Computer Science 15–212

Midterm Examination
Annotated Solutions
October 13, 1999

Name: __________________________________________

Andrew ID: ______________________________________

Instructions

• This is a closed-notes, closed-book, closed-notebook (computer) examination.

• There are 16 pages in this examination, including three worksheets.

• The examination consists of three (3) problems worth a total of 100 points, plus two extra-credit problems worth a total of 20 points. The extra credit will be recorded separately; so you should finish all of the regular questions before attempting the extra credit. Extra credit points are not necessarily interchangeable with ordinary points.

• Read each problem completely before attempting to solve any part.

• Your answers should be correct, simple and clean, and they should take advantage of the given invariants. In other respects, your functions do not need to be particularly efficient.

• 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.

• The worksheets attached to the end of this examination are for your own use; they will not be used in grading.

Grading

<table>
<thead>
<tr>
<th>Problem</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>Total</th>
<th>Extra Credit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum</td>
<td>35</td>
<td>35</td>
<td>30</td>
<td>100</td>
<td>20</td>
</tr>
</tbody>
</table>
1 Induction and Recursion [35 points + 10 points extra credit]

Recall from 211 that insertion sort is a simple sorting algorithm that works by recursively sorting a list's tail and inserting the head into the result. This problem asks you to implement insertion sort as a recursive function, and to prove its main helper function correct.

1.1 [10 points] Write a function

\[
\text{val insert : int * int list \rightarrow int list}
\]

where \( \text{insert} (x, l) \) returns the sorted integer list obtained by inserting the integer \( x \) into the sorted list \( l \). The list should be sorted in increasing order.

\[
\text{fun insert} (x, \text{nil}) = [x]
\]
\[
| \text{insert} (x, l \text{ as } h::t) = \text{if } x \leq h \text{ then } x::l
\]
\[
\text{else } h::\text{insert}(x,t)
\]

*Common errors:* The most common error was forgetting to put \( x \) in brackets (thus making it into a singleton list) for the base case. Another minor error was writing \((x::h)@t\) or something similar for the first clause of the if statement. This does not typecheck; the cons operator :: is of type 'a * 'a list \rightarrow 'a list.

A few people dropped the element if another element of the same value was already in the list. This was not the specification of insert, and it would result in an insertion sort which also removed duplicate entries from the list.

A few people made recursive calls in both cases (inserting \( h \) into \( t \) if \( x > h \)). While correct, this implementation does an enormous amount of needless work.
1.2 [5 points] Write a recursive function

```haskell
val insertion_sort : int list -> int list
```

that implements insertion sort using the `insert` function. Be sure to implement `insertion_sort` as a recursive function; do not use any built-in list processing functions.

```haskell
fun insertion_sort nil = nil
 | insertion_sort (h::t) = insert(h, insertion_sort(t))
```

1.3 [5 points] Re-implement `insertion_sort` using the built-in function `foldr`. Recall that

```haskell
val foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

and

```
foldr f b nil = b
foldr f b [x1,...,x_n] = f(x1,f(x2,...f(x_n,b)...))
```

```haskell
val insertion_sort = foldr insert nil
```

*another valid, but slightly less elegant solution is:*

```haskell
fun insertion_sort l = foldr insert nil l
```

*Many people put something similar to:*

```haskell
fun insertion_sort nil = nil
 | insertion_sort (h::t) = foldr insert [h] t
```

*which is correct, but significantly less elegant than either of the first two. A few people also used (fn (x,y) => insert(x,y)) instead of just insert, which is an unnecessary location.*

*Common errors: Some people didn’t realize that insert was the proper function to be passed to foldr. At least one person attempted to implement foldr itself. Always remember to read the problem statement carefully before attempting the problem.*
1.4 [15 points] Prove that your insert function works properly. Be sure to state your induction hypothesis clearly, to note the places where you invoke the induction hypothesis, and to verify any preconditions when you invoke the induction hypothesis.

Claim: For all integers \( x \) and all sorted lists of integers \( 1 \), insert\((x,1)\) evaluates to a sorted list containing \( x \) and all of the elements in \( 1 \).

Proof: Let the integer \( x \) be given. We proceed by induction on \( 1 \).

Base case: If \( 1 \) is nil, then insert\((x,1)\) evaluates to \([x]\). This list is sorted (it only has one element) and it contains \( x \) and all elements in \( 1 \) (of which there are none).

Induction step: Otherwise, \( 1 = h::t \) for suitable \( h \) and \( t \). We assume as our induction hypothesis that insert\((x,t)\) works as specified above. We also note that by the preconditions of the function, \( 1 \) is sorted, and therefore \( h \) is a least element of \( 1 \).

If \( x \leq h \), then insert\((x,1)\) evaluates to \( x::1 \). This list contains \( x \) and all of the elements of \( 1 \). Since \( h \) is a least element of \( 1 \), and \( x \leq h \), it follows that \( x \) is a least element of \( x::1 \). We also know that \( 1 \) is sorted. Therefore \( x::1 \) is also a sorted list. Therefore insert works properly.

Otherwise, \( x > h \). In this case insert\((x,1)\) evaluates to \( h::\text{insert}(x,t) \). By the induction hypothesis, insert\((x,t)\) evaluates to a sorted list containing \( x \) and all of the elements of \( t \). Let \( t' \) denote this list. Because \( h \) is a least element of \( 1 \), and \( h < x \), it follows that \( h::t' \) is also a sorted list. In addition, it contains \( x \) and all of the elements of \( 1 \). Therefore insert works properly.

Therefore, for all integers \( x \) and all sorted lists of integers \( 1 \), insert\((x,1)\) evaluates to a sorted list containing \( x \) and all of the elements in \( 1 \).

Comments: Note how the structure of the proof follows exactly the structure of the code. This is most apparent in the induction step. The recursive case of the function is a two-way if statement; thus the proof is by cases. One half of the if statement makes a recursive call; the other does not; thus one of the cases in the proof appeals to the induction hypothesis, while the other does not. (Not every proof is by induction!)

Study carefully this relationship between the structure of code and the structure of a proof of that code. Once you understand this relationship, proving code will be much easier for you. In many cases, the proof can be virtually read directly off of the code.

Common errors: Many collective points were taken off for failing to completely specify the desired behavior of insert, either in the induction hypothesis or in a theorem statement preceding the proof, as above. Merely stating the IH as “the function works” is insufficient; at some point you must explain exactly what this means. Many people stated that insert was to return “a sorted list”. This is also insufficient; you must specify which sorted list, by specifying its elements, as above. Note also how carefully it is shown in each case that this is exactly what the function returns.

Some people wrote as the induction hypothesis that if insert\((x,t)\) works, then insert\((x,h::t)\) works. This is not the induction hypothesis; this is what you have to prove. The induction hypothesis is merely the antecedent of this (i.e., that insert\((x,t)\) works).
1.5 [10 points extra credit]

Prove that your recursive `insertion_sort` function works properly.

Claim: For any integer list `l`, `insertion_sort l` evaluates to a sorted list containing all the elements of `l`.

Proof by induction on `l`.

Base case: If `l` is nil, then `insertion_sort l` evaluates to nil, which is of course sorted and which contains all the elements of `l` (of which there are none).

Induction step: Otherwise, `l = h::t` for suitable `h` and `t`. We assume the induction hypothesis that `insertion_sort t` works as specified.

`insertion_sort l` evaluates to `insert(h, insertion_sort t)`. By the induction hypothesis, `insertion_sort t` evaluates to a sorted list containing all of the elements of `t`. Let `t'` denote this list. `t'` satisfies the precondition for `insert`, and therefore by the previous theorem `insert(h, t')` returns a sorted list containing `h` and all of the elements of `t'` (which contains exactly the elements of `t`). Therefore `insertion_sort(l)` returns a sorted list containing all of the elements of `l`.

Comments: Again notice how the structure of the proof follows the structure of the code. We prove the base case directly, we prove the recursive case by reference to the induction hypothesis, and we appeal to previous theorems when we use helper functions.

Notice also that we made sure to point out that the precondition of `insert` was satisfied.
2 Data Abstraction and Representation Invariants

[35 points]

Sets play a central role in many areas of computer science. This problem will work with the following signature for representing sets of integers.

```
signature INTSET =
  sig
    type set
    exception NotFound
    val empty : set
    val insert : set * int -> set
    val remove : set * int -> set
    val union : set * set -> set
    val intersection : set * set -> set
    val isMember : set * int -> bool
    val isEmpty : set -> bool
  end
```

Each of the operations in the signature has the obvious specification. For example, the function application `remove(s,v)` removes the element `v` from the set `s` and returns the resulting set. If `v` does not belong to `s`, the exception `NotFound` is raised. A set can not have duplicate entries; thus, an attempt to remove an integer twice (with no intervening insertions) must result in the exception `NotFound` being raised.

2.1 [5 points] Choose a representation for your sets by defining the type `set`. Clearly state any representation invariants that you use. (For the purposes of this problem, a data structure that you can implement quickly may be preferable to a more complicated yet efficient one.)

```
type set = int list
```

**Invariants:** none

Other good representations include lists with no duplicates, or sorted lists with no duplicates. If your functions relied on assumptions about the input, you needed to state them here. In the interests of code brevity and ease of writing, this representation includes duplicates, which makes everything but remove and intersection very easy.
2.2 [15 points] Using the type you defined in the previous problem, write the code for an implementation of a structure `IntSet :> INTSET`, by defining the appropriate values. Invariants for any auxiliary functions should be clearly stated.

```ml
structure IntSet :> INTSET =
  struct
    (* invariants: none *)
    type set = int list
    exception NotFound
    val empty = nil
    val isEmpty = List.null
    val union = op @
    fun isMember (s,i) = List.exists (fn y => i = y) s
    fun insert (s,i) = i :: s
    fun remove (s, i) =
      if isMember (s, i) then List.filter (fn y => not (i = y)) s
      else raise NotFound
    fun intersection (a, b) = List.filter (fn y => isMember(a, y)) b
  end
```

Common errors:

Lots of people tried to pattern match on `empty` instead of `nil`. `empty` is just a variable name, so doing this will match any value, and bind `empty` to that value in the function body (as if you had written `x`). You can only pattern match against constants like `nil`, not the contents of variables.

Many people violated representation invariants (like removing only the first occurrence of an item in `remove`, when their `insert` might include duplicates). Others didn’t state invariants that they properly used and maintained.

Some people tried to represent `IntSet` as a function `int -> bool`, which is very elegant, but makes `isEmpty` impossible to write. It’s a good idea to think about all the things you need to do with a data structure before picking a representation.

Some people were confused over the difference between datatypes and types (like trying to declare a special value `Empty` as part of a datatype). If you lost points for this, please review the syntax and/or see us in office hours, as this is an important concept which we will use throughout the rest of the course.
2.3 [15 points] Give the type and value of each of the following expressions, or indicate that the expression is not well-typed, does not have a value, or raises an exception when evaluated. Assume a correct implementation of the structure IntSet :＞ INTSET.

1. IntSet.empty = []
   Not well-typed:
   IntSet.set is not the same type as 'a list – the representation is hidden because of opaque ascription (:'>.)

2. (fn x => IntSet.remove(IntSet.empty, 1))
   Type: 'a -> IntSet.set
   Value: function
   MANY people made the mistake of thinking that this raises an exception. While it does raise an exception every time it is applied, function bodies are not evaluated when making a function.
   Lots of other people said that this didn't have a value. In ML, functions are values (this expression "evaluates" to itself).

3. (fn x => IntSet.remove(IntSet.empty, 1)) ()
   Type: IntSet.set
   Value: No value (exception NotFound raised)
   Even though the result of this application is always an exception, this expression still has type IntSet.set – it is a valid expression wherever a value of type IntSet.set is needed. This is similar to (1 div 0), which has type int, but always raises exception Div.

4. IntSet.insert(IntSet.empty, 1.0)
   Not well-typed.
   IntSet.insert is a function IntSet.set * int -> IntSet.set, so a real constant (1.0) is not well-typed as input.

5. (fn s => fn x => IntSet.insert(s, x+1)) IntSet.empty
   Type: int -> IntSet.set Value: function
   Partially applying this function (set -> int -> set) to an argument of type set returns a function int -> set.

6. IntSet.isMember([1], 1)
   Not well-typed because of opaque ascription (int list does not match type IntSet.set).
3 Continuations and Higher-Order Functions
[30 points + 10 points extra credit]

Consider the language $\mathcal{S}$ of all Siamese strings such as "bobob" or "mamam", which consist of an arbitrary string repeated twice:

$$\mathcal{S} = \{ s \mid s = xx \text{ for some string } x \}$$

Note that the empty string $\varepsilon$ is a member of $\mathcal{S}$. It turns out that the language $\mathcal{S}$ is not the language of any regular expression. This problem asks you to implement an acceptor for this language using continuations.

3.1 [10 points] Using continuations, write a function

$$\text{val twin'} : \text{'a list} \to (\text{'a list} \to \text{'a list} \to \text{bool}) \to \text{bool}$$

satisfying the following specification:

1. $\text{twin'} s \ k \Rightarrow \text{true}$
   if there exist $s_1$ and $s_2$ such that $s = s_1 \oplus s_2$ and $k(s_1, s_2) \Rightarrow \text{true}$

2. $\text{twin'} s \ k \Rightarrow \text{false}$
   otherwise.

Note that $\text{twin'}$ is quite general and will be used in the next question to write an acceptor for Siamese strings.

$$\text{fun twin'} \ \text{nil} \ k = k(\text{nil}, \text{nil})$$
$$| \text{twin'} (s \ \text{as } x::t) \ k = k(\text{nil}, s) \ \text{orelse twin'} t \ (\text{fn } (s, t) \Rightarrow k(x::s, t))$$

Note that there was a typo in the specification of $\text{twin'}$: we require $s = s_1 \oplus s_2$.

People tended to get this problem either completely right or wrong. A common, fairly minor error was to have the wrong base case, writing for example $\text{twin'} (x::t) \ k = k(\text{nil}, t) \ \text{orelse}...$
3.2 [10 points] Using the function `twin'`, write a function

```plaintext
val twin : string -> bool
```
with the specification that `twin(s) ⇒ true` if \( s \in S \), and `twin(s) ⇒ false` otherwise. (You may wish to use the function `String.explode : string -> char list` for converting a string to a list of characters.)

```plaintext
fun twin s = twin' (String.explode s) (fn (s,t) => s=t)
```
3.3 [10 points] Prove the correctness of your function twin, assuming the correctness of the helper function twin'.

The proof is by cases.

Case 1. Suppose that s is a Siamese string, so that \( s = s_1s_2 \) with \( s_1 = s_2 \). Equivalently, we can write \( \text{String.explode } s = \overline{s_1} \odot \overline{s_2} \) where \( \overline{s_1} = \overline{s_2} \). Then the pair \((\overline{s_1}, \overline{s_2})\) satisfies the initial continuation: 
\[
(fn (s, t) \Rightarrow s=t)(\overline{s_1}, \overline{s_2}) \Rightarrow \text{true.}
\]
By assumption, twin' satisfies the specification. Therefore, 
\[
twin' (\overline{s_1} \odot \overline{s_2}) (fn (s, t) \Rightarrow s=t) \Rightarrow \text{true.}
\]

Case 2. Suppose that s is not a Siamese string. Equivalently, there is no way to write the corresponding list as \( \text{String.explode } s = \overline{s_1} \odot \overline{s_2} \) with \( \overline{s_1} = \overline{s_2} \). Then, the statement twin' \((\overline{s_1} \odot \overline{s_2}) (fn (s, t) \Rightarrow s=t) \Rightarrow \text{false follows from the second case in the specification of twin'}.\]

The most common error on this problem was to attempt to use structural induction on lists. This will not work, because when we have a string \( s \notin S \), the string \( xs \) obtained by adding a single character may or may not be in \( S \). For example, while "ama" is not a Siamese string, "mama" is Siamese and "lama" is not.
3.4 [10 points extra credit] Prove the correctness of the function $\text{twin}'$. 