Computer Science 15–212–X: Final Examination

December 17, 1996

Name

Name: ______________________________

Andrew ID: ______________________________

Instructions

- This is an open-book, open-notes examination.
- There are 18 pages in this examination, including five worksheets.
- This examination consists of 5 questions worth a total of 200 points. The point value of each question is given with the question.
- Read each question completely before attempting to solve any part.
- Write your answers legibly *in the space provided* on the examination sheet. If you use the back of a sheet, indicate clearly that you have done so on the front.
- Write your name and Andrew id in the space provided at the top of this page, and write your name at the top of each page in the space provided.
- Two worksheets are attached to the end of this examination for your personal use. *Under no circumstances will anything written on the worksheets count for or against your grade on this exam.*

Grading

Question	1	2	3	4	5	Total
Score						
Maximum	40	50	35	35	40	200

1 Induction and Recursion [40 pts]

We define binary trees by

```
datatype 'a tree = Empty | Node of 'a * 'a tree * 'a tree
```

We can uniquely identify a subtree of a binary tree by specifying a *path* from the root to the subtree: at each step during the top-down traversal we are instructed that we he arrived (`H`), that we have to go left (`L`) or that we have to go right(`R`). We thus represent paths by

```
datatype path = H | L of path | R of path
```

We call a path p *valid in a tree* t if the path designates some subtree of t. For example, the root of any tree has address `H`, the left subtree has address `L(H)`, the right subtree has address `R(H)`. The address `L(H)` is not valid in the empty tree `Empty`.

1. [10 pts] Write an ML function

   ```
   val subtree : path * 'a tree -> 'a tree
   ```

   such that `subtree`(p, t) returns the subtree of t at address p or raises the exception `Path` if the subtree does not exist.

2. [15 pts] Write an ML function

   ```
   val paths : 'a tree -> path list
   ```

   such that `paths`(t) returns a list containing precisely the valid paths in t. Your function need not be tail-recursive.

3. [15 pts] We define

```
fun mirror (H) = H
  | mirror (L(p)) = R(mirror(p))
  | mirror (R(p)) = L(mirror(p))
fun reflect (Empty) = Empty
  | reflect (Node(d, left, right)) =
      Node(d, reflect(right), reflect(left))
```

Prove that $\texttt{reflect}(\texttt{subtree}(p, t)) \equiv \texttt{subtree}(\texttt{mirror}(p), \texttt{reflect}(t))$ for any tree t and path p valid in t. Here $e_1 \equiv e_2$ means that $e_1 \hookrightarrow v$ if and only if $e_2 \hookrightarrow v$.

2 Streams [50 pts]

Streams are a finite representation of potentially infinite lists. This question develops the fundamentals of potentially infinite binary trees. Consider the basic signatures for lazy trees (`ltree`s).

```
signature LTREE =
sig
  type 'a ltree
  datatype 'a lroot = Empty | Node of 'a * 'a ltree * 'a ltree
  val delay : (unit -> 'a lroot) -> 'a ltree
  val expose : 'a ltree -> 'a lroot
  val map : ('a -> 'b) -> 'a ltree -> 'b ltree
end;
```

1. [20 pts] Implement a structure `LTree :> LTREE` in analogy with a basic implementation of streams that does *not* memoize. In particular, make sure that `map` is lazy.

2. [20 pts] The function `exists : ('a -> bool) -> 'a ltree -> bool` generalizes the corresponding function on binary trees. We specify that `exists` $p\,t$ traverses the lazy tree t and returns `true` if it contains a data element on which p returns `true` and `false` if no such element exists.

   Prove that such a function cannot be implemented, even if we assume that p is a decision procedure. You may assume the undecidability of the halting problem for ML and functions such as `bddEval : exp * int -> value option` or `reduce : exp -> exp` and `isValue : exp -> bool` in your proof.

3. [10 pts] A weaker specification states that for decision procedures p,

   - `exists` $p\,t \hookrightarrow$ `true` if t contains data on which p holds, and
   - `exists` $p\,t \hookrightarrow$ `false` or diverges if t contains no such data element.

   Explain the mistake in the following code and describe a correct implementation. You do not need to write it out in ML.

```
fun exists p t = delay (fn () => exists' p (expose t))
and exists' p (Empty) = Empty
  | exists' p (Node(d,left,right)) =
    p(d) orelse exists p left orelse exists p right
```

3 Abstract Types [35 pts]

As part of the implementation of an editor, we implement a buffer data structure. The first implementation is "persistent", that is, it does not use mutable references or arrays in order to facilitate undo operations.

```
signature BUFFER =
sig
  type buffer
  exception BoB
  exception EoB
  val empty : buffer
  val insert : buffer * char -> buffer
  val delete : buffer -> buffer         (* raises BoB *)
  val forward : buffer * int -> buffer  (* int >= 0, raises EoB *)
  val backward : buffer * int -> buffer (* int >= 0, raises BoB *)
  val point : buffer -> int
end;
```

The state of a buffer is determined by a list of characters and the "*point*", that is, the place in the buffer where insert, delete, and related operations apply. In order to facilitate efficient insert and delete operations, we represent the buffer as the triple consisting of the *reverse* of the list of characters to the left of *point*, the list of characters to the right of *point*, and the number of characters before *point*. For example, the buffer

```
"abcdef"
```

with *point* after the `d` would be represented by

```
([#"d", #"c", #"b", #"a"], [#"e", #"f"], 4)
```

Write the following declarations inside a structure of the form

```
structure Buffer :> BUFFER =
struct
  type buffer = char list * char list * int
  exception BoB
  exception EoB
  ...
end;
```

Be sure to respect the specified types from the signature and maintain the given invariants.

1. [5 pts] `empty`. The empty buffer.

2. [5 pts] `insert`(b, c). Insert character c into buffer b directly before *point*.

3. [5 pts] `delete`(b). Delete the character before *point*, raising `BoB` if *point* is at the beginning of the buffer.

4. [5 pts] `forward`(b, n). Move *point* forward n characters, where $n \geq 0$ may be assumed. Raises `EoB` if it would move past the end of the buffer.

5. [5 pts] `backward`(b, n). Move *point* backwards n characters, where $n \geq 0$ may be assumed. Raises `BoB` if it would move past the beginning of the buffer.

6. [5 pts] `point`(b). Return the position of *point* in buffer b, where 0 is before the first character, 1 right after the first character, *etc.*

7. [5 pts] In an environment where the structure `Buffer` is defined as above, define a function `goto`(b, n) which moves point to position n, raising appropriate exceptions if $n < 0$ or n is greater than the number of characters in b.

4 Mutable State [35 pts]

1. [15 pts] Write a signature `MBUFFER` which modifies the signature `BUFFER` from Question 3 so that it is appropriate for an implementation of a buffer as a *mutable* data structure. In particular, operations executed for effect should return a value of type `unit`.

2. [20 pts] Implement the body of a functor

```
functor MBuffer (structure B : BUFFER) :> MBUFFER =
struct
 ...
end;
```

which implements mutable buffers. Through functor application we can then take advantage of the pure implementation of buffers from Question 3.

5 Parsing and Continuations [40 pts]

In this question we consider the problem of parsing s-expressions. Assume the lexer returns values of type `token` defined by

```
datatype token =
  LPAREN | RPAREN |
  SYM of string | INT of int
```

The parser is supposed to map a list of tokens into a value of type `sexp` and the remaining list of tokens. Note that we do not use streams in this implementation.

```
datatype sexp =
    Sym of string
  | Int of int
  | List of sexp list
```

For example, `parse [LPAREN,LPAREN,SYM("a"),RPAREN,RPAREN,INT(3)]` should evaluate to `(List[List[Sym("a")]],[INT(3)])`. The following is the beginning of an implementation.

```
exception Error of string
fun parseSexp (LPAREN :: s) k = parseList s nil k
  | parseSexp (SYM(x) :: s) k = k s (Sym(x))
  | parseSexp (INT(n) :: s) k = k s (Int(n))
  | parseSexp (nil) k = raise Error ("Incomplete s-expression")
  | parseSexp (RPAREN :: s) k = raise Error ("Unexpected right paren")
and parseList (RPAREN :: s) stack k = k s (List(List.rev(stack)))
  | parseList s stack k =
      ... gap_1 ...
fun parse (s) = parseSexp s (... gap_2 ...)
```

1. [5 pts] Give the type of `parse`

2. [5 pts] Give the type of `parseSexp`

3. [5pts] Give the type of `parseList`

4. [15 pts] Fill gap_1.

5. [10 pts] Fill gap_2.

Worksheet

Worksheet

Worksheet

Worksheet

Worksheet