CS143 Final
Fall 2008

- Please read all instructions (including these) carefully.
- There are 5 questions on the exam, all with multiple parts. You have 2.5 hours to work on the exam.
- The exam is closed book, but you may refer to your four sheets of prepared notes.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. 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	30	
2	36	
3	12	
4	30	
5	12	
TOTAL	120	

1. **Arrays** (30 points)

We would like to add single-dimensional arrays to Cool. We introduce a new type 'TYPE[]', representing an array of values of type TYPE, where TYPE is a type identifier (Object, Int, and so forth). Array types TYPE[] can be used as the type of a class attribute, function formal argument, or the variable introduced by a `let`. We need two new expressions to create arrays and to access them, given respectively by the following grammar:

```
e ::= ... | new T[e] | e0[e1]
```

These arrays have a special property: Once initialized, the array's contents are immutable. While methods on the array cell objects can be called (possibly modifying the object's attributes), the array cells themselves cannot be changed to point to new objects.

An example program using arrays is as follows:

```
class A {
  f:Int;
  get_f():Int { f };
};

class B inherits A {
  g:Int;
};

class Main {
  foo():Int {
    let x:B[] <- new B[3] in
      bar(x, 0)
  };

  bar(x:A[], n:Int):Int {
    x[n].get_f()
  };
};
```

- Give subtyping rules for array types. Your rules should allow the call in the `foo()` method in the example above to be type checked (note that `x` has different types in `foo` and `bar`). You need not consider `SELF_TYPE` in your rules.

- Give a set of typechecking rules for the `new T[e]` and `e0[e1]`. expressions. As with the previous part, your rules should allow the example program to be type checked, and you need not consider `SELF_TYPE`.

- What runtime checks, if any, need to be added when generating code for the `new T[e]` and `e0[e1]` expressions?

- Describe a memory layout for representing arrays on the heap. Show how this layout is used for the result of the `new B[3]` expression in the `foo()` method of the example.

- Is your array memory layout compatible with the Cool garbage collector? Why or why not?

2. **Register Allocation and Optimization** (36 points)

Here is a simple C function:

```
void foo(int A[], int B[], int a, int b) {
      int i, c;
      for (i = 0; i < a + 3; i++) {
            c = A[2 * i];
            A[i] = a + c;
            b = c;
            B[i] = 2 * i + b;
      }
      A[0] = a + 3 + b;
}
```

and here is it's intermediate representation:

```
      i = 0
L:    t0 = a + 3
      if (i >= t0) goto M
      t1 = 2 * i
      c = A[t1]
      t2 = a + c
      store A[i], t2
      b = c
      t3 = 2 * i
      t4 = t3 + b
      store B[i], t4
      i = i + 1
      goto L
M:    t5 = a + 3
      t6 = t5 + b
      store A[0], t6
```

- Given the intermediate representation of `foo`, perform any necessary analysis to come up with a register interference graph. In your analysis, ignore the array variables `A` and `B` and focus instead on `a`, `b`, `c`, `i`, and the temporaries. You do not need to draw a register interference graph. Instead, please fill out the following table:

Node:	Edges To:
a	
b	
c	
i	
t0	
t1	
t2	
t3	
t4	
t5	
t6	

- For this graph, is it possible to assign variables to registers using the k-coloring algorithm in class for k = 4 without spilling? Why or why not? If it is possible, give a valid 4-coloring for the graph.

- The intermediate representation can be optimized. Show the result of common sub-expression elimination, then copy propagation, and then dead code elimination on `foo`.

- Assume we want to extend register allocation to double word values, such as 64-bit floating-point numbers. Assume any pair of registers can be used to hold a double word value. What changes, if any, need to be made to the construction of the register interference graph and the coloring heuristic?

- Consider any optimization that reorders the instructions in a control-flow graph without adding or removing any instructions (instruction scheduling is an example). Can reordering instructions (assuming the new order preserves program semantics) be detrimental to register allocation? Explain briefly why or why not.

3. **Memory Management and Runtime Organization** (12 points)
Some compilers do not use a stack but instead allocate everything in the heap. Consider modifying a Cool compiler so that the stack is a singly-linked list of activation records. When a method is called a new activation record r is allocated in the heap. A field *prev* of the activation record is set to point to the previous activation record and the stack pointer is set to point to r; when the method returns the stack pointer is loaded with the field *prev* to "pop" the stack.

- Consider the total work done to allocate, push, pop, and deallocate one activation record if the garbage collector is mark and sweep, and consider the total work to allocate/pop/push/deallocate one activation record using a standard contiguous stack. Do you think the speed of the heap-allocated stack will be generally faster, slower, or about the same performance as the contiguous stack, and why?

- The same question, using a stop-and-copy collector.

4. **Code Generation and Operational Semantics** (30 points)
Consider the following simple string manipulation language:

```
S -> next | replace c | again | c ? S, S | S; S
```

A program is a statement `S`. During execution, the program has a *position* in the string. If i is the current position, each of the statements has the following effect:

- `next` advances the position to `i + 1`;
- `replace c` replaces the ith element of the string by `'c'` (note that `'c'` can be any character) and leaves the position unchanged;
- `again` restarts execution of the program from the beginning and the position in the string is unchanged;
- `c`?$\mathtt{S}_1, \mathtt{S}_2$ executes statement $\mathtt{S}_1$ if the character at position i is `'c'` and executes statement $\mathtt{S}_2$ otherwise.
- $\mathtt{S}_1; \mathtt{S}_2$ executes $\mathtt{S}_1$ and then statement $\mathtt{S}_2$;

The program terminates when either the position moves to one past the last character of string or the program executes the last statement. For example, the program

```
x ? replace y, next; again
```

replaces all occurrences of the character `'x'` by `'y'` in a string.

- In the style of Cool operational semantics inference rules, give an operational semantics for this language. Your rules should have the form

$$E \vdash S \rightarrow s, i$$

  which says that in state (or environment) E the execution of statement S results in the string s and new position i.
  - Define the environment E—list what things need to be in the environment and explain (in English) what each of these things is.

– For each of the language constructs, give an operational inference rule defining its semantics.

```
next
```

```
replace c
```

```
again
```

```
c ? S1, S2
```

```
S1; S2
```

- Give a code generation function `cgen(S)` that takes as an argument a statement in the language and produces psuedo-assembly code. You may add additional arguments to `cgen` but you should clearly state what these arguments are. You may also use any reasonable and clear assembly programming notation. Assume that the string is represented as a contiguous block of bytes, that the machine is byte-addressable (i.e., you can load and store individual bytes), and that initially register `r1` has a pointer to the first character of the string and register `r2` has a pointer to first byte after the last character of the string.

```
cgen(next,              ) =
```

```
cgen(replace c,         ) =
```

```
cgen(again,             ) =
```

```
cgen(c ? S1, S2,        ) =
```

```
cgen(S1; S2,             ) =
```

5. **Miscellaneous Short Answer** (12 points)
For the following questions we are looking for *short* answers of 1 to 2 sentences.

- Function inlining replaces a function call with the body of the called function (note that variables may need to be renamed to guarantee that variables in the inlined function do not collide with variables in the calling function). What is an example of an optimization that is likely to be improved by function inlining, and why?

- What is a syntax-directed translation?

- What is a buffer overrun and how can it lead to a security vulnerability?