Sample questions for midterm 2
CS 421, Spring 2009

1. Recall that in lectures 11 and 12 we gave several translation schemes for expressions and statements: [e] produces a pair containing the code for e and the location of the result; [S] gives the code for S; $[e]_x$ gives the code to evaluate e and put its value in x; $[e]_{Lt,Lf}$ translates a Boolean expression e to code that branches to either Lt or Lf (this is called the "short-circuit evaluation scheme"), and $[S]_L$ translates S in a context in which L is the label for a break statement.

a. Generate code for the repeat-until statement: "repeat S until e" executes S and tests e, and repeats until e becomes true. Thus, it is equivalent to "S; while !e do S". Do this in two ways: (i) Using the regular scheme [e] to evaluate the condition; and (ii) Using the short-circuit evaluation scheme for e.

```
(i)     let (I, t) = [e]
            L1, L2 = new labels
        in
           L1:  [S]
                I
                CJUMP t, L1, L2
           L2:

(ii)    let (I, t) = [e]
            L1, L2 = new labels
        in
           L1:  [S]
                [e]_{L2,L1}
           L2:
```

b. Generate code for a multiple assignment statement: (x1, x2) = (e1, e2), which does both assignments "in parallel." Note that this is not that same as doing one assignment followed by the other, because variables x1 and x2 may appear in expressions e1 and e2. Use evaluation scheme $[e]_x$ where appropriate.

```
let t = new variable
in      [e1]_t
        [e2]_x2
        x1 = t
```

c. Give two schemes for conditional expression e1 ? e2 : e3, which gives the value of e2 if e1 is true, or e3 if e1 is false. The two schemes you should provide are (a) the standard [e1 ? e2 : e3], and (b) the assignment scheme $[e1 ? e2 :e3]_x$. You should use the short-circuit evaluation scheme for e1.

```
[e1 ? e2 : e3] =
  let (I_i, t_i) = [ei], i=2,3
      t = new variable
      L1, L2, L3 = new labels
  in    (       [e1]_{L1,L2}
        L1:     I_2
                t = t_2
                JUMP L3
```

```
[e1 ? e2 : e3]_x =
  let L1, L2, L3 = new labels
      in              [e1]_{L1,L2}
                L1:   [e2]_x
                      JUMP L3
                L2:   [e3]_x
                L3:
```

L2: $\mathbf{I_3}$
$\mathbf{t = t_3}$
L3: , t)

2. (a) Name the two parts of a compiler's front end.

Lexing (or lexical analysis) and parsing (or syntactic analysis)

(b) Name the two parts of a compiler's back end.

Optimization (or machine-independent optimization) and code generation (or machine-dependent optimization).

(c) What are the two outputs of the front end?

Abstract syntax tree and symbol table.

3. Name the items in an activation record.

Function arguments
Return address
Dynamic link
Saved registers
Local variables

4. Give two advantages of the copying garbage collection algorithm over the non-copying (mark-and-sweep) algorithm.

1. Copying g.c. takes time proportional to size of reachable data, rather than total size of heap
2. It can improve virtual memory performance by compacting data into fewer pages

5. Give two advantages of the non-copying (mark-and-sweep) garbage collection algorithm over the copying algorithm.

1. Mark-and-sweep does not require that half of memory be reserved at all times
2. It does not require relocation of data and changing of pointers; in some cases, it may be difficult to change all pointers into the heap.

6. Reference counting is not a popular algorithm. What drawback of this algorithm is the reason?

Cannot handle circular structures.

7. In APL, define multmat n which gives an nxn matrix where position i,j has the value i*j.

```
multmat 4;;
1 2 3 4
2 4 6 8
3 6 9 12
4 8 12 16
```

APL notation:

```
(³m) × m „ n nριn
```

APL-in-OCaml notation:

```
let multmat n = let m = rho (n ^@ n) (indx n)
    in m *@ (trans m);;
```

8. Define the following OCaml functions. *[Exam: we will provide definitions of fold_right and fold_left.]:*

(a) repeat_until: ('a -> bool) -> ('a -> 'a) -> 'a -> 'a. where repeat_until p f x = x, if p x, or f x if p (f x), or f (f x) if p (f (f x)), etc.

```
let rec repeat_until p f x =
    if p x then x else repeat_until p f (f x);;
```

(b) sift: (‘a -> bool) -> ‘a list -> ‘a list * ‘a list. sift p lis splits lis into a pair of lists (lis1, lis2), with lis1 containing those elements of lis that satisfy p and lis2 the others.

```
let rec sift p lis = match lis with
        [] -> ([],[])
      | (x::xs) -> let (lis1,lis2) = sift p xs
          in if p x then (x::lis1, lis2) else (lis1, x::lis2);;
```

(c) Write sift using fold_right. Specifically, define sift_base and sift_rec so that
fold_right (sift_rec p) lis sift_base = sift p lis

```
let sift_rec p x (xs, ys) = if p x then (x::xs,ys) else (xs,x::ys);;
let sift_base = ([],[]);;
```

(d) Write an OCaml function that reverses a list, using *fold_right* instead of explicit recursion.

```
let rev l = fold_right (fun x -> fun y -> y @ [x]) l []
```

(e) Write a function f such that map f lis returns a list that contains the absolute values of the elements in lis, in the same order. Do not use any library functions in the definition of f.

```
f = fun x -> if x < 0 then -1 * x else x;;
```

(f) Using fold_right and no explicit recursion, define a function that concatenates the elements of a string list.

```
let concat lst =  fold_right (fun x y -> x^y) lst "";;
```

(g) compose_all [f1;f2;…] a = f1 (f2 (… (fn a)…)). Define compose_all *and* say what its type is.

compose: ('a -> 'a) list -> 'a -> 'a
let rec compose_all flis a =
if flis=[] then a else (hd flis) (compose_all (tl flis) a);;

(h) graph_fun f [x1; x2; ...; xn] = [(x1, f x1); (x2, f x2); ...]. Define graph_fun and say what its type is.
`graph_fun`: $(\alpha \rightarrow \beta) \rightarrow \alpha$ list $\rightarrow (\alpha * \beta)$ list, where

```
('a -> 'b) -> 'a list -> ('a * 'b) list
let rec graph_fun f x =
   if x=[] then [] else (hd x, f (hd x)):: graph_fun f (tl x);;
```

9. What does this OCaml program evaluate to:

```
let x = 4
let y = 6
let f y = x + y
let x = 8
in f(y+x)
```

18

10. Suppose the following Java interface is defined:

```
interface FunObj {
  int apply (int) ;
}
```

A Java class might define a function like this:

```
void map (FunObj f, int[] a) {
  for (int i=0; i<a.length; i++)
        a[i] = f.apply(a[i]);
}
```

a. Define classes **`decrobj`** and **`sqrobj`** that implement **`FunObj`** so that **`map (new decrobj(), a, n)`** decrements each element of **`a`**, and **`map (new sqrobj(), a, n)`** squares each element of **`a.`**

```
class decrobj {
     int apply (int i) {
           return i-1;
     }
}

class sqrobj {
     int apply (int i) {
           return i*i;
     }
}
```

b. Define a class **compose** that implements **FunObj:**

```
class compose implements FunObj {
      FunObj f, g;

      compose (FunObj f, FunObj g) {
            this.f = f; this.g = g;
      }

      int apply (int i) {
            return f.apply(g.apply(i));
      }
}
```

that composes function objects, so that, for example, **map(new compose(new sqrobj(), new decrobj()), a, n)** changes every element a[i] to $(a[i]-1)^2$.