CS143 Final
Spring 2018

- Please read all instructions (including these) carefully.
- There are 6 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.
- 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.

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	10	
2	20	
3	20	
4	20	
5	30	
6	20	
TOTAL	120	

1. **Type Soundness** (10 points)
   Cool does not allow attributes to be redefined. Consider removing this restriction as follows: If class D inherits from class C, and class C defines attribute x to have type T_1, then D may redefine x to have any type T_2 such that $T_2 \leq T_1$. Consider the following code, which type checks using this unsound rule.

```
1         class A { };
2         class B inherits A{ foo() : Int {1}; };
3         class C { a : A <- new A; helper(t:A):A {a <- t};};
4         class D inherits C { a : B <- new B;
5                              caller():Int {a.foo()};};
6         class Main {
7           d:D <- new D;
8           c:C <- d;
9           main(): Int {{
10            c.helper(new A);
11            d.caller();
12          }};
13        };
```

   - On which line of code does this program crash when run? Explain why in no more than one sentence.

     The program crashes on line 5. The crash happens because `a` has dynamic type `A` at this point (from the previous call to `helper`) and thus has no `foo` method.

   - Change one token in the program so that the modified program type checks using the unsound rule but runs successfully to completion anyway. Indicate your change directly on the code above.

     On line 10, change the `A` in `new A` to `B`.

2. **Type Checking and Operational Semantics** (20 points)

Below is a full specification for type checking attributes in Cool, including the complete definition of helper functions used by the type checking rules:

$P(C)$ = The parent of class C.
$Attr(C) = [a_1 : T_1, ..., a_n : T_n]$ is the list of attributes textually declared in class C.
$Extend(O, C) = O[T_i/a_i]$ for every $a_i : T_i$ entry in $Attr(C)$
$Context(Object) = \emptyset$
$Context(C) = Extend(Context(P(C)), C)$ if $C \neq Object$
$O_C = Context(C)$

$$\frac{\begin{array}{l} O_C(x) = T_0 \\ O_C[\texttt{SELF\_TYPE}_C/self], M, C \vdash e_1 : T_1 \\ T_1 \leq T_0 \end{array}}{O_C, M, C \vdash x : T_0 \leftarrow e_1;} \quad \text{[Attr-Init]}$$

$$\frac{O_C(x) = T}{O_C, M, C \vdash x : T} \quad \text{[Attr-No-Init]}$$

(a) Attributes in Cool are *protected*, meaning they are visible to child classes but not outside of the class. A *private* attribute is visible only within the class where it is declared and is not visible even to child classes. We extend Cool with private attributes by using the syntax:

$$feature ::= \textbf{private}\ \text{ID} : \text{TYPE}[\leftarrow expr]$$

Give modified type checking rules for private attributes. Show the definitions of any new or modified helper functions used by your rules.

Define three new functions:
$ProtAttr(C) = [a_1 : T_1, ..., a_n : T_n]$ lists the attributes in the class that are protected (not private).
$ProtExtend(O, C) = O[T_i/a_i]$ for every $a_i : T_i$ entry in $ProtAttr(C)$
$ProtContext(Object) = \emptyset$
$ProtContext(C) = ProtExtend(ProtContext(P(C)), C)$
Redefine:
$Context(C) = Extend(ProtContext(P(C)), C)$
We can now use the same type checking rules with these new definitions.

(b) Give a succinct explanation for why no changes are required to the operational semantics of Cool to implement private attributes (i.e., at runtime private attributes are treated just like protected attributes).

Every program that executes correctly under the new type checking rules would also execute correctly under the old type checking rules, so no runtime changes are needed.

3. **Type Checking and Operational Semantics** (20 points)
Consider adding pairs to Cool. A pair is a tuple of length 2 where the elements can have different types:

$$Type ::= ... \mid (Type, Type)$$
$$expr ::= ... \mid \textbf{new } (expr, expr) \mid \textbf{first}(expr) \mid \textbf{second}(expr)$$

A `new`(e_1,e_2) expression allocates and initializes a pair using the values of the two expressions e_1 and e_2. For an expression e that evaluates to a pair, `first`(e) evaluates to the first component of the value of e and `second`(e) evaluates to the second component of the value of e.

(a) Give type checking rules for these three constructs.

$$\frac{O, M, C \vdash e_1 : T_1 \quad O, M, C \vdash e_2 : T_2}{O, M, C \vdash \textbf{new } (e_1, e_2) : (T_1, T_2)}$$

$$\frac{O, M, C \vdash e : (T_1, T_2)}{O, M, C \vdash \textbf{first}(e) : T_1}$$

$$\frac{O, M, C \vdash e : (T_1, T_2)}{O, M, C \vdash \textbf{second}(e) : T_2}$$

(b) Give operational semantics rules for these three constructs. Assume that pairs are a new kind of runtime value. If v_1 and v_2 are Cool values, then (v_1, v_2) is a pair value. Note that a value is not allocated in the heap—you don't need to allocate memory locations to construct a pair value.

$$\frac{so, S, E \vdash e_1 : v_1, S_1 \qquad so, S_1, E \vdash e_2 : v_2, S_2}{so, S, E \vdash \textbf{new } (e_1, e_2) : (v_1, v_2), S_2}$$

$$\frac{so, S, E \vdash e : (v_1, v_2), S'}{so, S, E \vdash \textbf{first}(e) : v_1, S'}$$

$$\frac{so, S, E \vdash e : (v_1, v_2), S'}{so, S, E \vdash \textbf{second}(e) : v_2, S'}$$

4. **Optimization** (20 points)
An important optimization that we did not discuss in class is *method inlining*: a function call is replaced by the body of the function. That is, given a method call

$$e_1.f(e_2)$$

and the definition of the method

$$f(x : T) : T'\{e_3\}$$

we replace the entire method dispatch by

$$\textbf{let } x : T \leftarrow e_2 \textbf{ in } e_3$$

and thereby eliminate the method call. Now assume that to use method inlining for method f, we replace *all* calls to f by the body of f as outlined above. Also assume that we never do method inlining for a recursive method.

(a) How does method inlining make programs faster: What computation is saved by inlining?

Inlining eliminates all the code associated with the calling convention.

(b) Name one cost of method inlining: How can it make programs worse?

Inlining all occurrences of a method will usually make the program larger.

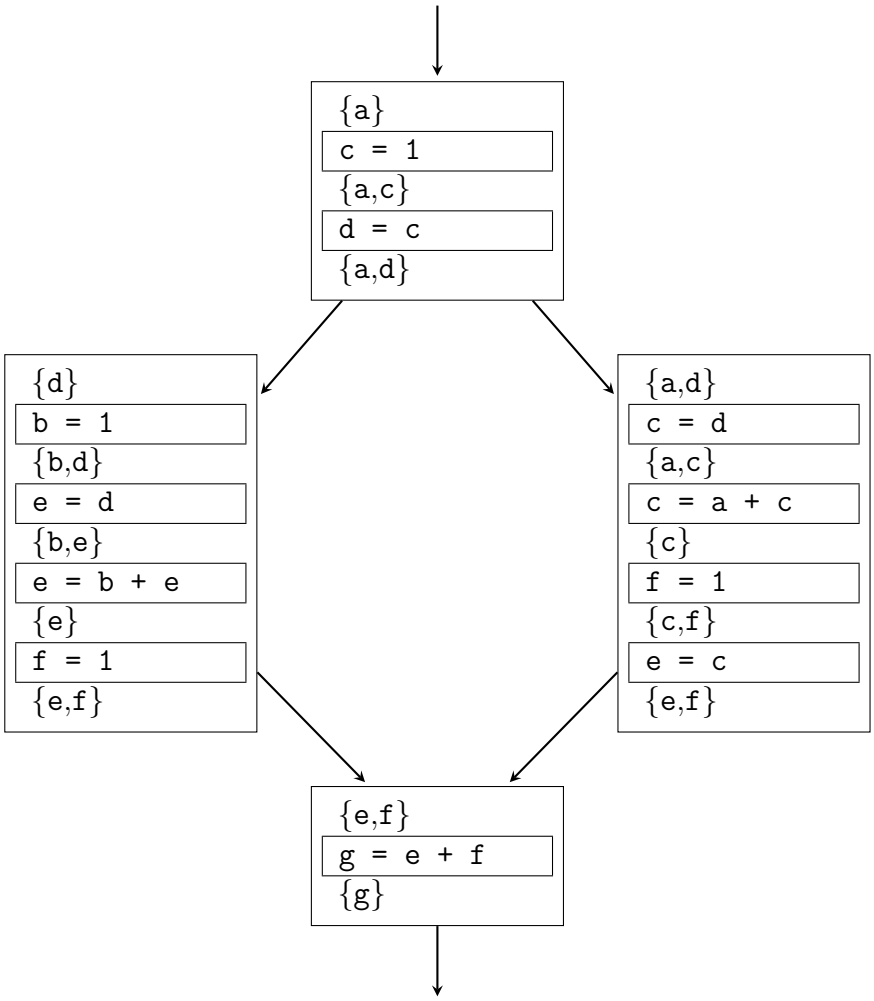
(c) What are the best candidates for method inlining? That is, what characteristic(s) of methods indicate that the cost/benefit ratio of using method inlining is good for a particular method call?

Small methods are good for inlining, they eliminating the most overhead for the minimum increase in code size.

5. **Liveness and Register Allocation** (30 points)

(a) Consider the following control-flow graph where the set of live variables at each point of the program are provided for you. Variable `a` is live upon entry and `g` is live upon exit. Listed from simplest to most complex, program statements can take one of the following forms: `x = 1`, `x = y` and `x = y + z`, where `x, y, z` are variables. Variables can be used more than once in a statement. Do not assign to dead variables.

Provide code in each of the boxes that is consistent with the computed set of live variables at every point of the program. When several program statements may work, pick the simplest among them.



(b) Give the register interference graph of the control flow graph in part (a) as an adjacency matrix by filling in the matrix below. Mark an **X** at entry (i, j) iff there is an edge between variables i and j in the RIG. You may also draw out the graph for your convenience, but we will only grade the adjacency matrix.

	a	b	c	d	e	f	g
a			X	X			
b				X	X		
c						X	
d							
e						X	
f							
g							

(c) Using the graph coloring heuristic provided in lecture 16, give the smallest number of colors k that enables the heuristic to succeed without spilling. If there are multiple nodes that could be deleted from the graph, break ties first by 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)

f
e
b
d
c
a
g

Bottom of stack (pushed first)

(d) Provide the register allocation by listing the variables assigned to each register in the table below. Use only the first k registers, where k is the number of colors you used for part (c).

Register	Variable
r1	a, b, f, g
r2	c, d, e
r3	
r4	
r5	
r6	
r7	

6. **Code Generation** (20 points)
The following is a valid Cool program:

```
class Main inherits IO {
  main() : Int {
    let a: Int in {
      a <- 1;
      while in_int() = 0 loop {
        a <- a * 3 + 1;
        out_int(a);
      } pool;
      a;
    }
  };
};
```

Below is MIPS assembly that implements this program, with some parts missing. We have also omitted the prototype objects and init labels since they are not needed to answer the question. Assume that `Int(x)` is the label of the integer constant `x`. Fill in the blanks (there are 5 lines in total) in the following assembly so that it correctly implements the program above.

```
Main_dispTab:
        .word   Object.abort
        .word   Object.type_name
        .word   Object.copy
        .word   IO.out_string
        .word   IO.out_int
        .word   IO.in_string
        .word   IO.in_int
        .word   Main.main
# ...
Main.main:
        addiu   $sp $sp [-28]
        sw      $fp 12($sp)
        sw      $s0 8($sp)
        sw      $ra 4($sp)
        addiu   $fp $sp 16
        move    $s0 $a0
        la      $a0 Int(1)
        sw      $a0 8($fp)
label0:
        move    $a0 $s0
        lw      $t1 8($a0)
        lw      $t1 24($t1)
        jalr    $t1
```

```
        sw      $a0 12 ( $fp )
        la      $t2 Int(0)
        lw      $t1 12($fp)
        la      $a0 bool_const1
        beq     $t1 $t2 label2
        la      $a1 bool_const0
        jal     equality_test
label2:
        lw      $t1 12($a0)
        beq     $t1 $zero label1
        lw      $t2 8($fp)
        sw      $t2 12($fp)
        la      $a0 Int(3)
        jal     Object.copy
        lw      $t2 12($a0)
        lw      $t3 12($fp)
        lw      $t1 12($t3)
        mul     $t1  $t1  $t2
        sw      $t1 12($a0)
        sw      $a0 12($fp)
        la      $a0 Int(1)
        jal     Object.copy
        lw      $t2 12($a0)
        lw      $t3 12($fp)
        lw      $t1 12($t3)
        add     $t1 $t1 $t2
        sw      $t1 12 ( $a0 )
        sw      $a0 8($fp)
        lw      $t3 8($fp)
        sw      $t3 0($sp)
        addiu   $sp $sp -4
        move    $a0 $s0
        lw      $t1 8($a0)
        lw      $t1 16($t1)
        jalr    $t1
        b       label0
label1:
        lw      $a0 8($fp)
        lw      $fp 12($sp)
        lw      $s0 8($sp)
        lw      $ra 4($sp)
        addiu   $sp $sp 28
        jr      $ra
```