This sample midterm is the midterm from Spring 2008. If you see any errors in the sample answer, please email me or tell me on Piazza. Good luck on your midterms!

1. “Ireland has leprechauns galore.” is an example of a particular kind of syntactic construct in English. Can you construct a similar example in C++, OCaml, or Java? If so, give an example; if not, explain why not

- Artificial languages are generally more carefully-designed than natural languages. Thus, these kinds of exceptions should be rare, if they exist at all.

2.

a) Write an OCaml function ‘twice’ that accepts a function f and returns a function g such that g(x) equals f(f(x)). For simplicity’s sake, you can assume that f is free of side effects, and you can impose other restrictions on f and x. Try to keep the restrictions as minor as possible, and explain any restrictions you impose. Or, if ‘twice’ cannot be written easily in OCaml, explain why not.

- let twice f x = f(f(x))
- restriction is that f is ‘a -> ‘a

b) Same as a) except write a function ‘half’ that accepts a function f and returns a function g such that f(x) equals g(g(x)).

- It is difficult, since its implementation must depend on the specific function f used.
- For example, if f is identity function f(x) = x, then g is easy-- also the identity function.
- However, if f is the sine function, then g is super difficult-- given x, g(g(x)) = sin(x)???
- The restriction, like a), is that f is ‘a->’a

c) Give the types of ‘twice’ and ‘half’

- twice: (‘a->’a) -> ‘a -> ‘a
- half: (‘a -> ‘a) -> ‘a -> ‘a

3. Consider the following grammar for a subset of the C++ language.

expression:
    expression ? expression : expression
    expression != expression
    expression + expression

```
! expression
INTEGER-CONSTANT
( expression )
```

For example, (!! 0+1 != 2 ? 3 : 4) is read as “if not-not-0 plus 1 does not equal 2, then 3 else 4, and evaluates to 4.

a) What are the tokens of this subset of C++?

```
? : != + ! INTEGER_CONSTANT ( )
```

b) Show that this grammar is ambiguous

Draw the two parse trees for expression 1+2+3

c) Rewrite the grammar so that it is no longer ambiguous, resolving any ambiguities in the same way that c++ does. Recall that in C-+, the expression

(0 != 1 != 2 || 3 + !4 + 5 || 6 ? 7 : 8 ? 9 : 10) is like

(((((0 != 1) != 2) || ((3 + (!4)) + 5)) || 6) ? 7 : (8 ? 9 : 10))

```
E -> E2 ? E2 : E | E2
E2 -> E2 != E3 | E3
E3 -> E3 + E4 | E4
E4 -> !E4 | E5
E5 -> INTEGER-CONST | ( E )
```

d) Translate the rewritten grammar into a syntax diagram

eg. E2

```
o------> E3 ----->o
    |<-- != <--|
```

4. A numerical analyst is really bothered by the special value of IEEE floating point, and asks you to modify ~~Java~~ C++ to fix what she view as a serious conceptual flaw. She wants her Java programs to throw an exception instead of returning infinities and NaNs. Is her request reasonable for ~~Java~~ C++ programs? Is it implementable? Why/why not? Don’t worry about compatibility with existing compilers, etc.; assume that you are the inventor of ~~Java~~ C++ and she is asking for this feature early in your language design process.

- It's kind of reasonable...maybe the numerical analyst has applications that don't allow Infs and NaNs. However, this gets rid of the choice of having these special values. To implement it, ~~Java~~ C++ can throw an exception whenever it encounters a special value.

5. Give an example of four distinct ~~Java~~ C++ types A, B, C, D such that A is a subtype of B, A is a subtype of C, B is a subtype of D, and C is a subtype of D. Or, if such an example is impossible, explain why not.

- class D {...}
- class A: public B, public C {...}
- class B: public D {...}
- class C: public D {...}

6. Explain how you would implement OCaml-style type checking, in an implementation that uses dynamic linking heavily. What problems do you foresee in programs that relink themselves on the fly?

- (verbatim from the professor) The problem is that one needs to do type checking as well as the usual name checking that link editing normally does. And the type checking needs to follow OCaml's rules. In particular it needs to work with generic types, where the type of the linked-to function does not exactly match the type needed by the caller, but the types will match after applying a unifier.

7.
a) Write a curried OCaml function "interleave C S L1 L2" that constructs a new list L from the lists L1 and L2, using the chooser C with seed S, and returns a pair (S1,L) where S1 is the resulting seed and L is the interleaved list. For example, "interleave C S [1;2] [3;4;5]" might invoke C four times and then return (S1, [1;3;4;2;5]). At each step of the iteration, "interleave" should use the choose to decide whether to choose the first item of L1 or the first item of L2, when deciding which of the two items to put next into L. If L1 is empty, the choose need not be used, since "interleave" will just return L2; and similarly if L2 is empty, "interleave" should just return L1 without invoking C.

Chooser C is defined
http://www.cs.ucla.edu/classes/spring08/cs131/hw/hw1.html

"interleave" should invoke C a minimal number of times, left to right across the lists L1 and L2. "interleave" should avoid side effects; it should be written in a functional style, without using OCaml libraries.

```
let rec interleave c s l1 l2 =
if l1 = [] then (s,l2) else if l2 = [] then (s,l1) else
match (c s) with
| (true, s1) -> (match l1 with
                    | h::t -> (let (s2,l) = (interleave c s1 t l2) in
                                              (s2, h::l)))
| (false, s1) -> (match l2 with
                    | h::t -> (let (s2,l) = (interleave c s1 l1 t) in
                                              (s2, h::l)))
```

b) Write a function “outerleave” that does the opposite of what “interleave does”: it splits a list into two sublists that can be interleaved to get the original list, and returns a triplet consisting of the new seed and the two sublists. That is, “outerleave C S [1;3;4;2;5]” might yield (S1, [1;3;2], [4;5]). If given a list of length N, “outerleave” always invokes the chooser N times.

```
let rec outerleave c s l =
match l with
| [] -> (s, [], [])
| h::t -> match (c s) with
                    | (true, s1) -> (let (s2, l1, l2) = (outerleave c s1 t) in
                                                (s2, h::l1, l2))
                    | (false, s1) -> (let (s2, l1, l2) = (outerleave c s1 t) in
                                                (s2, l1, h::l2))
```

c) Give the data types of all top-level values or functions defined in your answer to a) and b).

- interleave: ('a -> bool * 'a) -> 'a -> 'b list -> 'b list -> 'a * 'b list
- outerleave: ('a -> bool * 'a) -> 'a -> 'b list -> 'a * 'b list * 'b list
- chooser c: 'a -> (bool, 'a)
- seed s: 'a
- list l1/l2/l: 'b list