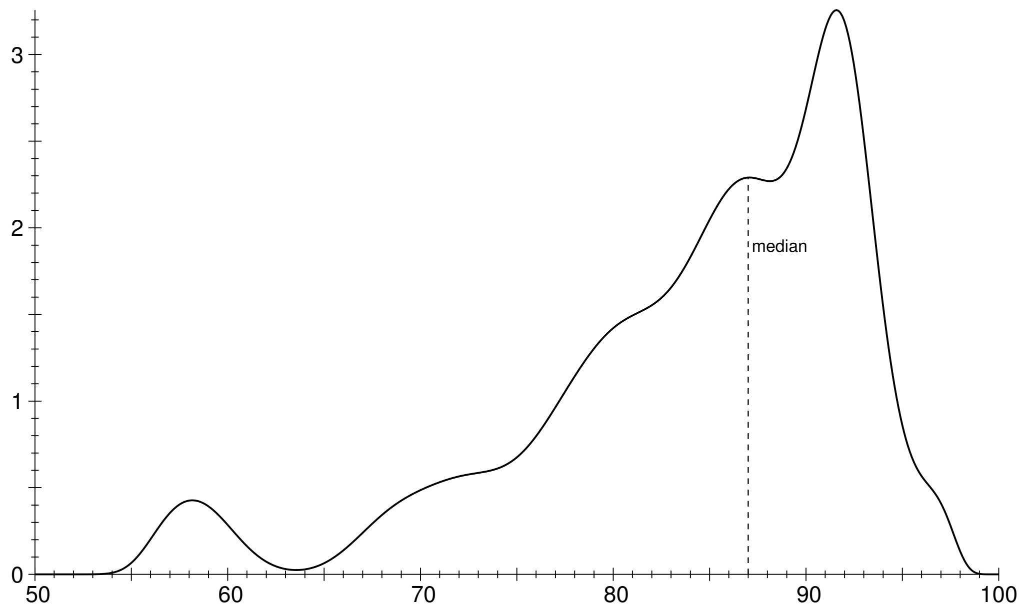
Preliminary Examination I
Computer Science 312, Cornell University 8 March 2007

Solutions

Overall, students did pretty well on the exam. The smoothed histogram of scores looked like this (vertical axis is number of students per unit score):



1. True/False [14 pts]

   (parts a–g; 2 points off for each wrong answer, 1 point off for each blank answer)

   (a) The SML `case` expression can bind new variables.

   True

   (b) Polymorphism allows a function to work on values of many different types.

   True

   (c) Anonymous functions in SML programs cannot be recursive.

   True

   (d) Glass-box test cases can be written using just the specification of the code that is being tested.

   False

(e) If $f(n)$ is $O(g(n))$, then $g(n)$ is also necessarily $O(f(n))$.

False

(f) If function specification A has a weaker precondition than function B, then A may refine B.

True

(g) If an ADT implementation has no representation invariant, to be correct it must be able to handle any possible value of the declared representation type.

True

2. Substitution model [20 pts]

For each of the following examples, give a correct type to fill in the box with. Then use the substitution model of evaluation to show the evaluation of the code. Show every step of evaluation. You may leave out type annotations when you show evaluation. You may use abbreviations as long as it is completely clear what is meant.

(a) [6 pts]

```
let val x = (4, fn x:int => x+1)
    val (y:int, z: [    ]) = x
in
    z(y) + y
end
```

Answer:

Type of z: `int->int`

Evaluation:

```
let val x = (4, fn x => x+1)
    val (y, z) = x
in
    z(y) + y
end
```

$\longrightarrow$

```
let val (y, z) = (4, fn x => x+1)
in
    z(y) + y
end
```

$\longrightarrow$

`(fn x => x+1)(4) + 4` $\longrightarrow$ `(4+1) + 4` $\longrightarrow$ `5 + 4` $\longrightarrow$ `9`

(b) [6 pts]

```
let val r: [    ] = {name = "Bo", age = 24}
    fun name(s: string): char list =
        String.explode("x" ^ s)
in
   name (#name r)
end
```

(Hint: recall that `String.explode` *returns a list containing all the characters in a string)*

Answer:

Type of r: `{name: string; age: int}`

Evaluation:

```
let val r = {name = "Bo", age = 24}
    fun name(s) = String.explode("x" ^ s)
in
   name (#name r)
end
```

$\longrightarrow$

```
let fun name(s) = String.explode("x" ^ s)
in
   name (#name {name = "Bo", age = 24})
end
```

$\longrightarrow$

```
(fn s => String.explode("x" ^ s)) (#name {name = "Bo", age = 24})
```

$\longrightarrow$

```
(fn s => String.explode("x" ^ s)) ("Bo")
```

$\longrightarrow$

```
String.explode("x" ^ "Bo")
```

$\longrightarrow$

`String.explode("xBo")` $\longrightarrow \ldots \longrightarrow$ `[#"x",#"B",#"o"]`

(c) [8 pts]

```
let fun f(n: [    ]): int = case n of
    0 => 0
  | _ => n + f(n-1)
in
   f(1)
end
```

(Hint: define an abbreviation F that you can use to make writing out the evaluation much shorter)

Answer:
type of n: int
Define F = `fn n => case n of 0=>0 | _ => n + •(n-1)`, *where the dot (•) has an arrow going back to the* `fn`.
Evaluation:

```
let fun f(n) = case n of
    0 => 0
  | _ => n + f(n-1)
in
   f(1)
end
⟶ F(1)
⟶ case 1 of 0=>0 | _ => 1 + F(1-1)
⟶ 1 + F(1-1) ⟶ 1 + F(0)
⟶ 1 + (case 0 of 0=>0 | _ => 0 + F(0-1))
⟶ 1 + 0 ⟶ 1
```

3. Zardoz [24 pts] (parts a–c)

For each box, provide a *value* that will cause the entire expression to evaluate to the correct answer: 42. (A list of values $[v_1, \ldots, v_n]$ is considered to be a value.)

(a) [8 pts]

```
let val lst: string list = [   ]
in
 case List.map (fn(x) =>
                   case Int.fromString(x) of
                       SOME n => n
                     | _ => 0)
               lst
 of
    x::y::z::_ => x*x + y*y + z*z
    | _ => 41
end
```

Answer: `["5", "4 ", "1 "]`

(b) [8 pts]

```
let val f = [   ]
    val r: Random.rand = Random.rand(1,1)
    val n = Random.randInt(r) mod 10
in
    f(n) + n
end
```

Answer: `fn x:int => 42 - x`

(**Hint:** Given a seed value of type `Random.rand`, `Random.randInt` returns a pseudo-random (i.e., unpredictable) integer.)

(c) [8 pts]

```
let val zaphod = [    ]
    val {a,b,c} = zaphod
in
    (a*10 + (List.length b)) div (case c of SOME x => 0 | _ => 2)
end
```

Answer: `{a = 8, b = [1,2,3,4], c = NONE}`

4. Data abstraction [42 pts] (parts a–g)

Consider the following signature describing a priority queue ADT:

```
signature PQUEUE = sig
  (* An 'a pqueue is a priority queue, in which elements are ordered,
   * and elements are removed in order.  Elements are of some type 'a.
   * The queue has an ordering function built in. Elements "equal" according
   * to this ordering have the same priority. *)
  type 'a pqueue
  (* create(l, ord) creates a priority queue containing all the
   * elements of l, ordered by ord.
   * Requires: ord is an ordering on 'a. *)
  val create: 'a list * ('a*'a->order) -> 'a pqueue
  (* first(q) is the lowest element in the queue according to its element
   * ordering. *)
  val first: 'a pqueue -> 'a
  (* push(q, x) is the priority queue containing all the elements of q,
   * and also x. *)
  val push: 'a pqueue * 'a -> 'a pqueue
  (* pop(q) is a pair (fst, q') where fst is the lowest element in q and
   * q' contains all the elements of q except fst. *)
  val pop: 'a pqueue -> 'a * 'a pqueue
end
```

Recall that the type `order` is a datatype defined as follows:

```
datatype order = LESS | EQUAL | GREATER
```

(a) [5 pts] Suggest an additional ADT operation that seems likely to be important to clients of this abstraction. Briefly justify adding this operation, and write the declaration and specification.

Answer:

```
(* isEmpty(q) is whether q is an empty queue. *)
isEmpty: 'a pqueue -> bool
```

There is no way to test whether the queue is empty with the current interface, and it's a precondition of some operations. So it will be hard to use the queue as is.

(b) [8 pts] The specifications of `first` and pop have some problems. Write better specifications.

Answer:
There were two things to fix.

First, both operations should check or require that the queue be nonempty, or else define an exception to be raised if the queue is empty.

Second, the specs are written as though there is always a single lowest element. But as suggested by the rep invariant, that isn't true. So these operations can only return one of the elements that is lowest in the ordering. Which one is returned doesn't need to be specified.

Here is a possible implementation of the PQUEUE abstraction. Notice that the rep invariant does *not* require that the underlying list is kept in sorted order. The `first` function has not yet been implemented. (There is an extra copy of this code on the last page of the exam that you may remove for your reference.)

```
structure PQueue :> PQUEUE = struct
  type 'a pqueue = 'a list * ('a * 'a -> order)
  (* AF: (lst, ord) represents the priority queue containing all the elements
   *     in lst, ordered by the function ord.
   * RI: The first element of lst, if any, is less than or equal to
   *     all other elements in lst, according to ord. *)
  fun create(lst: 'a list, ord: 'a * 'a -> order) = case lst of
    [] => ([], ord)
  | h::t =>
        let val (min: 'a, rest:'a list) =
            foldl (fn (x: 'a, (curmin: 'a, elems:'a list)) =>
                     case ord(x, curmin) of
                       LESS => (x, curmin::elems)
                     | _ => (curmin, x::elems))
                  (h, [])
                  t
        in
          (min::rest, ord)
        end
  fun first(lst, ord) = raise Fail "Not implemented"
  fun push(q,x) = case q of
    ([], ord) => ([x], ord)
  | (y::t, ord) => (case ord(x,y) of
                      LESS => (x::y::t, ord)
                    | _ => (y::x::t, ord))
  fun pop(q) = let val (elems, ord) = q in
    (hd elems, create(tl elems, ord))
  end
end
```

(c) [5 pts] Suppose we make the call `create([15,10,33], Int.compare)` (where `Int.compare` has type `int*int->order`). The function `foldl` causes the anonymous function defined in `create` to be called twice. What are the successive values bound to the variables `x`, `curmin` and `elems` in those calls? What is the final result of the call to create?

Answer:
x = `10`, *curmin* = `15`, *elems* = `[]`
x = `33`, *curmin* = `10`, *elems* = `[15]`
Result = `([10, 33, 15], Int.compare)`

(d) [5 pts] Explain in 2–3 sentences how it is possible that this implementation can work despite the fact that its representation of a priority queue does not keep the list in sorted order.

Answer:
The list doesn't have to be kept in sorted order in order for the client's view to be that of a sorted queue. What's important is that it seem sorted from the standpoint of the interface; that is, first and pop should deliver the smallest element.

(e) [4 pts] Some of the implemented operations have constant asymptotic complexity, and some have linear complexity. Which is which?

Answer:
Push is $O(1)$. *create and pop are* $O(n)$.

(f) [5 pts] Give a very simple $O(1)$ implementation of `first`, and explain briefly why your implementation of `first` is correct with respect to the `PQUEUE` signature.

Answer:

```
fun first(lst, ord) = hd lst
```

The rep invariant guarantees that the first element in the list is lowest in the ordering. So that first element is the correct result.

(g) [10 pts] Using the abstraction function and rep invariant, argue that the implementation of `push` is correct. Make sure you consider all the possible cases, and argue not only that the result is correct but that it satisfies any relevant invariants.

Answer:

Either the input priority queue is empty or not. If empty, the result is a list `[x]`*, which correctly represents the priority queue containing that single element (that is, AF(*`[x]`*) is as required by the spec), and it trivially satisfies the rep invariant.*

If the priority queue contains at least one element, evaluation proceeds to the nested `case` *expression. There are two cases: either the element pushed is smaller than the head of the element list in the queue, or it is not. In either case, the resulting list of elements (*`x::y::t` *or* `y::x::t`*) contains all of the elements in the old priority queue (*`y`*, and everything in* `t`*), plus the new element (*`x`*). So the result correctly represents the priority queue containing all the old element and also the new element.*

We also need to argue that the rep invariant is maintained. By the rep invariant, we know `y` *is smaller than or equal to any element in* `t`*. If* `x` *is smaller than* `y`*, then the first nested case arm is evaluated. Because* `y::t` *satisfies the rep invariant, transitivity of the ordering implies* `x` *must be smaller than any element in* `y::t`*. Because the resulting list has* `x` *as its first element, it satisfies the rep invariant. Suppose* `x` *is not smaller than* `y`*, in which case the second case is evaluated. The value of* `y` *must be smaller than or equal to* `x` *(because* `ord` *is an ordering). So the list* `y::x::t` *satisfies the rep invariant because* `y` *is smaller than* `x` *or anything in* `t`*.*

Therefore the implementation is correct.