CS143 Final
Spring 2020

- Please read all instructions (including these) carefully.
- There are four questions on the exam, some with multiple parts. The final is scoped to 80 minutes, so do not feel you have to spend more time than that. In light of recent events and the need for accommodations, however, you have 24 hours to complete it as you see fit. At 3pm pacific time on Wednesday June 10th, gradescope will close, so make sure submit what you have by then.
- The exam is open note. You may use laptops, phones, e-readers, and the internet, but you may not consult anyone.
- You must upload your answers to gradescope and tag each question, just like the written assignments. You may submit images of hand-written answers taken with your phone (but allow yourself time to send the image to your computer, so you can upload it to gradescope). It is your responsibility, however, to ensure they are legible. Computer typed answers as a PDF are also permitted.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

Problem	Max points	Points
1	30	
2	25	
3	15	
4	30	
TOTAL	100	

In accordance with both the letter and spirit of the Honor Code, I have neither given nor received assistance on this examination.

Type your name as a signature below:

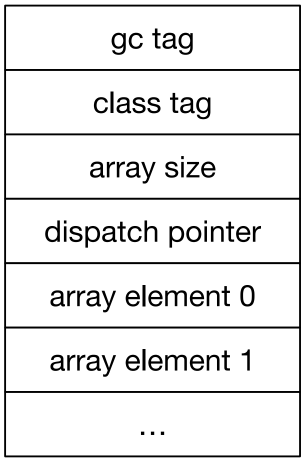
1. **Arrays**

In WA3 we extended Cool with arrays. In this question, we will further develop Cool's new array feature. The context-free grammar of the arrays are:

```
expr := new TYPE[expr]
          | expr[expr]
          | expr[expr] <- expr
```

a) Describe a runtime layout of arrays. That is, what data must be placed on the heap when allocating an array?

Answer:



The class tag and dispatch pointer maintain the invariant that everything in Cool is an object.

b) Describe how you would extend mark and sweep garbage collection to handle arrays. Answer in 1–3 sentences.

Answer:

- During mark phase: Follow the regular algorithm, except if you encounter an array, add all of the pointers in the array to the TODO list.
- During sweep phase: Nothing changes here. If an array ends up un-marked add it to the free list as usual. The individual pointers that are in an array may be accessed through other means so they may not end up on the free list.

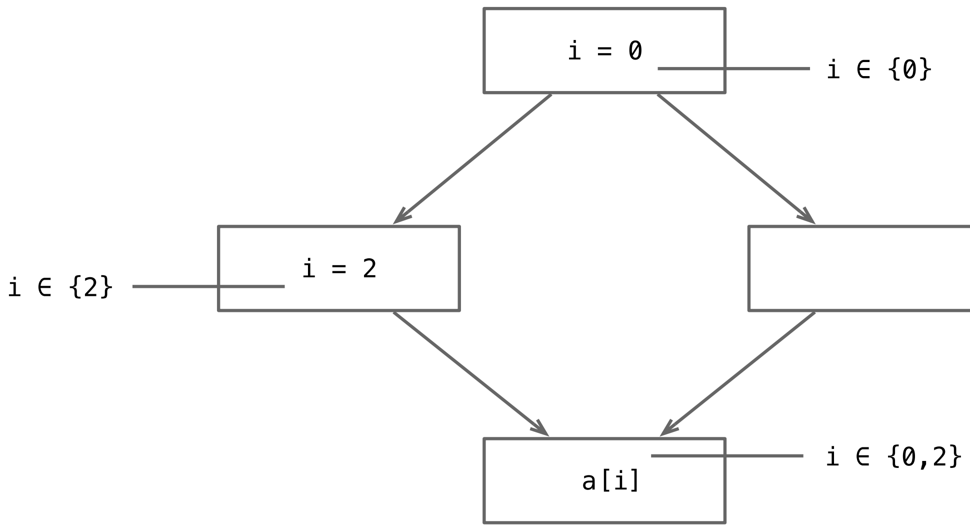
c) We would like to report an error when a used indexes an array out of bounds (i.e., index less than 0 or more than the array size). Given your runtime design, write the mips assembly code that your compiler would generate for the load `a[i]`. This could should jump to the label `error` if `i` is out of bounds. You may assume the address to `a` is in register `r1` and that the value of integer `i` is in register `r2`. Place the result of the load in register `r3`.

Answer:

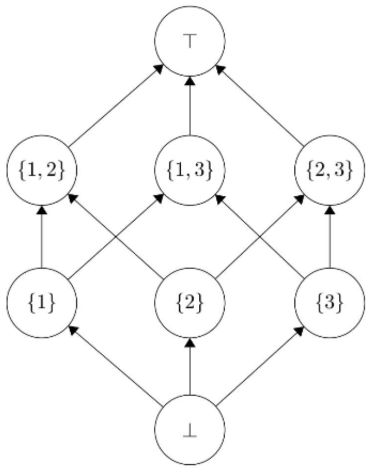
```
bltz $r2, error
lw $t1, 4($r1)
sub $t1, $t1, 3      // assumes object size includes metadata
bge $r2, $t1, error
sll $r2, $r2, 2      // convert words to bytes
addiu $r1, $r1, 12   // move to start of array
addiu $r1, $r2, $r1  // get correct location in array
lw $r3, 0($r2)
```

2. **Dataflow Analysis**

Bounds checking on every load from an array is expensive and we would like to avoid it when we can, without compromising safety. We can move the check to compile time if we know the values used to access the array. For example, given the array load `a[10]`, we can check that it is not less than `0` at compile time, thus removing the lower bound check. If we also statically know the size of the array, we can also move the upper bound check to compile time. We will design a dataflow analysis to compute all the constants that reach the use of an array. Here is one example, but the questions below pertain to any control-flow graph:



a) Let $\top$ mean a variable may have any value and $\perp$ mean we do not care because the code is never executed. The other values are sets of constants. Sketch or describe the ordering of values given the constants 1, 2, and 3. Hint: $\{1\}$ cannot be compared to $\{2\}$, but $\{1\} < \{1, 2\}$.



b) Describe the transition rules through assignment statements and from predecessors to successors that hold for any program.

Answer:

- $C(s, x, in) = \text{lub}\ \{C(p, x, out) \mid \text{p is a predecessor of s}\}$
- $C(s, x, out) = \perp$ if $C(s, x, in) = \perp$
- $C(x := c, x, out) = \{c\}$ if c is a constant
- $C(x := e, x, out) = \top$ where e is an expression that is not a constant
- $C(y := \ldots, x, out) = C(y := \ldots, x, in)$ if $x \neq y$

c) How many passes will this dataflow analysis require over the source code, in the worst case, if 1, 2, and 3 are the only constants that appear in the code after all optimizations are executed? Recall, for example, that the global liveness analysis algorithm in class required at most two passes before reaching a stable state.

Answer: 3

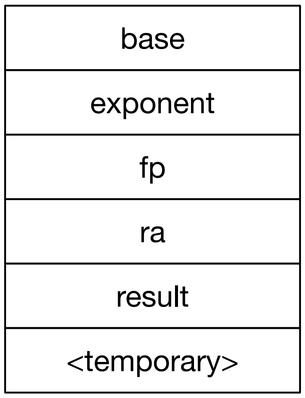
3. **Activation Records**

Consider the following Cool method definition:

```
exp(base: Int,  exponent: Int) : Int {
  let result: Int <- 1 in {
    while 0 < exponent loop {
      result <- result * base;
      exponent <- exponent - 1;
    } pool;
    result;
  }
};
```

What is the activation record of this method? Assume all temporaries are stored in the activation record and minimize storage. List all values, their location, and the total size of the record.

Answer:

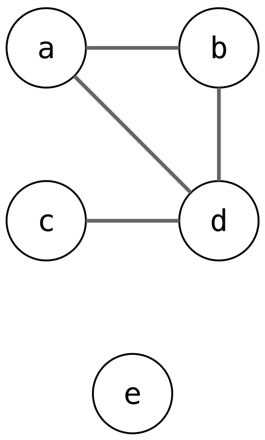


Total size: 24 bytes.

Note that the temporary is necessary to store the result of the comparison, given the requirement that all temporary values must be in memory. We also allow a slot for the self-object, however, it is not strictly necessary, because no attributes are accessed.

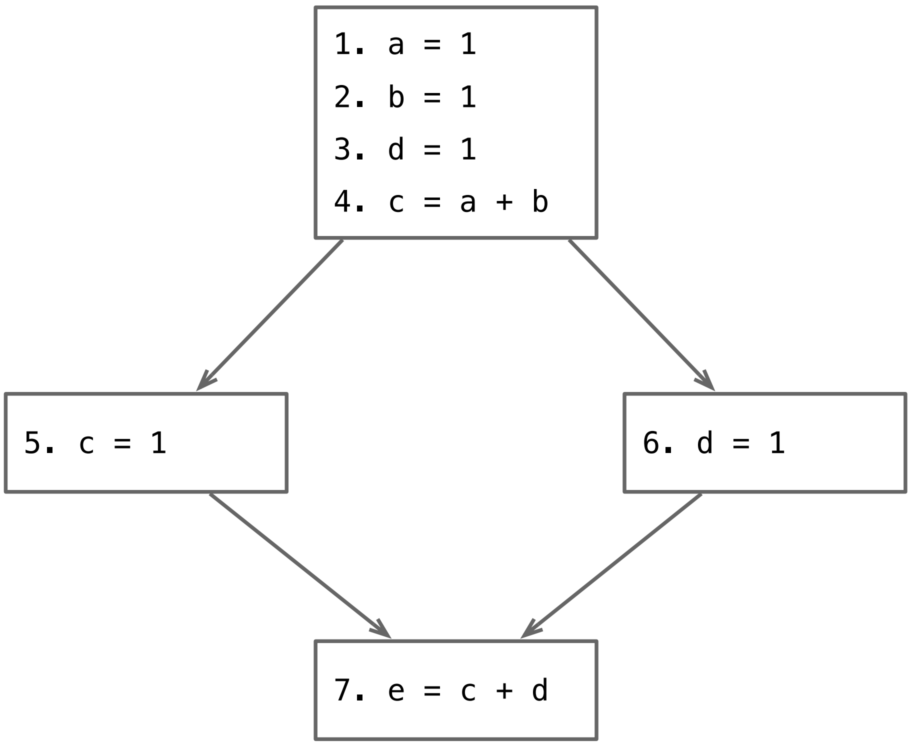
4. **Register Allocation**

Consider the following register inference graph:



a) Fill in all 7 statements in the below control-flow graph, so that it results in the register interference graph. Use only statements of the form `x = 1` and `x = y + x`, where `x`, `y`, and `z` are variables, and assume only `e` is live on exit.

Answer:



```
1. a = 1
2. b = 1
3. d = 1
4. c = a + b
```

```
5. c = 1
```

```
6. d = 1
```

```
7. e = c + d
```

Note that several other solutions are possible, as long as they produce the given register inference graph.

b) Using the graph coloring heuristics in lecture 16, give the smallest number of colors k that enable the heuristic to complete without spilling. If there are multiple nodes that could be deleted from the graph, break ties by first selecting a node with the fewest neighbors and second by choosing the node whose label is first in alphabetical order. Using your provided k, give the state of the stack when all nodes have been deleted from the graph.

Answer:

Value of k: 3

Top of stack (pushed last)

d
b
a
c
e

Bottom of stack (pushed first)

c) Provide the register allocation by listing the variables assigned to each register (named `r1`, `r2`, `r3`, `r4`, `r5`, `r6`, and `r7`). Use the minimal number of registers.

Answer:
r1: e, c, a
r2: b
r3: d