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 12 pages to the exam. Please verify that you have all 12 pages.
- Please write your name and NetID in the spaces above, and at the top of every page.

Question	**Value**	**Score**
1	18	
2	15	
3	15 + 5 XC	
4	8	
5	10 + 5 XC	
6	12	
7	12	
8	10	
Total	**100 + 10**	

1. (18 pts) In class, we gave the following translation schemes for translating source programs into an intermediate representation (IR). Each one maps an AST (expression or statement) to a sequence of IR instructions.

   [S] : translate statement S to IR

   $[e]_{t}$: translate expression e to code that stores value of e in variable t

   $[e]_{Lt,Lf}$: translate boolean expression e to code that branches to Lt if e is true, or Lf otherwise (the short-circuit evaluation scheme)

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

   Here are some clauses of these translation schemes. Note that temporary variables (t_i) and program labels (L_i) are created fresh for each use of the translation scheme; we have omitted the calls to "newloc()" and "newlabel()". The last one is an "if-then;" in class, we gave a scheme for "if-then-else."

```
[x]_y =              y = x, if y and x are different variables
                     nothing, if y and x are the same variable

[e1 < e2]_Lt,Lf =    [e1]_t1
                     [e2]_t2
                     t = t1 < t2
                     CJUMP t, Lt, Lf

[e1 && e2]_Lt,Lf =        [e1]_L1,Lf
                     L1 : [e2]_Lt,Lf

[if (e) S] =              [e]_L1,L2
                     L1 :  [S]
                     L2 :
```

   (a) Translate this statement using short-circuit evaluation for the conditional; your answer will include "[S]":

   [if (x < y && y < z) S] =

(b) Give these translation schemes:

$[e_1 \mid\mid e_2]_{\mathrm{Lt,Lf}} =$

$[!e]_{\mathrm{Lt,Lf}} =$

(c) Give a translation scheme for a "do-while" statement: `do` *S* `while(`*e*`)` executes *S* and tests *e*, in that order, until *e* becomes false.

do S while (e) =

2. (15 pts)

   (a) Write an OCaml function *double*, of type $(\alpha \to \alpha) \to (\alpha \to \alpha)$, such that `double f x` is a function that returns 2 times the result of applying `f` to `x`. For example, if `incr` is the function that increments an integer, then `double incr 3` = 8.

   (b) Write an OCaml function `cond_sum`, of type (int $\to$ bool) $\to$ int list $\to$ int, such that `cond_sum f l` returns the sum of the elements of `l` for which `f` is true. For example, `cond_sum (fun x -> x > 0) [1;2;-4] = 3`. You must use `fold_right` instead of explicit recursion. Remember that `fold_right` has type (α `->` β `->` β) `->` α `list` `->` β `->` β.

   (c) Write an OCaml function `apply_all` such that `apply_all [`f_1`; `f_2`; ...; `f_n`] arg` returns `[(`f_1` arg); (`f_2` arg); ...; (`f_n` arg)]`. For example, `apply_all [(fun x -> x * 2); (fun x -> x + 2)] 1 = [2; 3]`. You must use *map* instead of explicit recursion; recall that *map* has type $(\alpha \to \beta) \to \alpha$ list $\to \beta$ list .

3. (15 pts + 5XC) In homework 8, you defined sets to be functions of type `intset = int -> bool`. A *multiset* is a set that can contain multiple copies of an element. Just like sets, multisets are not ordered. We represent multisets with functions, just like sets. A multiset function returns the number of occurrences of the given element. `type multiset = int -> int`. Here are a few functions on multisets:

```
(* count: int -> multiset -> int - number of occurrences of an int in a multiset *)
let count n ms = ms n

(* add: int -> multiset -> multiset - adds another copy of an int to a multiset *)
let add n ms = ms n + 1

(* member: int -> multiset -> bool - say whether an int occurs in a multiset *)
let member n ms = ms n > 0
```

Define these functions on multisets:

(a) `transform: (int->int) -> multiset -> multiset`. `transform f ms` is the multiset that contains the same elements as `ms`, but if `v` is in `ms`, in `transform f ms v` has `f (ms v)` elements. For example, `transform incr ms` adds one occurrence to every element of `ms`. (If `count x ms` is zero, it remains zero.)

(b) `filter: multiset -> set -> multiset`, such that `filter ms s` removes (all copies of) any value from `ms` if that value does not occur in `s`.

(c) `filterout: (int -> bool) -> multiset -> multiset`. `filter f ms` is the same multiset as `ms` except that for any value `v`, if `f v` is true, then it has one fewer occurrence of `v` than `ms` does.

(d) (5 XC) `maxfromlist: intlist -> multiset -> int*int`. `maxfromlist lis ms` returns the element of `lis` that has the most occurrences in `ms`, together with the number of occurrences; if `lis` or `ms` is empty, it returns `(0,0)`. *You must use* `fold_right`*.*

```
let maxfromlist lis ms =

    fold_right
```

4. (8 pts) In class, we gave a definition of the higher-order function `map` in Java, i.e. using function objects. In the following, the final line should increment every element of array A. Fill in the blanks to make this work.

```
abstract class IntFun {
   abstract int apply(int x);
}

class Incr extends IntFun {


   int apply  ---------------------------------------
}

public class Map {
   // Apply f to ever element of A
   static void map(IntFun f, int A[]){
      for(int i = 0; i < A.length; i++)


         --------------------------------------------------
   }

   public static void main(String[] args)
   {
      int[] A = {1,2,3,4};
      map(new Incr(), A);
   }
}
```

5. (10 pts + 5 XC) We write an interpreter for a language with the following abstract syntax:

```
type expr = ...
type stmt = Assign of id * expr    (* id = e *)
| Seq of stmt * stmt               (* s1; s2 *)
| If of expr * stmt * stmt         (* if e then s1 else s2 *)
| While of expr * stmt             (* while e do s *)
| Return                           (* halt the program *)
```

The interpreter consists of two functions: eval, which evaluates expressions, and exec, which executes statements. The interpreter keeps track of the current state:

```
type state = string -> int     (* function from identifier to it's value *)
```

To implement `return`, we need the interpreter to keep track of an extra bit of information, which indicates whether the code we are executing has returned — in which case, we shouldn't execute it. That is, if we have a sequence of statements `Seq(s1, s2)`, then if `s1` returns, `s2` should be ignored. Thus, `exec` has type: `exec: stmt -> state -> (state * bool)`; it executes `stmt` in state `rho` and returns a pair `(rho', returned)` of a new state `rho'` and a boolean `returned`, which is true when the program was ended by a `Return` statement. Here are the first two clauses:

```
let rec exec stmt rho = match stmt with
  | Assign(s, e)  -> (bind s (eval e rho) rho, false)
  | Return        -> (rho, true)
```

(a) (5 pts) Implement the sequencing operator.

```
| Seq(s1, s2)  ->
```

(b) (5 pts) Implement the if operator. (Conditional expressions return 1 for true and 0 for false.)

```
| If(e, s1, s2) ->
```

(c) (5 XC) Implement the while operator.

```
| While(e, s)  ->
```

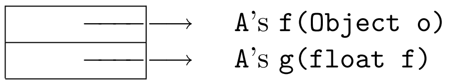
6. (12 pts) Consider the following Java classes:

```
class A
{
   public void f(Object o) { }
   public void g(float f) { }
}
```

```
class B1 extends A
{
   public void f(String s) { }
}

class B2 extends A
{
   public void f(Object o) { }
   public void f(int i) { }
}
```

(a) (6 pts) A non-static method call goes through a table of pointers to methods, called a *virtual function table*, or simply *v-table*. For example, `A`'s *v-table* is the following:



Draw `B1`'s *v-table* and `B2`'s *v-table*. Mention the method's enclosing class name and arguments as well to avoid ambiguity; e.g., "`B1`'s `f(String s)`" and "`B2`'s `f(int i)`".

(b) (6 pts) Assume the following variables are defined:

```
String strval = ...; Object objval = ...; int intval = ...;
A  v1 = new A();
A  v2 = new B1();
B1 v3 = new B1();
A  v4 = new B2();
B2 v5 = new B2();
```

For each of the following method calls, write which class's `f` is invoked at runtime. If a method call does not compile, say so.

`v1.f(strval);`	________	`v4.f(strval);`	________
`v1.f(objval);`	________	`v4.f(objval);`	________
`v2.f(strval);`	________	`v4.f(intval);`	________
`v2.f(objval);`	________	`v5.f(strval);`	________
`v3.f(strval);`	________	`v5.f(objval);`	________
`v3.f(objval);`	________	`v5.f(intval);`	________

7. (12 pts) Use these simplification rules:

(β)	`(fun x -> e) v` $\Rightarrow$ `e[v/x]`
(*let*)	`let x=v in e` $\Rightarrow$ `e[v/x]`
(δ)	$1 + 1 \Rightarrow 2$, etc.

to simplify the following expression to an integer. Label each simplification step with the rule used. Write out all expression in full — do not use "...".) There are eight steps.

```
let f = fun g -> fun x -> g (g x)
in let incr = fun y -> y+1
   in f incr 0
```

8. (10 pts) Fill in "true" or "false" for each statement:

   ________ 1. Object-oriented programming is characterized by the evaluation of expressions instead of executing commands.

   ________ 2. In Java, casting a value to an ancestor (i.e., up-casting) is always legal.

   ________ 3. In Java, casting a value to a descendant (i.e., down-casting) is always legal.

   ________ 4. Java objects are stored on the stack.

   ________ 5. Java uses "call-by-value".

   ________ 6. Function overloading in Java is determined statically.

   ________ 7. In Java, calls of the form `super.`f() are dynamically bound.

   ________ 8. When non-compacting garbage collection is used, at all times, every heap cell is either reachable or on the free list.

   ________ 9. When using reference-counting, the counts must be updated whenever a pointer assignment is done.

   ________ 10. Compacting garbage collection can improve memory performance by increasing locality.