CS312 Preliminary Exam 1
Cornell University, Spring 2005 March 10, 2005

Instructions

Write your name and netid on this page. There are 5 problems on this exam; check now that you have the whole exam. The prelim is worth 100 points total. The point breakdown for the parts of each problem is printed with the problem. Do all written work on the exam itself. If you are running low on space, write on the back of the exam sheets and be sure to write (OVER) on the front side. This is an closed-book examination; you *may not* use outside materials, calculators, computers, etc. You have 90 minutes. **Good luck!**

Name and NetID ____

Problem	Points	Score
1	15	
2	10	
3	23	
4	30	
5	22	
Total	100	

1. Short Answers [15 pts] (parts a–e)

   Briefly answer each of the questions below.

   (a) [3 pts] What are polymorphic functions and what is the advantage of using them?

   (b) [3 pts] Can different cases in a `case` construct evaluate to expressions of different types? If yes, show an example. If not, explain why.

   (c) [3 pts] For each of the following specifications, indicate if it specifies a precondition, a post-condition, none, or both:

   The `Requires` clause: ____________________
   The `Checks` clause: ____________________
   The representation invariant: ____________________
   The `Returns` clause: ____________________

   (d) [3 pts] Can a curried function `foo x y z = x + z + y` be invoked with parameters `x` and `z` only? Explain briefly.

   (e) [3 pts] Can a call to an $O(n)$ function run much slower than a call to a $\Theta(n^2)$ function, for the same input? Explain briefly.

2. Folding Lists [10 pts]   (parts a–b)

Write the following functions using `foldl`. Remember that function `foldl` is defined as follows:

```
fun foldl (f:'a*'b -> 'b) (v:'b) (l:'a list):'b =
  case l of [] => v
          | h::t => foldl f (f(h,v)) t
```

(a) [5 pts]   Write function `alt` that takes a list $[a_1, a_2, a_3, ..., a_n]$ of integers and computes $a_1 - a_2 + a_3 - ... + (-1)^{n+1} a_n$. For instance:

```
alt([2,1,4,2]) = 3
```

(b) [5 pts]   Write function `diff` that takes a list $[a_1, a_2, a_3, ..., a_n]$ of integers and produces a list $[a_1, a_2 - a_1, a_3 - a_2, ..., a_n - a_{n-1}]$ whose elements are differences btween the elements of the original list, except for the leftmost element, which remains unchanged. For instance:

```
diff([4,1,2,4]) = [4,~3,1,2]
```

3. Zardoz [23 pts] (parts a–d)

For each of the following expressions, fill in the box with a *value* (not an expression) that causes each of the programs to evaluate to 42. Note that lists of values, anonymous functions, and datatype constructors with value arguments are all considered values.

(a) [5 pts]

```
case #zardoz (#ford [    ] ) of
  SOME (m) => m * (m-1)
| NONE => 41
```

Your answer: [   ]

(b) [6 pts]

```
let
  val zardoz = fn x => fn y => y x
in
  zardoz 41 [    ] 43
end
```

Your answer: [   ]

(c) [6 pts]

```
foldl (fn (x,y) => x*x + y) 1 [    ]
```

Your answer: [   ]

(d) [6 pts]

```
let
  datatype ford = Ford | Arthur of ford * ford
  fun zardoz(t:ford) : int =
    case t of
      Ford => 2
    | Arthur(l,r) => zardoz(l) * (zardoz(r) + 1)
in
  zardoz [    ]
end
```

Your answer: [   ]

4. **Abstraction** [30 pts] (parts a–f)

In this problem, you will work with a new queue ADT – a *double-ended queue*, or *deque* for short. A deque is a generalization of a standard queue. In a normal queue, you enqueue elements at one end and dequeue them at the other; in a deque, either operation can occur at either end. That is, given a deque containing the elements:

```
1, 2, 17, 8
```

you can enqueue the element 5 either from the left, to produce:

```
5, 1, 2, 17, 8
```

or from the right, to produce:

```
1, 2, 17, 8, 5
```

Starting with the original queue, you can also dequeue from the left, to produce:

```
2, 17, 8
```

or from the right, to produce:

```
1, 2, 17
```

All other aspects of deques are the same as for queues.

Below is a possible signature for a deque.

```
signature DEQUE = sig
   exception Empty
   exception NotFound

   datatype side = LHS | RHS
   type 'a deque

   (* empty is the empty deque. *)
   val empty : 'a deque
   (* isEmpty q is true iff q has no elements. *)
   val isEmpty : 'a deque -> bool
   (* size q is the number of elements of q. *)
   val size : 'a deque -> int

   (* enqueue s e q = deque q with e enqueued at side s *)
   val enqueue : side -> 'a -> 'a deque -> 'a deque
   (* dequeue s q = (e,q') where e is the element at side s of q,
    * and q' is the deque after removing e. *)
   val dequeue : side -> 'a deque -> 'a * 'a deque
   (* lookup p q = is the leftmost element in the queue for
    * which predicate p is true. *)
   val lookup : ('a -> bool) -> 'a deque -> 'a
end
```

(a) [4 pts] Two pre-conditions are missing in the above specifications. Indicate the functions that need to be fixed and write the missing specifications.

(b) [3 pts] Users of this abstraction may consider that the specification for `lookup` is too strong. Propose a weaker, non-deterministic specification for this function.

A possible implementation is using a pair of two lists that model the two ends of the deque. For instance, the following deque:

```
5, 1, 2, 17, 8
```

can be represented as the pair `([5,1], [8,17,2])`.) Using this representation, a partial implementation of a deque is as follows:

```
structure Deque :> DEQUE = struct
  exception Empty
  exception NotFound

  datatype side = LHS | RHS
  type 'a deque = ('a list) * ('a list)

  val empty = ([],[])

  fun isEmpty((l1,l2): 'a deque) : bool =
    (null l1) andalso (null l2)

  fun size((l1,l2):'a deque):int =
    (length l1) + (length l2)

  fun enqueue (s:side) (e:'a) ((l1,l2):'a deque):'a deque =
    case s of
      LHS => (e::l1,l2)
    | RHS => (l1,e::l2)
end
```

(c) [6 pts] Specify the abstraction function for this implementation.

(d) [6 pts] Below is a partial implementation for function `dequeue`. Fill in the rest of this function. Make sure your code works correctly for all cases.

```
fun dequeue (s:side) ((l1,l2):'a deque):'a * 'a deque =
  case (s,l1,l2) of
    (LHS,h::t,l2) => (h, (t,l2))
```

(e) [5 pts] Your `dequeue` function is probably inefficient if one of the lists is empty and the program performs successive dequeue operations, alternating the ends from where elements are being removed. For instance:

```
dequeue RHS (#2 (deqeue LHS (#2 (dequeue RHS ([],[1,2,3])))))
```

Briefly describe an approach to improve the efficiency of the program in such situations. You don't have to write code.

(f) [6 pts] Now you want to test your code. Assume that you perform Black-Box testing, so you cannot access the implementation structures. An example test case for `size` is "`size {4,3,2,1}`". Here, `{4,3,2,1}` is just a notation for a deque with four elements.

Give 8 good test cases for your `enqueue` and `dequeue` functions (4 test cases for each function). For each case, explain briefly how it provides good coverage (three or four words is enough). You will not receive points for redundant test cases.

5. Induction [22 pts] (parts a–c)

Consider the following function:

```
fun f(m:int, n:int, p:int) : int =
   if p = 0 then m
   else f(n, 2*n, p-1) + m
```

(a) [8 pts] Show the evaluation of the expression `f(3,1,2)` using the evaluation model discussed in class. At each step, underline the subexpression that is being reduced. You may use ellipses (...) for parts that remain unchanged from one step to another.

(b) [10 pts] Prove by induction that $\texttt{f}(\texttt{m}, \texttt{n}, \texttt{p}) = \texttt{m} + \texttt{n}(2^{\texttt{p}} - \texttt{1})$. Be sure that you clearly specify what you're inducting on, what you're proving, what the induction hypothesis is, and when you use it.

(c) [4 pts] What is the asymptotic complexity of this function with respect to parameter `n`. That is, for fixed values of `m` and `p`, what is the order of growth of `f(m,n,p)` relative to `n`? You don't have to write a proof, but just state the complexity.