CS 421 Spring 2010 Midterm 2

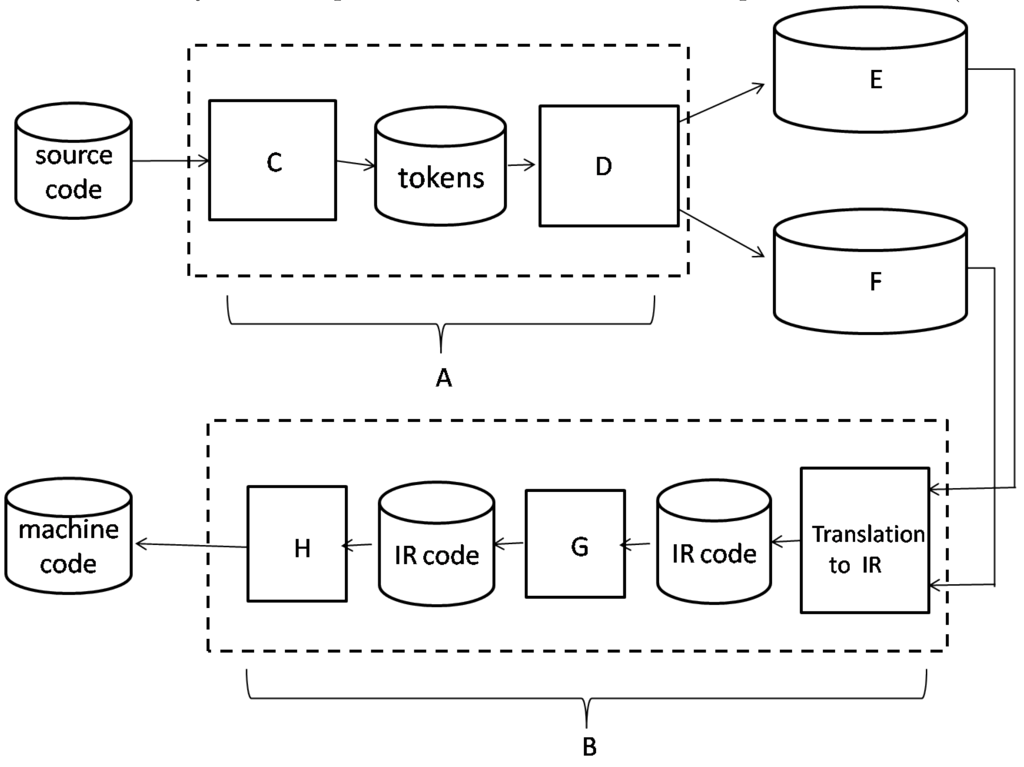
Wednesday, April 7, 2010

Name	
NetID	

- You have **75 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 11 pages to the exam. Please verify that you have all 11 pages.
- Please write your name and NetID in the spaces above, and at the top of every page.

Question	**Value**	**Score**
1	8	
2	12	
3	14 + 5 XC	
4	15	
5	22 + 5 XC	
6	15	
7	14	
Total	**100 + 10**	

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 <u>front-end</u>

B <u>back-end</u>

C <u>lexer</u>

D <u>parser</u>

E <u>AST</u>

F <u>symbol table</u>

G <u>optimization</u>

H <u>code generation</u>

2. (12 pts) For each of the statements below, indicate which memory management approach(es) it describes: reference counting (RC), non-copying garbage collection (NG), or copying garbage collection (CG). If a statement applies to more than one approach, you should write all of the approaches it describes.

   Cannot handle cyclical references

   RC

   Uses a “free area” model to represent free memory

   CG

   Is best for spreading out the cost of garbage collection throughout the program

   RC

   At any time, only half of memory is in use

   CG

   Unreachable memory may not be freed immediately

   NG, CG

   Iterates over the entire heap at once (not just reachable memory)

   NG

   Does not move reachable data

   RC, NG

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 short-circuit evaluation scheme)

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

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

      i. $[e_1 + e_2]$

         let t1, t2, t3 = getloc() in
         $[e_1]_{t1}$
         $[e_2]_{t2}$
         t3 = t1 + t2

      ii. $[e_1 \text{ ? } e_2 \text{ : } e_3]_x$ (for full credit, use the short-circuit scheme for e_1)

         $[e_1]_{L1,L2}$
         L1: $[e_2]_x$
         JUMP L3
         L2: $[e_3]_x$
         L3:

      iii. $[e_1 \&\& !e_2]_{Lt,Lf}$ (e_2 should not be evaluated if e_1 is false)

         $[e_1]_{L1,Lf}$
         L1: $[e_2]_{Lf,Lt}$

(b) (5 pts extra credit) Give IR code for a for loop. A for loop has the form "for(S_1; e; S_2) S_3", where S_1 is executed before the loop begins, the loop ends when e evaluates to false, S_2 is executed at the end of each iteration of the loop, and S_3 is the loop body. For full credit, use the short-circuit scheme for e.

$[S_1]$
JUMP L2
L1: $[S_3]_{L3}$
$[S_2]$
L2: $[e]_{L1,L3}$
L3:

4. (15 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) 3 times the product of the elements of the vector V

   $3 * */V$

   (b) A vector containing only the non-negative elements of the vector V

   $(V \geq 0)\ /\ V$

   (c) An m-by-n matrix filled with the number r

   $(m,n)\ \rho\ r$

   (d) The identity matrix of size n

   $(n,n)\ \rho\ 1,\ n\ \rho\ 0$

   (e) An n-by-n matrix with 0's above the diagonal and 1's on and below the diagonal

   ⍉$m \geq m \leftarrow (n,n)\ \rho\ \iota n$

5. (22 pts)

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

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

   (b) Write an OCaml function *update* such that update f a b is a function that returns b when given a as input but otherwise behaves the same as f.

   let update f a b = fun x -> if x = a then b else f x

   (c) Write an OCaml function *double* that duplicates each element of a list, using fold_right instead of explicit recursion. For example, double [1; 2; 3] = [1; 1; 2; 2; 3; 3]. Remember that fold_right has type (α -> β -> β) -> α list -> β -> β.

   let double lis = fold_right (fun x y -> x :: x :: y) lis []

   (d) Write an OCaml function *sum_pairs* that takes a list of pairs and returns a list containing the sum of the elements of each pair, using map instead of explicit recursion. For example, sum_pairs [(1, 2); (3, 4); (5, 6)] = [3; 7; 11].

   let sum_pairs = map (fun (x, y) -> x + y)

(e) (5 pts extra credit) Write an OCaml function *maxf* that takes a function f and a list lst and returns a pair (max, index), where *max* is the largest value produced by applying f to an element of lst, and *index* is the index in lst of the element x such that f x = max, where the first element of the list has index 0. If there are multiple such elements, you may return the index of any one of them. For example, maxf (fun x -> x + 2) [1; 2; 3] = (5, 2). You may assume that lst is never empty. You may also assume that f takes elements of lst and returns only positive integers. Your function should use fold_right instead of explicit recursion.

let maxf f lst = fold_right (fun x (m, i) -> if f x > m then (f x, 0) else (m, i+1)) lst (0,0)

6. (15 pts) In homework 9, you defined multisets to be functions of type α -> int; in particular, you used the definition `type 'a multiset = '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) fromSet: 'a set -> 'a multiset, such that fromSet s returns a multiset containing 1 copy of each element in s. Recall that the set type is defined by `type 'a set = 'a -> bool`.

   let fromSet s = fun x -> if s x then 1 else 0

   (b) count: 'a multiset -> 'a list -> int, such that count m lst returns the total number of occurrences of elements from lst in m. You may assume that lst contains no duplicate elements.

   let count m lst = fold_right (+) (map m lst) 0

   (c) subtract: 'a multiset -> 'a multiset -> 'a multiset, such that subtract a b has n copies of the value x if a has p copies and b has q copies and n = p - q. If b has more copies of x than a, then subtract a b should have 0 copies of x.

   let subtract a b = fun x -> max (a x - b x) 0

7. (14 pts) Write a function object in Java for the OCaml function *apply_pos*, defined as follows:

   apply_pos f lst = map (fun x -> if x > 0 then f x else x) lst

   For simplicity, we assume that lst is a list of integers. As in the OCaml code, your Java solution should call Map.map, which is given here:

```
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 Apply_Pos {
   static int[] apply_pos (final IntFun f, int lis[]) {
      // complete this method
      IntFun g = new IntFun(){
        int apply(int n){
          return n > 0 ? f.apply(n) : n;
        }
      };
      return Map.map(g, lis);
   }
}
```

APL Reference

Operation	*Expression*	*Value*
Sample data	`A ; a 2,3-matrix`	1 2 3 4 5 6
	`V ; a 3-vector`	2 4 6
	`C ; a logical 2-vector`	1 0
	`D ; a logical 3-vector`	1 0 1
Arithmetic	`A *@ A`	1 4 9 16 25 36
	`V -@ (newint 1)`	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
	`rho (shape V) C`	1 0 1
Catenation	`A ^@ C`	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