# Computer Science 15–212–X: Midterm Examination

October 3, 1996

Sample Solution

## Name

Name: **Sample Solution**

Andrew ID: `fp`

## Instructions

- This is a closed-book, closed-notes examination.
- There are 10 pages in this examination, including two worksheets.
- This examination consists of 3 questions worth a total of 100 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 | Total |
|---|---|---|---|---|
| Score | | | | |
| Maximum | 35 | 25 | 40 | 100 |

# 1 Induction and Recursion [35 points]

We define

```
type key = int
type 'a entry = key * 'a
datatype 'a tree = Empty | Node of 'a entry * 'a tree * 'a tree
```

as the type of binary search trees with integer keys and data of arbitrary type `'a`.

1. [5 pts] State the representation invariant of binary search trees.

   For every node `Node((key,datum),left,right)`, all keys occurring in `left` are strictly smaller than `key`, while all keys occurring in `right` are strictly greater than `key`.

2. [5 pts] Define a function `leaves` to count the number of `Empty` nodes in a binary search tree.

   ```
   (* val leaves : 'a tree -> int *)
   fun leaves (Empty) = 1
     | leaves (Node(e,left,right)) = leaves(left) + leaves(right);
   ```

3. [10 pts] Let $|t|$ be the number of *entries* in a binary search tree $t$.

   Prove that $\texttt{leaves}(t) \hookrightarrow \overline{|t|+1}$, where $\overline{n}$ is the representation of $n$ in ML.

   There are many correct proofs of this assertion. Perhaps the most direct one uses induction on the structure of $t$.

   - **Base Case:** $t = \texttt{Empty}$. Then

   $$\begin{aligned} &\texttt{leaves}(t) = \texttt{leaves}(\texttt{Empty}) \\ &\hookrightarrow \texttt{1} = \overline{1} = \overline{0+1} = \overline{|\texttt{Empty}|+1} \end{aligned}$$

   - **Induction Step:** $t = \texttt{Node}(e, \mathit{left}, \mathit{right})$. We assume the induction hypothesis on *left* and *right*:

   $$\begin{aligned} &\texttt{leaves}(\mathit{left}) \hookrightarrow \overline{|\mathit{left}|+1} \quad \text{and} \\ &\texttt{leaves}(\mathit{right}) \hookrightarrow \overline{|\mathit{right}|+1}. \end{aligned}$$

   Then

   $$\begin{aligned} &\texttt{leaves}(t) = \texttt{leaves}(\texttt{Node}(e, \mathit{left}, \mathit{right})) \\ &\Longrightarrow \texttt{leaves}(\mathit{left}) \texttt{ + } \texttt{leaves}(\mathit{right}) \\ &\Longrightarrow^* \overline{|\mathit{left}|+1} \texttt{ + } \texttt{leaves}(\mathit{right}) \quad \text{(by induction hypothesis on } \mathit{left}\text{)} \\ &\Longrightarrow^* \overline{|\mathit{left}|+1} \texttt{ + } \overline{|\mathit{right}|+1} \quad \text{(by induction hypothesis on } \mathit{right}\text{)} \\ &\hookrightarrow \overline{|\mathit{left}|+1+|\mathit{right}|+1} = \overline{(1+|\mathit{left}|+|\mathit{right}|)+1} \\ &= \overline{|\texttt{Node}(e, \mathit{left}, \mathit{right})|+1} = \overline{|t|+1}. \end{aligned}$$

4. [15 pts] Write a function

```
interval : int * int -> 'a tree -> 'a tree
```

where `interval` $(\bar{i}, \bar{j})$ $t$ returns a binary search tree with those entries in $t$ with a key strictly between $i$ and $j$. You may assume without checking that $i < j$. Your function should exploit the representation invariant on $t$.

```
fun interval (i,j) (Empty) = Empty
  | interval (i,j) (Node(e as (key,datum), left, right)) =
      if i >= key then interval (i,j) right
      else if j <= key then interval (i,j) left
           else Node(e, interval (i,j) left, interval (i,j) right);
```

## 2 Abstract Types [25 points]

Consider a signature for positive rational numbers.

```
signature RATIONALS =
sig
  type rat
  exception Undefined
  val quotient : int * int -> rat
  val fraction : rat -> int * int
  val plus : rat * rat -> rat
  val equal : rat * rat -> bool
end;
```

We write $\overline{p}$ for the representation of an integer $p$ in ML, and $\ulcorner \frac{p}{q} \urcorner$ for the representation of the rational $\frac{p}{q}$. This representation remains unspecified in the signature, but we require that a correct implementation of this signature satisfies the following properties.

- `quotient`$(\overline{p}, \overline{q}) \hookrightarrow \ulcorner \frac{p}{q} \urcorner$ if $p \geq 0$ and $q > 0$ and raises exception `Undefined` otherwise;
- `fraction`$(\ulcorner \frac{p}{q} \urcorner) \hookrightarrow (\overline{p'}, \overline{q'})$ where $\frac{p}{q} = \frac{p'}{q'}$ and $\frac{p'}{q'}$ is in cancelled form;
- `plus` and `equal` implement the obvious functions on positive rationals.

1. [15 pts] Implement a structure `R :> RATIONALS` where rationals are represented as pairs (numerator, denominator) in cancelled form. You may assume you have a function `gcd` defined by

```
fun gcd (0,n) = n
  | gcd (m,n) = gcd (n mod m, m);
```

```
structure R :> RATIONALS =
struct
  type rat = int * int
  exception Undefined
  (* val norm : int * int -> int * int *)
  (* Assumes: p >= 0, q > 0 *)
  fun norm (p,q) =
      let val d = gcd(p,q)
      in (p div d, q div d) end
  fun quotient (p,q) =
       if p < 0 orelse q <= 0 then raise Undefined
       else norm (p,q)
  fun fraction r = r
  fun plus ((p,q),(p',q')) = norm (p * q' + q * p', q * q')
  fun equal (r1,r2) = (r1 = r2)
end;
```

2. After a correct declaration of the form `structure R :> RATIONALS = ...;`, what will be the results of the following expressions? In each case,

   - indicate either the type or a type error and explain the reason for the type error, and
   - if the expression is well-typed, give the value or the exception raised.

   (a) [2 pts] `R.fraction(R.quotient(15,6));`
   Type: `int * int`, Value: `(5,2)`.

   (b) [2 pts] `R.fraction(R.quotient(0,5)) handle R.Undefined => 0;`
   Type error: The handler for the exception returns `0 : int` while the expression has type `int * int`.

   (c) [2 pts] `R.equal(R.quotient(2,3),R.quotient(~2,~3));`
   Type: `bool`, no value, raises exception `R.Undefined`.

   (d) [2 pts] `let open R in quotient(1,0) handle undefined => (0,1) end;`
   Type error: The handler for the exception returns `(0,1) : int * int` while the expression has type `rat`. These are different, even inside the scope of `open R`, which only has an effect on identifier names. The fact that `undefined` has been misspelled is irrelevant, since it is simply interpreted as a variable of type `exn`.

   (e) [2 pts] `let open R in quotient((1,0) handle undefined => (0,1)) end;`
   Type: `R.rat`, no value, raises exception `R.Undefined`, since the exception handler applies only to the pair `(1,0)` which is already a value.

# 3 Higher-Order Functions [40 pts]

An infinite sequence $x_0, x_1, x_2, \ldots$ of real numbers can be represented by a function `x : int -> real` where `x`$(\bar{i}) \hookrightarrow \overline{x_i}$ for $i \geq 0$. This question explores this representation technique.

For each function you write below give its type, including auxiliary functions where necessary. You may always assume that the integer arguments to your functions are greater or equal to 0 without checking. Recall that the built-in function `real : int -> real` coerces an integer to the corresponding real number.

1. [5 pts] Give a representation of the sequence $1, \frac{1}{2}, \frac{1}{3}, \ldots$

```
type seq = int -> real
val h : seq = (fn i => 1.0/real(i+1));
```

2. [5 pts] Write a function `add` which, given a pair of sequences $x_0, x_1, \ldots$ and $y_0, y_1, \ldots$ computes the sequence $x_0 + y_0, x_1 + y_1, \ldots$.

```
(* val add : (int -> real) * (int -> real) -> (int -> real) *)
fun add (x:seq) (y:seq) = (fn i => x(i) + y(i));
```

3. [5 pts] Write a function `diff` which, given $x_0, x_1, x_2, \ldots$ computes the sequence of differences, consisting of $x_1 - x_0, x_2 - x_1, \ldots$.

```
(* val diff : (int -> real) -> (int -> real) *)
fun diff (x:seq) = (fn i => x(i+1) - x(i));
```

4. [10 pts] Write a function `sum` which, given a sequence $x_0, x_1, x_2, \ldots$ returns the sequence consisting of the partial sums $x_0, x_0 + x_1, x_0 + x_1 + x_2, \ldots$.

```
(* val sum : (int -> real) -> (int -> real) *)
fun sum (x:seq) = (fn 0 => x(0)
                   | i => (sum x) (i-1) + x(i));
```

5. [15 pts] Write a function `seqToList` which, given a sequence $x_0, x_1, \ldots$ and an integer $n \geq 0$ returns the list $[x_0, \ldots, x_{n-1}]$ of the first $n$ elements of the sequence. In particular, `seqToList` $s$ `0` should always return the empty list.

```
(* val seqToList : (int -> real) -> int -> real list *)
(* n >= 0 *)
fun seqToList (x:seq) (n:int) =
      let
          fun stl (0,l) = l
            | stl (i,l) = stl (i-1, x(i)::l)
      in
        stl (n,nil)
      end;
```