Collection}^{+}[U]}$$

---

# Problem 2: Code Generation for Switch (10 points)

In the following exercise we consider compilation to a stack machine that uses JVM instructions.
Suppose that we extend our language with a special additional switch-case statement on integers, with a way to reswitch on another integer if needed.

switch(i) { (case $(n_k \mid \_)$ => $(e_k \mid$ reswitch $(e_k)))^*$ }

switch executes the code $e_k$ corresponding to the expression $n_k$ when the result of $n_k$ is equal to the result of $i$, and if nothing matches it executes the default statement introduced by case _.

reswitch$(e_k)$ is a jumping expression. It computes the value of $e_k$ which should be an integer and re-runs the first outer switch on the resulting value. Its effectively creates a loop.

An example of a switch statement using expressions written as a, b, c, e, f, g, h, i can be the following:

```
switch(i) {
  case a => e
  case b => reswitch(f)
  case c => g
  case _ => h
}
```

**a)** **[10 pts]** Give the translation of the switch construct above to JVM instructions.

For this question, you can assume a number of 3 case statements, a reswitch on the second, and one default statement as above. See the available jvm bytecode on the next page. To avoid recomputing i, you might want to duplicate the value for multiple comparisons. Use square brackets ⟦ ⟧ around the generic expressions like i or a to compute them.

## solution

```
         ⟦i⟧
top      dup
         ⟦a⟧
         if_icmpne case2
         pop
         ⟦e⟧
         goto end
case2    dup
         ⟦b⟧
         if_icmpne case3
         pop
         ⟦f⟧
         goto top
case3    dup
         ⟦c⟧
         if_icmpne default
         pop
         ⟦g⟧
         goto end
default  pop
         ⟦h⟧
         goto end
end
```

These are selected bytecode instructions, mostly for integers, for your quick reference.

| | |
|---|---|
| iload_x | Loads the integer value of the local variable in slot x on the stack. $x \in \{0, 1, 2, 3\}$ |
| iload X | Loads the value of the local variable pointed to by index X on the top of the stack. |
| iconst_x | Loads the integer constant x on the stack. $x \in \{0, 1, 2, 3, 4, 5\}$. |
| istore_x | Stores the current value on top of the stack in the local variable in slot x. $x \in \{0, 1, 2, 3\}$ |
| istore X | Stores the current value on top of the stack in the local variable indexed by X. |
| ireturn | Method return statement (note that the return value has to have been put on the top of the stack beforehand. |
| iadd | Pop two (integer) values from the stack, add them and put the result back on the stack. |
| isub | Pop two (integer) values from the stack, subtract them and put the result back on the stack. |
| imult | Pop two (integer) values from the stack, multiply them and put the result back on the stack. |
| idiv | Pop two (integer) values from the stack, divide them and put the result back on the stack. |
| irem | Pop two (integer) values from the stack, put the result of $x_1 \% x_2$ back on the stack. |
| ineg | Negate the value on the stack. |
| iinc x, y | Increment the variable in slot x by amount y. |
| ior | Bitwise OR for the two integer values on the stack. |
| iand | Bitwise AND for the two integer values on the stack. |
| ixor | Bitwise XOR for the two integer values on the stack. |
| ifXX L | Pop one value from the stack, compare it zero according to the operator XX. If the condition is satisfied, jump to the instruction given by label L. XX ∈ { eq, lt, le, ne, gt, ge, null, nonnull } |
| if_icmpXX L | Pop two values from the stack and compare against each other. Rest as above. |
| goto L | Unconditional jump to instruction given by the label L. |
| pop | Discard word currently on top of the stack. |
| dup | Duplicate word currently on top of the stack. |
| swap | Swaps the two top values on the stack. |
| aload_x | Loads an object reference from slot x. |
| aload X | Loads an object reference from local variable indexed by X. |
| iaload | Loads onto the stack an integer from an array. The stack must contain the array reference and the index. |
| iastore | Stores an integer in an array. The stack must contain the arrayreference, the index and the value, in that order. |

# Problem 3: Data Flow Analysis for Variable Ranges (20 points)

Consider the following code fragment. Assume that the function *input()* returns a (random) integer in the range $[-5, 5]$ (i.e. $-5$ and 5 included). Suppose our language now also allows `break` and `continue` statements within loops with the following meaning:

**continue** : stop the current loop iteration and continue with the next loop iteration

**break** : jump out of the current loop iteration and continue executing after the loop

We wish to perform a range analysis of the variables using the domain of intervals. That is, at any program point each variable can have the following value:

- empty interval (noted $\bot$, bottom element of the lattice)
- regular interval $[a, b] = \{x \mid a \leq x \leq b\}$, with $-128 \leq a \leq b \leq 127$.
- $\top$ (top of the lattice, denoting any value is possible, including error values stemming from division by zero)

Operations on intervals are defined in the usual way:

$$[a, b] \circ [c, d] = [min(S), max(S)] \text{ where } S = \{x \circ y \mid a \leq x \leq b \;\&\&\; c \leq y \leq d\}$$

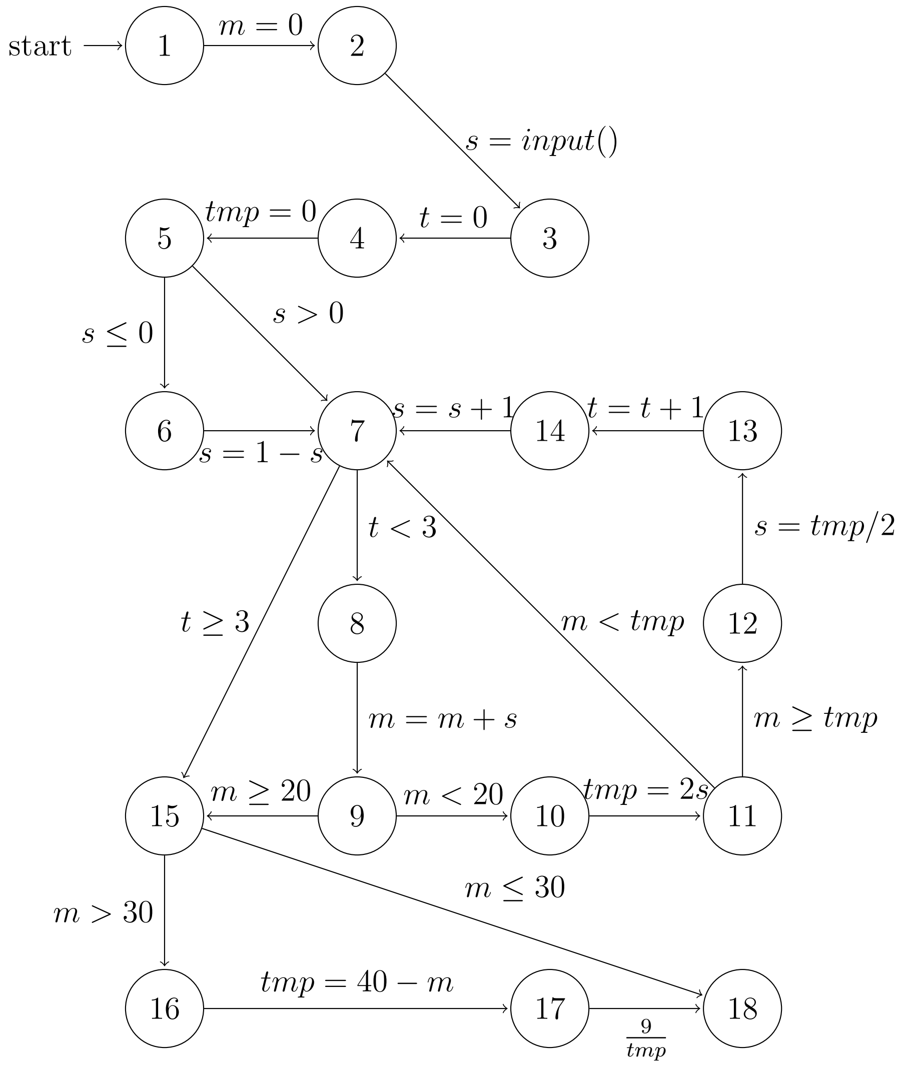
where $\circ \in \{+, -, *, /\}$.

```
var m = 0 // Money
var s = input() // Salary, in the range [−5,5]
var t = 0 // Time
var tmp = 0
if( s ≤ 0 )
  s = 1−s

while ( t < 3 ) {
  m = m + s

  if (m ≥ 20)
    break
  tmp = 2 ∗ s
  if (m < tmp)
    continue
  s = tmp / 2
  t = t + 1
  s = s + 1
}
if(m > 30) {
  tmp = 40−m
  tmp = 9/tmp
}
```

**a)** [**5 pts**] Draw the control flow graph for the code.

---

## solution



---

**b)** [**10 pts**] Now run the dataflow analysis as done in class using the domain of all integer intervals, until convergence.

Indicate the ranges of the variables $m$, $s$, $t$ and $tmp$ at each point in your control flow graph. Recall that the join operation on intervals is defined as follows:

$$[a, b] \sqcup [c, d] = [min(a, c), \quad max(b, d)]$$

You may want to deal with assignment first, then with the content of the while loop, and finally the remaining part.

---

## solution

Here is the first iteration, without processing the backward arrows. We process the states on 12, 13 and 14 when the loop converges. Remaining iterations will skip unchanging states 1 to 5

| *num* | *m* | *s* | *t* | *tmp* |
|---|---|---|---|---|
| 1 | $\top$ | $\top$ | $\top$ | $\top$ |
| 2 | $[0,0]$ | $\top$ | $\top$ | $\top$ |
| 3 | $[0,0]$ | $[-5,5]$ | $\top$ | $\top$ |
| 4 | $[0,0]$ | $[-5,5]$ | $[0,0]$ | $\top$ |
| 5 | $[0,0]$ | $[-5,5]$ | $[0,0]$ | $[0,0]$ |
| 6 | $[0,0]$ | $[-5,0]$ | $[0,0]$ | $[0,0]$ |
| 7 | $[0,0]$ | $[1,6]$ | $[0,0]$ | $[0,0]$ |
| 8 | $[0,0]$ | $[1,6]$ | $[0,0]$ | $[0,0]$ |
| 9 | $[1,6]$ | $[1,6]$ | $[0,0]$ | $[0,0]$ |
| 10 | $[1,6]$ | $[1,6]$ | $[0,0]$ | $[0,0]$ |
| 11 | $[1,6]$ | $[1,6]$ | $[0,0]$ | $[2,12] \Rightarrow 7$ |
| 12 | $[1,6]$ | $[1,3]$ | $[0,0]$ | $[2,6]$ |
| 13 | $[2,6]$ | $[1,3]$ | $[0,0]$ | $[2,6]$ |
| 14 | $[2,6]$ | $[1,3]$ | $[1,1]$ | $[2,6] \Rightarrow 7$ |

The second iteration after processing the backward arrows give the following:

| *num* | *m* | *s* | *t* | *tmp* |
|---|---|---|---|---|
| 7 | $[0,6]$ | $[1,6]$ | $[0,1]$ | $[0,12]$ |
| 8 | $[0,6]$ | $[1,6]$ | $[0,1]$ | $[0,12]$ |
| 9 | $[1,12]$ | $[1,6]$ | $[0,1]$ | $[0,12]$ |
| 10 | $[1,12]$ | $[1,6]$ | $[0,1]$ | $[0,12]$ |
| 11 | $[1,12]$ | $[1,6]$ | $[0,1]$ | $[2,12]$ |
| 12 | $[2,12]$ | $[1,6]$ | $[0,1]$ | $[2,12]$ |
| 13 | $[2,12]$ | $[1,6]$ | $[0,1]$ | $[2,12]$ |
| 14 | $[2,12]$ | $[1,6]$ | $[1,2]$ | $[2,12] \Rightarrow 7$ |

Iteration 3:

| *num* | *m* | *s* | *t* | *tmp* |
|---|---|---|---|---|
| 7 | $[0,12]$ | $[1,7]$ | $[0,2]$ | $[0,12]$ |
| 8 | $[0,12]$ | $[1,7]$ | $[0,2]$ | $[0,12]$ |
| 9 | $[1,19]$ | $[1,7]$ | $[0,2]$ | $[0,12]$ |
| 10 | $[1,19]$ | $[1,7]$ | $[0,2]$ | $[0,12]$ |
| 11 | $[1,19]$ | $[1,7]$ | $[0,2]$ | $[2,14] \Rightarrow 7$ |
| 12 | $[2,19]$ | $[1,7]$ | $[0,2]$ | $[2,14]$ |
| 13 | $[2,19]$ | $[1,7]$ | $[0,2]$ | $[2,14]$ |
| 14 | $[2,19]$ | $[1,7]$ | $[1,3]$ | $[2,14] \Rightarrow 7$ |

Iteration 4:

| $num$ | $m$ | $s$ | $t$ | $tmp$ |
|---|---|---|---|---|
| 7 | $[0, 19]$ | $[1, 8]$ | $[0, 3]$ | $[0, 14]$ |
| 8 | $[0, 19]$ | $[1, 8]$ | $[0, 2]$ | $[0, 14]$ |
| 9 | $[1, 27]$ | $[1, 8]$ | $[0, 2]$ | $[0, 14]$ |
| 10 | $[1, 19]$ | $[1, 8]$ | $[0, 2]$ | $[0, 14]$ |
| 11 | $[1, 19]$ | $[1, 8]$ | $[0, 2]$ | $[2, 16] \Rightarrow 7$ |
| 12 | $[2, 19]$ | $[1, 8]$ | $[0, 2]$ | $[2, 16]$ |
| 13 | $[2, 19]$ | $[1, 8]$ | $[0, 2]$ | $[2, 16]$ |
| 14 | $[2, 19]$ | $[1, 8]$ | $[1, 3]$ | $[2, 16] \Rightarrow 7$ |

Iteration 5:

| $num$ | $m$ | $s$ | $t$ | $tmp$ |
|---|---|---|---|---|
| 7 | $[0, 19]$ | $[1, 9]$ | $[0, 3]$ | $[0, 16]$ |
| 8 | $[0, 19]$ | $[1, 9]$ | $[0, 2]$ | $[0, 16]$ |
| 9 | $[1, 28]$ | $[1, 9]$ | $[0, 2]$ | $[0, 16]$ |
| 10 | $[1, 19]$ | $[1, 9]$ | $[0, 2]$ | $[0, 16]$ |
| 11 | $[1, 19]$ | $[1, 9]$ | $[0, 2]$ | $[2, 18] \Rightarrow 7$ |
| 12 | $[2, 19]$ | $[1, 9]$ | $[0, 2]$ | $[2, 18]$ |
| 13 | $[2, 19]$ | $[1, 9]$ | $[0, 2]$ | $[2, 18]$ |
| 14 | $[2, 19]$ | $[1, 9]$ | $[1, 3]$ | $[2, 18] \Rightarrow 7$ |

Iteration 6:

| $num$ | $m$ | $s$ | $t$ | $tmp$ |
|---|---|---|---|---|
| 7 | $[0, 19]$ | $[1, 10]$ | $[0, 3]$ | $[0, 18]$ |
| 8 | $[0, 19]$ | $[1, 10]$ | $[0, 2]$ | $[0, 18]$ |
| 9 | $[1, 29]$ | $[1, 10]$ | $[0, 2]$ | $[0, 18]$ |
| 10 | $[1, 19]$ | $[1, 10]$ | $[0, 2]$ | $[0, 18]$ |
| 11 | $[1, 19]$ | $[1, 10]$ | $[0, 2]$ | $[2, 20] \Rightarrow 7$ |
| 12 | $[2, 19]$ | $[1, 10]$ | $[0, 2]$ | $[2, 19]$ |
| 13 | $[2, 19]$ | $[1, 9]$ | $[0, 2]$ | $[2, 19]$ |
| 14 | $[2, 19]$ | $[1, 9]$ | $[1, 3]$ | $[2, 19] \Rightarrow 7$ |

Iteration 7:

| $num$ | $m$ | $s$ | $t$ | $tmp$ |
|---|---|---|---|---|
| 7 | $[0, 19]$ | $[1, 10]$ | $[0, 3]$ | $[0, 20]$ |
| 8 | $[0, 19]$ | $[1, 10]$ | $[0, 2]$ | $[0, 20]$ |
| 9 | $[1, 29]$ | $[1, 10]$ | $[0, 2]$ | $[0, 20]$ |
| 10 | $[1, 19]$ | $[1, 10]$ | $[0, 2]$ | $[0, 20]$ |
| 11 | $[1, 19]$ | $[1, 10]$ | $[0, 2]$ | $[2, 20]$ |
| 12 | $[2, 19]$ | $[1, 10]$ | $[0, 2]$ | $[2, 19]$ |
| 13 | $[2, 19]$ | $[1, 9]$ | $[0, 2]$ | $[2, 19]$ |
| 14 | $[2, 19]$ | $[1, 9]$ | $[1, 3]$ | $[2, 19]$ |

We finish the remaining states.

| $num$ | $m$ | $s$ | $t$ | $tmp$ |
|---|---|---|---|---|
| 15 | $[0, 29]$ | $[1, 10]$ | $[0, 3]$ | $[0, 20]$ |
| 16 | $[]$ | $[1, 10]$ | $[0, 3]$ | $[0, 20]$ |
| 17 | $[]$ | $[1, 10]$ | $[0, 3]$ | $[]$ |
| 18 | $[0, 29]$ | $[1, 10]$ | $[0, 3]$ | $[0, 20]$ |

**c)** [**5 pts**] What are the possible ranges of the variables at the end of the code fragment as computed by the analysis? In particular, what does the analysis say about the possibility of division by zero in `9/tmp`?

# Problem 4: Register Allocation (20 points)

We next consider compilation and register allocation for the assignment statement

$$x = x - y * z - u * (z - y);$$

**a)** [**3 pts**] Draw an abstract syntax tree for this assignment statement.

**b)** [**3 pts**] Transform this statement into a sequence $S$ of assignments, each with only one operator on the right-hand side, by introducing finitely many fresh variables $F = \{t_1, \ldots, t_n\}$ for sub-expressions of the right-hand side. Your translation should introduce a distinct fresh variable for each proper non-trivial sub-expression of right-hand side (so, do not introduce fresh variables for the variables $x, y, z, u$ themselves, nor for the entire right-hand side). Show the translated sequence of statements $S$ and write down the set of fresh variables $F$ that the translation introduced.

**c)** [**3 pts**] Compute the set of live variables between each two assignment statements in the above sequence $S$. Assume that the set of live variables at the point after the end of the sequence $S$ is $\{x, y, z, u\}$.

**d)** [**3 pts**] Draw an interference graph whose nodes are variables $\{x, y, z, u\} \cup F$ and whose edges are computed based on the live variables computed above for each program point. Explain why it is not possible to color this graph using 4 colors.

**e)** [**5 pts**] Find a coloring of the graph using 5 colors $C = \{a, b, c, d, e\}$. Feel free to use whatever method you wish to find the coloring (including guessing or running the algorithm we showed in the class), as long as the coloring is correct. Show the final coloring by listing an element of $C$ next to each variable. Verify correctness of your coloring by computing, for each program point in $S$, the set of colors corresponding to live variables; this set should have the same number of elements as the sets of live variables.

**f)** [**3 pts**] Map the sequence of statements $S$ into a new sequence $S'$ of register instructions that, instead of variables $\{x, y, z, u\} \cup F$ use registers $\{RA, RB, RC, RD, RE\}$. The sequences $S$ and $S'$ should have the same length.