CS421 Spring 2009 Midterm 2

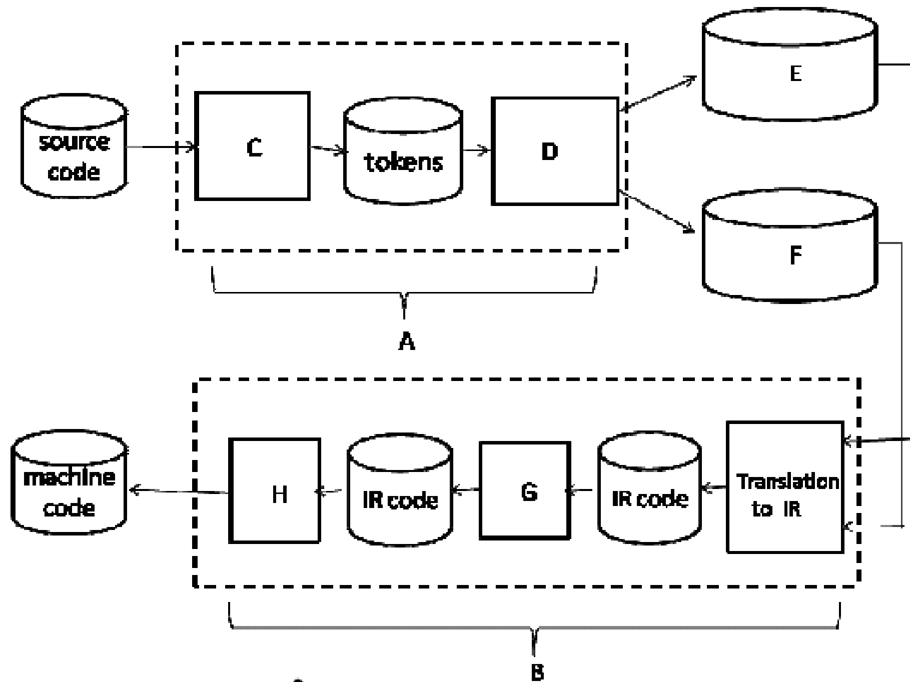
Thursday, April 9, 2009

Name:	
NetID:	

- You have **90 minutes** to complete this exam.
- This is a **closed-book** exam.
- Do not share anything with other students. Do not talk to other students. Do not look at another student's exam. Do not expose your exam to easy viewing by other students. Violation of any of these rules will count as cheating.
- If you believe there is an error, or an ambiguous question, seek clarification from one of the TAs. You must use a whisper, or write your question out.
- Including this cover sheet, there are 10 pages to the exam. Please verify that you have all 10 pages.
- Please write your name and NetID in the spaces above, and at the top of every page.

Question	**Possible points**	**Points earned**	**EC points**
1	8		
2	8		
3	14		
3(b)(iv) (ec)	5		
4	12		
5	16		
6	6		
7	12		
8	12		
9	12		
Total	100 + 5		

1. (8 pts) Fill in the blanks below, giving the names of the various parts of a compiler. (Recall that the cylinders represent data and the boxes represent actions (i.e. functions).)



(A) Front-end

(B) Back-end

(C) Lexer

(D) Parser

(E) AST

(F) Symbol table

(G) Optimization

(H) Code generation

2. (8 pts) Match these terms with *all* applicable statements below. Note that some of the statements below apply to more than one of these terms.

Reference-counting d, e
Non-copying garbage collector a, b, d, f
Copying garbage collector a, c, g

(a) Can handle circular heap structures easily.
(b) Also called "mark-and-sweep."
(c) Can improve virtual memory performance
(d) Uses "free list" representation
(e) Cost of recovering free cells proportional to number of cells freed
(f) Cost of recovering free cells proportional to total number of cells in heap
(g) Cost of recovering free cells proportional to number of reachable cells

3. (14 pts) In class, we gave the following translation schemes for translating source programs into an intermediate representation (IR). All but the first take an AST (expression or statement) to a sequence of IR instructions.

[e] : translate expression e to IR; returns pair (IR instruction list, location of value)

[S] : translate statement S to IR

$[e]_x$: translate expression e to code that stores value of e in variable x

$[S]_L$: translate statement S in context of a loop or switch statement, where L is the target of a break statement

$[e]_{Lt,Lf}$: translate expression e to code that branches to Lt if e is true, or Lf otherwise

The instructions in our intermediate representation were: x = n; x = y; x = y + z (for any operation +); JUMP L; CJUMP v, L1, L2; x = LOADIND y; and JUMPIND x.

(a) Give the following translations. (You may use functions getloc() and getlabel() to get fresh memory locations or fresh instruction labels.)

(i) $[e1 + e2]_x$ = let y = newloc ()
in $[e_1]_y$
$[e_2]_x$
x = y + x
tlet

(ii) $[e1 \mid\mid e2]_{Lt,Lf}$ = (note: use short-circuit evaluation for e1 and e2)

let L = newlabel ()
in $[e1]_{Lt, L}$
L: $[e2]_{Lt, Lf}$

(iii) $[\text{break}]_{L}$ =

JUMP L

(b) There were a few translations we never gave in class. Fill these in:

(i) $[x]_{Lt,Lf}$ = (where x is a variable of type bool)

CJUMP x, Lt, Lf

(ii) $[\text{true}]_{x}$ =

x = true

(iii) $[e1 \&\& e2]_{x}$ = (use $[e1 \&\& e2]_{Lt,Lf}$ to do this translation)

let Lt, Lf, L = newlabels ()
in $[e1 \&\& e2]_{Lt, Lf}$
Lt: x = true
JUMP L
Lf: x = false
L:

(iv) (5 pts extra credit) Here, e2 and e3 are boolean expressions.

$[e1 ? e2 : e3]_{Lt,Lf}$

let L1, L2 = newlabels ()
in $[e1]_{L1, L2}$
L1 : $[e2]_{Lt, Lf}$
L2 : $[e3]_{Lt, Lf}$

(c) Suppose we add the logical xor boolean operation (not bitwise xor, which is ^ in C or Java; there is no logical xor in those languages), and we want to define $[e1 \text{ xor } e2]_{Lt,Lf}$. One way to do this is:

$[e1 \text{ xor } e2]_{Lt,Lf}$ = let L1, L2 = newlabel()
in $[e1]_{L1,L2}$
L1: $[e2]_{Lf,Lt}$
L2: $[e2]_{Lt,Lf}$

However, this scheme requires that we create two copies of the code for e2. To avoid this, we should start by evaluating and storing the value of e1. (Note that, unlike && and ||, xor always evaluates both its arguments.) Complete this translation in a way that avoids duplicating code for e1 or e2:

$[e1 \text{ xor } e2]_{Lt,Lf}$ = let x = newlocation() , L1, L2 = newlabels ( )
in $[e1]_x$
$[e2]_{L1,L2}$
L1: CJUMP x, Lf, Lt
L2: CJUMP x, Lt, Lf

4. (12 pts) Write expressions in APL for the following calculations. You may use either real APL syntax or the syntax you used for MP 8. (The APL reference sheet is included at the end of this exam.)

(a) the sum of the squares of the elements of a vector V

+/ V × V

(b) the product of all positive elements of a vector V

×/ (V > 0) / V

(c) an *n*-by-*n* matrix filled with the number *r*. (*n* and *r* are variables.)

n n ρ r

(d) an *n*-by-*n* matrix filled with 0's below (i.e. southwest of) the diagonal and 1 on and above the diagonal.

⍉ M ≤ M ← n n ρ 1 n

5. (16 pts)

(a) Give the type of the following function: *fun f -> fun g -> fun x -> f (g x)*

$(\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$

(b) Write an OCaml function that reverses a list, using *fold_right* instead of explicit recursion. (Reminder: fold_right has type $(\alpha\rightarrow\beta\rightarrow\beta) \rightarrow \alpha$ list $\rightarrow \beta \rightarrow \beta$.)

```
fold_right (fun x y -> y @ [x]) lst []
```

(c) Use *map* to write a function *map_first f l* which applies *f* to the first element of each item in *l*, assuming that *l* is a list of pairs, e.g. map_first succ [(1,'a'); (2, 'b'); (8, 'z')] = [(2,'a'); (3, 'b'); (9, 'z')]

```
let map_first f l = map (fun (x,y) -> (f x, y)) l
```

(d) Using fold_right and no explicit recursion, define a function maxi lst that returns a pair where the first item of the pair is the maximum element of lst, and the second is the index of that element. Indexing starts from 1. Assume that lst contains distinct positive integers only. The value that should be returned for an empty list is (0,0). Example: maxi [4;7;9;2;7;5] = (9,3)

```
fold_right (fun x (m,i) ->
    if x > m then (x,1) else (m,i+1))
        (0,0) lst
```

6. (6 pts) What is the result (i.e. the value of the call f(x+y) at the end) of the following OCaml program?

```
let y = 5 in
let f = let y= 10 in (fun x -> x + y) in
let x = let y = 20 in y+15 in
f (x+y)
```

50

7. (12 pts) In homework 10A, you defined multisets to be functions of type α → int; in particular, you used type

```
type 'a multset = 'a -> int;;
```

In that homework, you defined functions add, member, union, disjointUnion, intersection, remove, filter, and fromList. Define the following additional functions on multisets:

(a) toSet: 'a multiset -> 'a set, where 'a set is defined by: type 'a set = 'a -> bool

```
let toSet ms = fun n -> (ms n) > 0
```

(b) subtract: 'a multiset -> 'a multiset -> 'a multiset, such that subtract a b has n copies of value x if a has p copies and b has q copies and n = p-q (except that, of course, n cannot be less than zero).

```
let subtract a b =
    fun x -> let n = a x - b x
             in if n<0 then 0 else n
```

(c) addrange: int -> int -> int multiset -> int multiset, where addrange m n a adds one more copy of each of the elements m, m+1, …, n to a.

```
let addrange m n a =
    fun x -> let p = a x
             in if x >= m & x <= n
                then p+1 else p
```

8. (12 pts) In class, we used functions to implement sets. For this problem, we will use a slightly different representation, for sets of *positive* integers, that allows us to enumerate all the elements in a set (although not very efficiently):

```
type finiteintset = (int -> bool) * int * int
```

with functions

```
let member n (f, _, _) = f n;;
let minelt (_, m, _) = m;;
let maxelet (_, _, n) = n;;
```

where m and n will be zero for the empty set (and only for the empty set). To give two examples:

```
let emptyset = fun x -> (false, 0, 0)
fun addelt x (f, m, n) = (fun y -> (y = x) or f y,
                          if m=0 or x<m then x else m,
                          if n=0 or x>n then x else n)
```

(a) Define the following functions (you may use function min and max: int*int -> int):

```
(* union: finiteintset -> finiteintset -> finiteintset *)
let union (f1, m1, n1) (f2, m2, n2)
  = (fun x -> f1 x or f2 x,
     if m=0 or n=0 then max(m,n) else min(m1, m2),
     max (n1, n2))
```

```
(* setToList: finiteintset -> int list *)
let setToList (f, m, n)
  = let rec aux i = if i > n then [ ]
                    else if f i then i :: aux (i+1)
                    else aux (i+1)
    in aux m
```

9. (12 pts) Write a function object for case_map (see the OCaml definition below). For the sake of simplicity, we assume that f : int -> bool, g,h : int -> int.

OCaml definition:

```
let case_map f g h lis = map (fun x -> if (f x) then (g x) else (h x)) lis;;
```

As in the OCaml code, your Java solution should call Map.map, which is given here:

```
interface BoolFun {
    boolean apply (int n);
}
interface IntFun {
    int apply (int n);
}

class Map {
    static int[] map (IntFun f, int lis[]) {
        int lis2[] = new int[lis.length];
        for(int i = 0; i < lis.length; i++)
            lis2[i] = f.apply(lis[i]);
        return lis2;
    }
}

class Case_Map {
    static int[] case_map (BoolFun f, IntFun g, IntFun h, int lis[]) {
        // complete this method
        IntFun fgh = new IntFun () {
            int apply (int n) {
                return f.apply(n)? g.apply(n) : h.apply(n);
        }
        return Map.map(fgh, lis);
    }
}
```

APL Reference

Operation	*Expression*	*Value*
Sample data	`A ; a 2,3-matrix`	1 2 3 4 5 6
	`V ; a 3-vector` `C ; a logical 2-vector`	2 4 6 1 0
	`D ; a logical 3-vector`	1 0 1

Arithmetic	`A *@ A`
	
	`V -@ (newint 1)`
		1 4 9
		16 25 36
		1 3 5

Relational	`A >@ (newint 4)`	0 0 0 0 1 1
Reduction	`!+ V`	12
	`maxR A`	3 6
Compression	`D % V`	2 6
	`C % A`	1 2 3 (a 1,3-matrix)
Shape	`shape A`	2 3
Ravelling	`ravel A` `ravel (newint 1)`	1 2 3 4 5 6 1
Restructuring	`rho (shape A) V`	2 4 6 2 4 6
Catenation	`rho (shape V) C` `A ^@ C`	1 0 1 1 2 3 4 5 6 1 0
Index generation	`indx (newint 5)`	1 2 3 4 5
Transposition	`trans A`	1 4 2 5 3 6
Subscripting	`V @@ (indx (newint 2))`	2 4
	`A @@ (newint 1)`	1 2 3 (a 1,3-matrix)
	`(trans A) @@ (indx (newint 2))`	1 4 2 5