Sample questions for final, CS 421, Spring 2008

These questions cover material that may be on the final, and are at about the level of difficulty of the exam questions. However, there are some differences. We have not debugged them as carefully as the questions on the final; if you have questions, post them in the newsgroup and we'll respond. We also haven't formatted them as carefully as we will on the exam, and have not left space for the solutions. Nor is this intended in any way to be indicative of the length of the exam. Lastly, in several cases, we have not provided some information that we will provide on the exam (*if* we ask a similar question); we indicate in each question when that is the case.

1. 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.

(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.

(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

(d) sublist: 'a list -> int -> int -> 'a list. This function, when applied as sublist lst lo hi, returns the elements of the list lst that are positioned between lo and hi (indexing from zero). The function should return empty set when lo > hi, or when lo is out of the bounds of the input list; if hi is out of bounds, just return the elements from lo to the end of the list.

(e) Write "map" using "fold_left".

(f) app_all [f1;f2;...] a = [f1 a; f2 a; ...]. Define app_all *and* say what its type is.

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

2. Which regular expression differs from the others in its acceptance set? _____

a. a*(b | c)*a*
b. (a | b)*c*a*
c. a*b*c*(b | c)*a*

Give an example of a string that shows the difference (i.e. either is accepted by the odd regular expression and not by the others, or vice versa).

3. Consider the "stratified" expression grammar:

$E \rightarrow T + E \mid T$
$T \rightarrow T * P \mid P$
$P \rightarrow id \mid (E)$

(a) Give the parse tree for: x + (u * v * w)

(b) What associativity of + is implied by this grammar? How about *?

(c) Give a new grammar, based on this one, that includes operator ^ (exponentiation), with higher precedence than *. ^ is right-associative.

4. Three grammars are given below. For each, write whether the grammar is suitable for LL(1) (i.e. top-down/recursive descent) parsing. Explain why.

a)
```
S ::= M + S
M ::= M * id | id
```

b)
```
E ::= if B then A else A | if B then A
A ::= id + A | 1 | 0
B ::= true | false
```

c)
```
L ::= '(' A
A ::= id B | ')'
B ::= ',' id B | ')'
```

5. This question concerns the translation of program to a 3-address intermediate representation. We use an IR like the one we used in class and on the midterm. The instructions are:

```
x = y
x = n
x = &y          (get address of variable y)
x = a op b,     where a and b are names or constants, and op is any arithmetic,
                comparison, or boolean operator
JUMP label
CJUMP  x, Lt, Lf        (jump to Lt if x is true, Lf  otherwise)
x = LOADIND y           (get value in location contained in y)
ERROR                   (return an error)
```

You may use any of the following translation schemes, which were discussed in class:

[S] = instruction list to compute statement S
[e] = pair containing instruction list to compute expression e and variable name giving location of result

[e]$_{tlab,flab}$ = instructions to calculate boolean-valued expression e and jump to tlab if it is true, flab otherwise. (This is called the "short-circuit evaluation" scheme.)

Some languages have multi-level break statements: break *n* breaks out of *n* levels of switch or while statements, e.g.

```
while (...) {
  while (...) {
    ...
    break 2;   // this statement jumps to
    ...
  }
}
// here
```

Thus, a plain "break" is equivalent to "break 1".

The scheme for translation to intermediate form for statements in such a language must know how many while or switch statements the statement being translated is within, and the labels of those containing statements. Thus, the scheme has the form

[S]$_{BL}$**,** where BL is a list of labels, b_1, b_2, ..., b_n. This represents the translation of S to intermediate form, given that it is within *n* while statements, and labels b_1, b_2, ..., b_n are the labels of the instructions that follow those containing statements, from innermost to outermost. (For purposes of this question, ignore switch statements.) That is, a "break 1" in S should jump to b_1, a "break 2" should jump to b_2, etc.

Give translation schemes for the while statement (using the short-circuit scheme for the condition) and for the break *n* statement.

[while (e) S]$_{BL}$ =

[break *n*]$_{BL}$ =

6. 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
```

7. Lambda-calculus *[Exam: we will include a reminder about the definition of Church numerals.]*

(a) Encode the following functions in lambda calculus using Church numerals.

(1) fun x -> x + 1
(2) fun x -> 2 * x

(b) Similar to numerals, booleans can be encoded in lambda calculus:

true = λ x . λ y . x
false = λ x . λ y . y

Give the lambda calculus encoding of the following functions based on the definition of true and false above.

(1) AND, where AND true true = true, and otherwise AND x y = false
(2) NOT.

8. Adam and Mary are two students who do implementation in OCaml. Adam likes quick programming. He wants to save himself from typing extra characters. He defines the following function as a shortcut for if-then-else.

```
let iF b t e = if b then t else e
```

Whenever Adam needs to write if B then T else E, he uses iF B T E instead, making his programs a lot more concise.

Mary doesn't mind typing a few more characters. She prefers to use if-then-else. She writes the following program for a course MP:

```
let f x y = y/x
in let g a b = if a=0 then b else f a b
in g 0 10
```

Adam writes the same program as

```
let f x y = y/x
in let g a b = iF (a=0) b (f a b)
in g 0 10
```

Mary's program runs successfully, but Adam's program does not. Explain Adam's mistake to him.

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)
```

a) using static scope? _____ b) using dynamic scope? _____

10. The following function computes binomial coefficients using "dynamic programming."

```
let rec mkvec m f = if m=0 then [] else f() :: mkvec (m-1) f;;
let binom n k =
   let table = mkvec (n+1) (fun () -> mkvec (k+1) (fun () -> ref (-1)))
   in let rec binom' n k =
          let v = nth (nth table n) k
          in if !v != -1
                then !v
                else (if k=0 or k=n
                      then v := 1
                      else v := (binom' (n-1) (k-1) + binom' (n-1) k);
                     !v)
      in binom' n k;;
```

The function nth gets the nth element in a list, indexing from zero. table is a list of lists, in effect, a two dimensional array of references, and v is table[n,k].

(a) What is the type of table?
(b) What is the type of v?
(c) Show the value of table after calling binom 3 2; that is, show the values to which each element in table refers.

11. Give the full proof tree for the judgment:

{y->5}, let f=fun x->x+y in f y $\Downarrow$ 10

using the environment-based dynamic semantics (lecture 4/3, slides 10–11) *[Exam: we will give the dynamic semantics.]*

12. Some functional languages have a "list comprehension" notation that mimics the mathematical "set comprehension" notation. Specifically, it has the form

[e | x <- e']

where e is an expression containing x and e' is a list-valued expression. The expression returns a list of all the values of e where x takes on all the values in e'. For example [x*x | x <- [1;2;3]] would be the list [1;4;9]. In fact, the list comprehension expression above is exactly equivalent to map (fun x -> e) e'.

(a) Give a type rule for list comprehensions:

$$\frac{}{\Gamma \vdash [\, e \mid x \leftarrow e' \,] : \tau \text{ list}}$$

(b) Actually, list comprehensions have a more general form, namely:

[e | x <- e'; y <- e'']

where e contains x and y, and e'' contains x. Here, x takes on all values in the list e', and for each such value, y takes on all the values in the list e''; then the expression returns the list of values e for all the pairs of x, y values. For example, [(x,y) | x <- [1;2;3]; y <- [x+10:x+20]] returns [(1,11);(1,21);(2,12);(2,22);(3;13);(3;23)]. Give a type rule for this more general form of list comprehension:

$$\Gamma \;\vdash\; [\, e \mid x \leftarrow e';\ y \leftarrow e'' \,] : \tau \text{ list}$$

13. Prove that the following functions terminate by providing a well-founded ordering on the function arguments such that the arguments to any recursive calls are smaller than the arguments to this call. Assume x is an integer, and n is a natural number. (Note: part b is by far the hardest of these three problems.)

(a)
```
let rec exp x n =
        match n with
            0 -> 1
          | 1 -> x
          | _ -> if(n mod 2 = 1)
                 then x * exp x (n-1)
                 else exp (x*x) (n/2);;
```

(b)
```
let rec exp x n =
        match n with
          0 -> 1
        | 1 -> x
        | _ -> if(n mod 2 = 1)
          then exp x (n+1) /x
          else exp (x*x) (n/2);;
```

(c)
```
let rec exp x n =
        match n with
                0 -> 1
              | 1 -> x
              | _ -> x * exp x (n-1);;
```

14. Consider the following program P that computes x to the power n.

```
int a = n-1;
int x = 1, y = 1;
while (a > 0){
  int t = x;
  x = y;
  y = y+t;
  a = a-1;
}
```

The partial correctness judgment for this program is

n>0 {P} y = fib n

Find the loop invariant. Keep in mind that the invariant, together with the negation of condition (a > 0), must imply the post-condition. (Recall that fib n = fib (n-1) + fib (n-2) for n>1, and fib n = 1 otherwise.)