CS143 Final Solution
Spring 2014

- Please read all instructions (including these) carefully.
- There are 6 questions on the exam, each with multiple parts. You have 180 minutes to work on the exam.
- The exam is open note and open laptop, but no Internet connectivity or computation may be used—you can use your laptop to read electronic notes, but nothing more.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. Please do not use any additional scratch paper.
- 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	20	
2	20	
3	20	
4	20	
5	20	
6	20	
TOTAL	120	

1. **Code Generation and Runtime Organization** (20 points)
   For each of the following modifications to Cool, explain how you can generate more efficient code than for the standard definition of Cool. Each part is a single change to the full Cool language (the changes are not cumulative). Give a short (2-3 sentence) explanation of how you would modify code generation and why the generated code would be faster.

   - Disallow method override—methods cannot be redefined in subclasses.

     Without method override, there is no need for dynamic dispatch. The called method can be statically determined and so there is no need for method tables: the compiler can always determine which method will be called in a dispatch expression and simply generate code to call that method.

   - Disallow recursive method calls. No method can call itself, directly or indirectly (i.e., no recursive or mutually recursive methods).

     Since only one instance of a method can be executing an any given time, there is no need for a runtime stack. All local variables of all methods can be statically allocated in the static data area.

   - Disallow inheritance. Classes do not inherit from other classes.

     Without inheritance there is no subtyping—static and dynamic types become the same. Thus, the case expression is trivial (really, useless) as we can statically tell which branch will match and generate code just for that branch. Likewise, static dispatch becomes trivial. Since disallowing inheritance also disallows method override, method calls are also simplified as in the answer to the first part.

2. **Register Allocation** (20 points)
Give a program with the minimum number of statements that satisfies the following conditions:

   - All statements are of the form `a` = `b` + `c` where `a`, `b` and `c` are program variables (and note, they need not be literally just the names `a`, `b` and `c`, but any variable names).
   - Assume that only the variable on the left-hand side of the last assignment is live on exit from the program.
   - The register interference graph of the program has 4 nodes and all possible edges (i.e., it is the complete graph of 4 nodes).

   Do not be concerned with the values of the variables (in particular, don't worry about how or whether they are initialized). Explain why your solution is the shortest possible program.

```
a = a + b
d = d + c
```

The live set before the first instruction is $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$, so the RIG will be fully connected.

To see that it is not possible to answer the question with a single assignment statement, observe that a statement with 2 arguments can increase the number of live variables by at most 2. If there is only one variable live on exit, then a single statement program can have at most 3 live variables before that statement. This is sufficient to show that at least two statments are required, but the problem statement also specifies that the only variable that is live on exit from the code is the last one assigned, so in fact there can be at most 2 live variables before the last assignment statement. Thus, any program that has fully connected RIG of size 4 must have at least two statements.

3. **Garbage Collection** (20 points)
Consider a programming language with the following grammar:

$$
\begin{array}{rcl}
P & \rightarrow & S; P \\
 & | & \epsilon \\
S & \rightarrow & \texttt{VAR} = \texttt{new} \\
 & | & \texttt{VAR} = \texttt{VAR.1} \\
 & | & \texttt{VAR} = \texttt{VAR.2} \\
 & | & \texttt{VAR.1} = \texttt{VAR} \\
 & | & \texttt{VAR.2} = \texttt{VAR}
\end{array}
$$

A program is a sequence of statements. A `new` statement allocates a pair, an object with two fields named "1" and "2". The two fields are initialized to null pointers. The next two statements are field reads from "1" and "2" respectively, and the last two statements are field writes, again to "1" and "2" respectively. `VAR` is a variable name. Now consider the following program fragment.

```
x = new
y = new
y.1 = x
z = new
z.2 = y
w = new
w.1 = y
v = z.2
z.1 = z
x.2 = v
```

For both of the following questions, assume that when a garbage collection is invoked, the root set is the set of live variables and that only the variable `w` is live on exit from this program fragment.

- Assume a Stop and Copy garbage collection takes place after the last statement. Show the state of the local variables and heap after the collection.

```
x -> p1 (dead)
p1.1 -> null
p1.2 -> p2
y -> p2 (dead)
p2.1 -> p1
p2.2 -> null
z -> invalid pointer (points to collected memory)
v -> p2 (dead)
```

```
w -> p4
p4.1 -> p2
p4.2 -> null
```

- At what point in the program would a Mark and Sweep collection result in the largest set of retained (uncollected) objects? If there are multiple points that result in the same size set of retained objects, choose the last one. Show the heap that results from running a collection at this point.

  The latest point for a collection that results in the maximum heap is before the last use of z in the second-to-last statement. At this point all objects in the heap are reachable from live variables and the full heap is retaine

```
x -> p1  (live)
p1.1 -> null
p1.2 -> null
y -> p2  (dead)
p2.1 -> p1
p2.2 -> null
z -> p3  (live)
p3.1 -> nil
p3.2 -> p2
v -> p2  (live)
w -> p4  (live)
p4.1 -> p3
p4.2 -> null
```

4. **Dataflow Analysis** (20 points)
Consider the following program:

```
a = 1
b = 2
c = 3
e = b
d = b
f = a
while * {
    if * then
         if * then
             e = a
             f = c
             d = e
         else
             e = d
             f = c
             d = b
    else
         if * then
             f = e
             e = b
             f = d
         else
             /* nothing */
 }
```

You should assume the omitted predicates marked with * are sometimes true and sometimes false. Otherwise, they play no role in this problem.

- Recall that constant propagation as presented in lecture is done for a single variable at a time. Assuming constant propagation can be done as many times as desired (including repeating it for some variables if necessary), mark the variable-variable assignments that can be replaced by variable-constant assignments in the program above by crossing out the right-hand side and writing in the corresponding constant. Briefly (one or two sentences) justify your answer.

```
a = 1
b = 2
c = 3
e = b -> 2
d = b -> 2
```

```
f = a -> 1
while * {
    if * then
         if * then
             e = a -> 1
             f = c -> 3
             d = e -> 1
         else
             e = d
             f = c -> 3
             d = b -> 2
    else
         if * then
             f = e
             e = b -> 2
             f = d
         else
             /* nothing */
 }
```

The values of a, b and c are never updated, so all uses of those variables can be converted to the corresponding constants. The only other use that is reached by a single constant is the use of e in the first basic block inside the loop.

- Give a shortest sequence of constant propagation analysis passes that will result in all possible constants being propagated in this program. It is sufficient to give the order of the variables analyzed.

  a,b,c,e is one possible order. Other permutations of these four variables are also correct answers, provided that the propsagation of e comes after the propagation of a.

- Modify the right-hand side of just one assignment operation so that as many additional constants can be propagated as possible. On the copy of the original program below, mark which statement is modified and what the new right-hand side is. Then show all the additional statements (beyond the answer to the first part) where constants can be propagated and what those constants are by crossing out the right-hand side and writing in the corresponding constant.

```
a = 1
b = 2
c = 3
e = b 2
d = b 2
```

```
    f = a 1
    while * {
        if * then
             if * then
                 e = b ->  2  /* changed assignment */
                 f = c 3
                 d = e ->  2
             else
                 e = d ->  2
                 f = c 3
                 d = b 2
        else
             if * then
                 f = e ->  2
                 e = b 2
                 f = d ->  2
             else
                 /* nothing */
     }
%
```

5. **Types and Runtime Organization** (20 points)
Consider the following class declarations:

```
class X {
  foo() : Foo { ... };
  bar() : Int { ... };
  b : Bool;
  c : String;

};


class Y inherits X {
  bar() : Int { ... };
  baz() : Object { ... };
  d : Int;

};


class Z inherits Y {

  foo() : Foo { ... };
  qux() : Int { ... };
  a : Object;

};
```

- Give the object layouts and dispatch tables for each of the three classes. You may ignore the inherited methods of the Object class.

```
X:

X class tag
5
X dispatch pointer
attribute b
attirbute c

X's dispatch table (acceptable to omit the Object methods)
Object.abort
Object.type_name
Object.copy
X.foo
X.bar


Y:
Y class tag
6
Y dispatch pointer
attribute b
attribute c
attribute d

Y's dispatch table:
Object.abort
Object.type_name
Object.copy
X.foo
Y.bar
Y.baz


Z:
Z class tag
7
Z dispatch pointer
attribute b
attribute c
```

```
attribute d
attribute a

Z's dispatch table:
Object.abort
Object.type_name
Object.copy
Z.foo
Y.bar
Y.baz
Z.qux
```

- Consider the definition of method `foo()` in class `Z`. Cool requires that the declared return type of an overriden method be exactly the same as the return type of the original method. If instead we allow `foo`'s return type `foo()` : `Bazz` to be some class such that `Bazz` $\leq$ `Foo`, can a program that typechecks crash due to trying to access a method or attribute that does not exist? If your answer is no, argue why not. If your answer is yes, sketch a (short) example program that has this behavior.

  This change preserves type safety. Consider a dispatch

  `e.foo()`

  Now consider the possible dynamic types of the expression `e`. If `e` has static type `X` or `Y`, then the static return type will be `Foo`. If `e` has dynamic type `X` or `Y` then the `foo` method in class `X` is called an everything works as usual. If `e` has dynamic type `Z` then `Z`'s `foo` method is called, which returns something compatible with `Bazz`, which is also compatible with `Foo` since `Bazz` $\leq$ `Foo`.

- Still considering the definition of `foo` in class `Z`, consider the case where we allow `foo` to have the declared return type `foo()` : `Bazz` where `Foo` $\leq$ `Bazz`. Can a program that typechecks under this rule crash due to trying to access a method or attribute that does not exist? If your answer is no, argue why not. If your answer is yes, sktech a (short) example program that has this behavior.

  This change violates type safety. Consider a dispatch

  `(let x:  X <- new Z in x.foo()).foo_only_method()`

  where `x` has static type `X` and dynamic type`Z`. Then the static return type is `Foo` but the dynamic return type can be `Bazz`, which may lack some of the methods and attributes of `Foo` since `Foo` $\leq$ `Bazz`.

6. **Code Generation** (20 points)
Below is the (partial) output of a Cool compiler. Show Cool source code that could compile to this assembly. Just show the fragment corresponding to this code (you do not need to show a complete Cool program). Please write your answer on this page.

```
int_const0:
        .word   2
        .word   4
        .word   Int_dispTab
        .word   0
        .word   -1
bool_const0:
        .word   3
        .word   4
        .word   Bool_dispTab
        .word   0
        .word   -1
bool_const1:
        .word   3
        .word   4
        .word   Bool_dispTab
        .word   1
```

```
Main.f:
        addiu   $sp $sp -16
        sw      $fp 16($sp)
        sw      $s0 12($sp)
        sw      $ra 8($sp)
        addiu   $fp $sp 4
        move    $s0 $a0
        sw      $s1 0($fp)
        la      $s1 int_const0
        lw      $a0 20($fp)
        lw      $t1 12($s1)
        lw      $t2 12($a0)
        la      $a0 bool_const1
        blt     $t1 $t2 label2
        la      $a0 bool_const0
label2:
        lw      $t1 12($a0)
        beqz    $t1 label0
        lw      $s1 12($s0)
        lw      $a0 16($fp)
        jal     Object.copy
        lw      $t2 12($a0)
        lw      $t1 12($s1)
        add     $t1 $t1 $t2
        sw      $t1 12($a0)
        sw      $a0 12($s0)
        b       label1
label0:
        la      $a0 int_const0
label1:
        move    $a0 $s0
        lw      $s1 0($fp)
        lw      $fp 16($sp)
        lw      $s0 12($sp)
        lw      $ra 8($sp)
        addiu   $sp $sp 24
        jr      $ra
```

Answer:

```
Class Main {

  a: Int;

  f(x: Int, y: Int): SELF_TYPE
   {{ if 0 < x then
         a <- a + y
      else
         0
      fi;
      self; }};
  ...
};
```