CS 143 Final
Spring 2023

- Please read all instructions (including these) carefully.
- There are 5 questions on the exam, some with multiple parts. You have 180 minutes to work on the exam.
- The exam is open note. You may use laptops, phones and e-readers to read electronic notes, but not for computation or access to the internet for any reason other than to access the class webpage.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. Do not write on the back of exam pages or other pages.
- 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.

SUNET ID: _____

NAME: _____

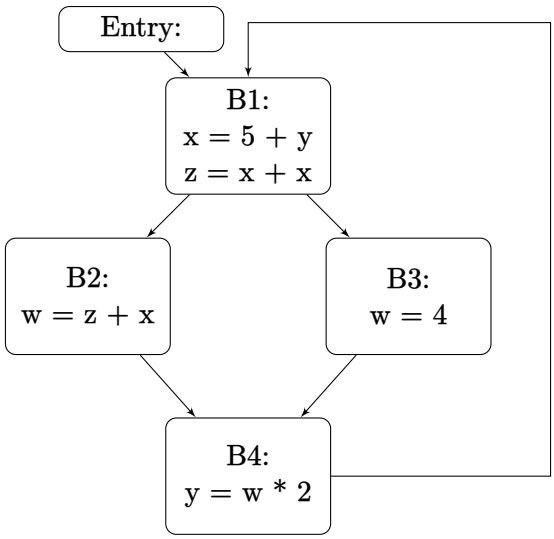
In accordance with both the letter and spirit of the Honor Code, I have neither given nor received assistance on this examination.

SIGNATURE: _____

Problem	Max points	Points
1	20	
2	20	
3	15	
4	15	
5	10	
6	20	
TOTAL	100	

1. Dataflow Analysis

In this problem, you will develop a dataflow analysis that computes whether a variable is always defined at a particular point in a program. More precisely, we are interested in knowing whenever a variable x is used in a program statement s, whether x is defined on all program paths that lead to s. As an example, consider the following control-flow graph:



In this graph,

- The use of x in B1 and B2 are always defined.
- The use of w in B4 is always defined.
- The use of y in B1 is not always defined.

The analysis computes for each program point p and variable x whether x is always defined at p. The function A refers to the results of the "always-defined" analysis. Specifically, $A(s, x, out)$ refers to its result for variable x at the program point just after statement s, and $A(s, x, in)$ refers to its result for x at the program point just before s.

(a) How many values can A take at a given program point? Give names to those values.

Answer:

A can take one of two values: `defined` or `undefined`.

(b) What value should all program points be initialized to?

Answer:

Initialize all points to `undefined`.

(c) What is the value of $A(x := e, x, \text{out})$ for any expression e?

Answer:

x is `defined` at the program point just after being written to, since writing to a variable defines it.

(d) How do you compute the value of $A(x := e, x, \text{in})$ for any expression e?

Answer:

x is `defined` if for all predecessors p of s, A(p, x, out) = `defined`. Otherwise it is `undefined`.

2. Code Generation

In this question, we introduce a short-circuiting AND operator to the Cool programming language. We first augment the Cool expression grammar as follows:

$$
\begin{aligned}
\mathit{expr} ::= & \ldots \\
& \mid \mathit{expr}\ \textbf{and}\ \mathit{expr}
\end{aligned}
$$

We define that the expression e_1 **and** e_2 should have the same meaning as the Cool expression:

```
if e1 then
    if e2 then true else false fi
else
    false
fi
```

Notice that under this definition, the **and** operator *short-circuits*: if e_1 turns out to be **false**, then e_2 should not be evaluated at all. As a more concrete example, if crash() is a function that always crashes, then "**false and** crash()" should evaluate to **false** and *not* crash.

(a) Give the formal operational semantic rule(s) to describe the runtime behavior of the new **and** operator, in the style of Cool manual §13.4. Use no more than two rules.

Your answer must be independent of the **if**-expression, meaning that you cannot merely rewrite the **and**-expression into an equivalent **if**-expression.

Answer:

$$\frac{so, S_1, E \vdash e_1 \mapsto \mathit{Bool}(\mathit{true}), S_2 \quad so, S_2, E \vdash e_2 \mapsto v_2, S_3}{so, S_1, E \vdash e_1\ \textbf{and}\ e_2 \mapsto v_2, S_3} \qquad \frac{so, S_1, E \vdash e_1 \mapsto \mathit{Bool}(\mathit{false}), S_2}{so, S_1, E \vdash e_1\ \textbf{and}\ e_2 \mapsto \mathit{Bool}(\mathit{false}), S_2}$$

(b) Using the format of lecture 12, write down the MIPS code generation function *cgen* for the **and**-expression.

In addition to the short-circuit behavior, the convention for Boolean-valued expressions is: if e is an expression that evaluates to a Boolean value, then after $cgen(e)$, the register **$a0** contains 0 if e evaluates to false or 1 if e evaluates to true.

Your cgen function should contain no more than 6 lines. Once again, your answer must be independent of the **if**-expression, meaning that you cannot merely rewrite the **and**-expression into an equivalent **if**-expression.

Answer:

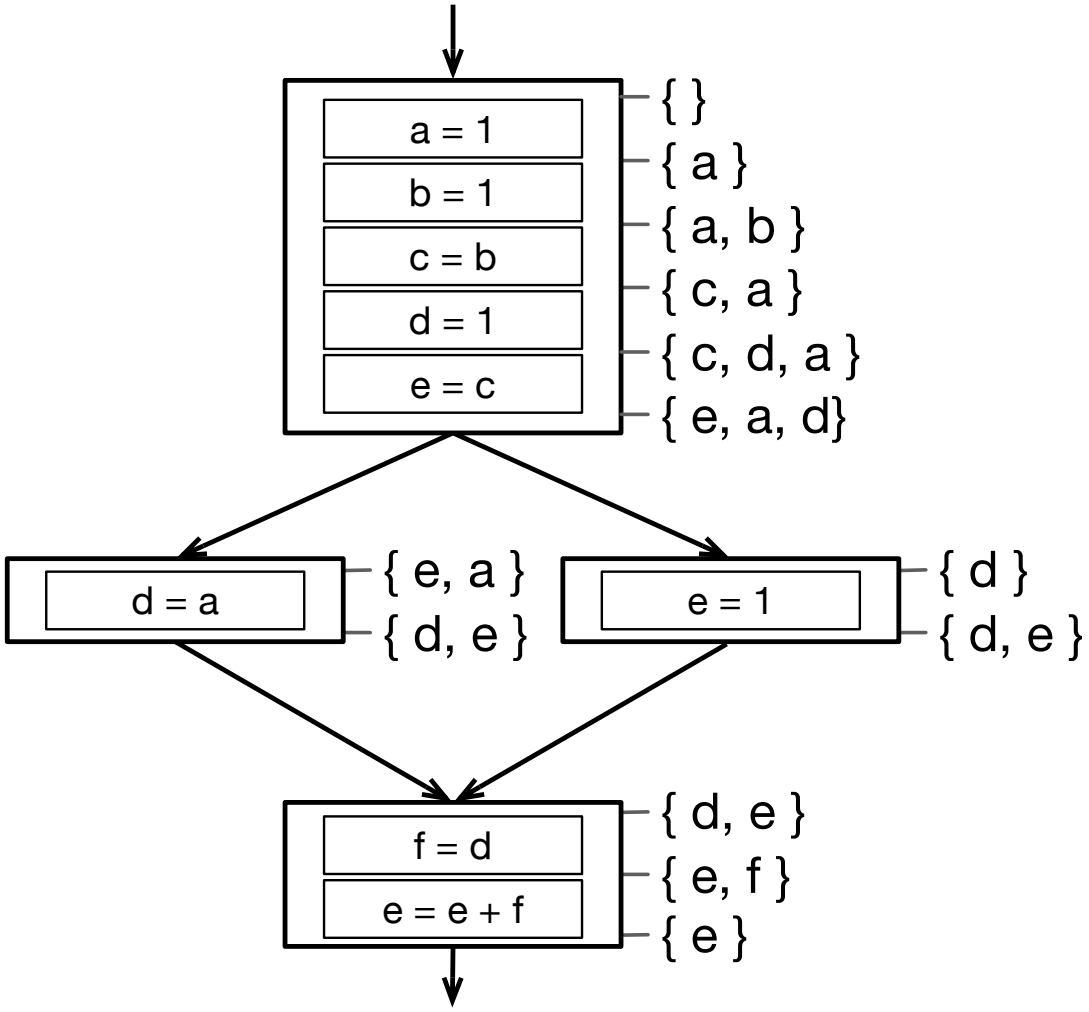
$cgen(e_1$ **and** $e_2) =$
    $cgen(e_1)$
    beq $a0 $zero end
    $cgen(e_2)$
  end:

3. Register Allocation

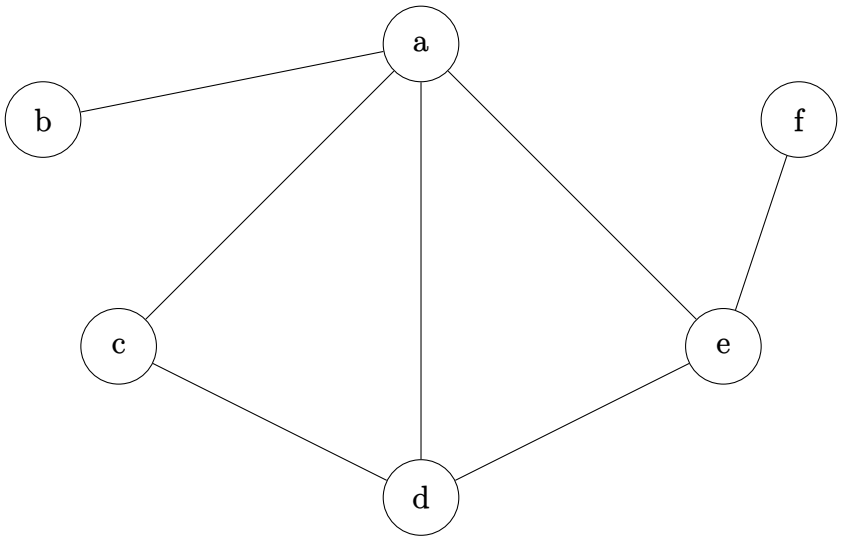
(a) You are given the following control flow graph that uses variables a to f. The program statements have been replaced by empty boxes. Above/below every such box we give the set of variables that are live right before/after the corresponding statement.

Fill in program statements in the boxes, so that the given live sets are consistent with the code. Program statements can take the form `x = 1`, `x = y`, or `x = y + z`, where `x`, `y`, and `z` can be any of the variables used by the control flow graph.

If more than one statement would work in some position, pick the simplest one (prefer `x = 1` over `x = y` over `x = y + z`).



(b) Fill in the following register interference graph by drawing edges between nodes where applicable. Assume only **e** is live on exit of the component of the program corresponding to the above control-flow graph.



(c) 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.

VALUE OF k: 3

Top of stack (pushed last)

e
d
c
a
f
b

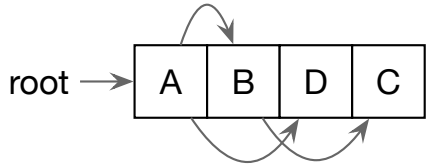
Bottom of stack (pushed first)

4. Garbage Collection

In this question, we will compare the three garbage collection techniques we discussed in class: mark-and-sweep, stop-and-copy, and reference counting. When we refer to garbage collection, we include both the garbage collection phases of mark-and-sweep/stop-and-copy and the memory freeing when the reference count of a reference counted object drops to zero.

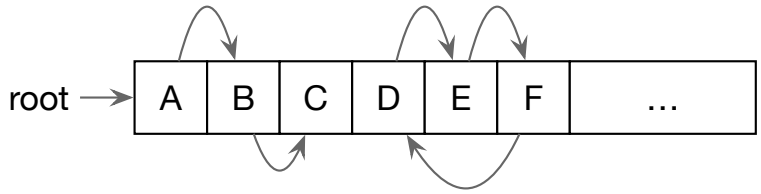
Assume we have allocated objects A to F in lexicographical order and that they were placed at the beginning of the heap in order with no gaps. The objects A to F were allocated before any garbage collection, but other objects may have been allocated later. Garbage collection has run one or more times and garbage has just been collected. Only one register (`root`) points to the heap and no stack values point to the heap.

(a) Given the above assumptions, what garbage collection algorithm(s) may result in the following heap? Please explain why in one sentence.



Answer: Stop-and-copy. The objects C and D have been swapped and stop-and-copy is the only garbage collection technique that can move objects.

(b) Given the above assumptions, what garbage collection algorithm(s) may result in the following heap? Please explain why in one sentence.



Answer: Reference counting. There is an unreachable cycle and reference counting is the only technique that can leave unreachable objects after garbage collection.

(c) Ignore the cost of allocation and marking. Calculate how much faster or slower mark-and-sweep is over stop-and-copy if 8% of objects survive garbage collection and it is 20 times more expensive for stop-and-copy to copy/scan an object than it is for mark-and-sweep to sweep it.

Answer:

Let N be the number of of objects on the heap. The sweep phase of mark-and-sweep steps through all the objects, at a cost of cN, where c is the cost of sweeping one object. Stop-and-copy only needs to copy/scan the 8% surviving objects at a cost of $20c$ per object. Thus, the cost of stop-and-copy is $0.08n \cdot 20c = 1.6cN$, making mark-and-sweep 1.6 times faster.

5. Language Design

In this question we are going to explore an alternative for loop to the one we saw in WA4. The [For-Str] loop iterates through a string one character at a time. Like COOL while loops, the for loop evaluates to void.

$$
\begin{aligned}
&\textit{expr} ::= \ldots \\
&\quad \mid \textbf{for}\ \text{ID}\ \textbf{in}\ \textit{expr}\ \textbf{loop}\ \textit{expr}\ \textbf{pool}
\end{aligned}
$$

The [For-Str] loop iterates through each character in the string provided by the first expression. Thus, in each iteration, ID is a String containing a single character. We assume that ID has been defined before its use in the for loop.

(a) Give type checking rule for the [For-Str] loop construct.

Answer:

$$
\frac{\begin{array}{l} O, M, C \vdash ID : \textit{String} \\ O, M, C \vdash e_1 : \textit{String} \\ O, M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \textbf{for}\ \text{ID}\ \textbf{in}\ e_1\ \textbf{loop}\ e_2\ \textbf{pool}\ : \textit{Object}}\ [\text{For-Str}]
$$

(b) Let us extend Cool so that we can inherit from base classes. Thus, the following code would be permitted:

```
class A inherits String {
    wordCount():Int { ... };
    firstWord():String { ... };
};
```

Would the typing rule for [For-Str] change if inheritance from basic classes in Cool was permitted? Either explain why the rules would stay the same or give the new type rules for [For-Str].

Answer:

Yes, the typing rule for [For-Str] would change. The new rule is shown below.

$$\frac{\begin{array}{l} O, M, C \vdash ID : String \\ O, M, C \vdash e_1 : T_1 \\ \qquad T_1 \leq String \\ O, M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \textbf{for ID in } e_1 \textbf{ loop } e_2 \textbf{ pool } : Object} \text{ [For-Str]}$$

6. Runtime Organization

Consider the following COOL class and method definitions

```
1  class Main {
2      main() : Object {
3          let a: Int ← 4, b: Int ← 5 in bar(a,b)
4      };
5
6      foo(z: Int) : Int {
7          let c: Int ← 7 in {
8              8 + baz(z,c,false);
9          }
10     };
11
12     bar(x: Int, y: Int) : Object {
13         if x+3 = foo(y+1) then
14             baz(x,y,true)
15         fi
16     };
17
18     baz(d: Int, e: Int, f: Bool) : Int {
19         if f then
20             if d − e = 0 then
21                 1
22             else
23                 d + e
24             fi
25         else {
26             d ← d + 1;
27             e + f;
28         }
29     };
30 };
```

A compiler with an unknown runtime management strategy is used to compile the following Cool code. In particular, the strategy may not be exactly one that you have seen before. A partial listing of the stack at a certain program point in the execution is given below. There are three kinds of entries on the stack: a return address, a stack address, and a reference to an object on the heap (e.g., Int(4) or Bool(false)).

(a) Fill in the missing entries in the table.

Address	Contents
100	OS frame pointer
104	OS return address
108	Int(4)
112	Int(5)
116	100
120	Int(4)
124	Int(5)
128	return address of bar(a,b)
132	Int(7)
136	116
140	Int(6)
144	Int(7)
148	return address of foo(y)
152	Int(8)
156	136
160	Int(7)
164	Int(7)
168	Bool(false)
172	return address of baz(z,c,false)

(b) What is the first program point in the execution of the program that could have this stack content? Give the program point by listing the line number of the last statement that executed.

Answer:

The first program point consistent with the stack is the point just after line 26. Since address 160 contains `Int(7)` the incrementing assignment must have executed.

(blank page for extended answers)