15-150 Spring 2012
Homework 06

Out: Saturday, 25 February 2012
Due: Saturday, 3 March 2012, 23:59 EST

1 Introduction

This homework will focus on using higher order functions and continuations to solve interesting problems elegantly. The assignment will also help you mesh writing code with proving its correctness.

In Sections 3 and 4, you will investigate the idea that

Recursion templates can be represented by higher-order functions.

for several different recursive types. When you write out an instance of a template, you repeat a lot of code. By abstracting repeated patterns into a higher-order function, you can make your code shorter and easier to read and maintain.

In general, the problems on this assignment will be tricky but the solutions will not be that long when you're done.

1.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository as usual.

1.2 Submitting The Homework Assignment

To submit your solutions, place your `hw06.pdf` and modified `hw06.sml` files in your handin directory on AFS:

```
/afs/andrew.cmu.edu/course/15/150/handin/<yourandrewid>/hw06/
```

Your files must be named exactly `hw06.pdf` and `hw06.sml`. After you place your files in this directory, run the check script located at

```
/afs/andrew.cmu.edu/course/15/150/bin/check/06/check.pl
```

then fix any and all errors it reports.

Remember that the check script is *not* a grading script—a timely submission that passes the check script will be graded, but will not necessarily receive full credit.

Also remember that your written solutions must be submitted in PDF format—we do not accept MS Word files.

Your `hw06.sml` file must contain all the code that you want to have graded for this assignment and compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Methodology

You must use the five step methodology for writing functions for every function you write on this assignment. In particular, you will lose points for omitting the purpose, examples, or tests even if the implementation of the function is correct.

1.4 Style

We will continue grading this assignment based on the style of your submission as well as its correctness. Please consult course staff, your previously graded homeworks, or the published style guide as questions about style arise.

1.5 Due Date

This assignment is due on Saturday, 3 March 2012, 23:59 EST. Note that this is not the normal day or time! Remember that this deadline is final and that we do not accept late submissions.

1.6 Characters

The type `char` represents single characters. Here are some useful functions involving characters:

```
String.explode : string -> char list
String.implode : char list -> string
Char.compare : char * char -> order
```

`String.explode` "blows up" a string into the list of characters in that string in order; `String.implode` is the inverse. `Char.compare` orders character.

2 Regular Expressions

In class, we introduced six different operators to describe regular expressions:

- The empty set 0
- The empty string 1
- Characters **c**
- Concatenation $r_1 r_2$
- Alternative $r_1 + r_2$
- Repetition r^*

From time to time it is helpful to have some more constructs available to form regular expressions, such as

- A character wildcard symbol _ which accepts any one character:

$$L(\_) = \{ \text{“}\mathbf{c}\text{”} \mid \mathbf{c} \text{ is a character } \}$$

- Intersection $r_1 \,\&\, r_2$ which accepts a string if and only if it is simultaneously in both $L(r_1)$ and in $L(r_2)$:

$$L(r_1 \,\&\, r_2) = \{s \mid s \text{ in } L(r_1) \text{ and } s \text{ in } L(r_2)\}$$

- A string wildcard **T** which accepts any string:

$$L(\mathbf{T}) = \{s \mid s \text{ is a string } \}$$

The regular expression matcher `match` from class is included in the support code for the assignment. We have extended the `datatype` definition of `regexp` to include the new constructors `Wild`, `Both`, and `Any`, which correspond to _, &, and **T**, respectively. Your job is to extend `match` to deal with these new constructors, and prove parts of the correctness of your implementation. In the notes for Lecture 12, you will find the full statement of the correctness theorem for `match`, including both *soundness* and *completeness*. Here we will ask you to show cases of soundness:

Theorem 1 (Soundness). *For all* r : `regexp`*,* cs : `char list`*,* k : `char list` $\rightarrow$ `bool`*, if* `match` $r\ cs\ k \cong$ `true` *then there exist* p, s *such that* $p@s \cong cs$ *with* $p \in L(r)$ *and* $k\ s \cong$ `true`*.*

In each of the following coding tasks, we strongly recommend that you think through the correctness spec when you are writing the code. If you're stuck on the implementation, try doing the proof of soundness and/or completeness—this will guide you to the answer. However, we will only ask you to hand in soundness for each.

Task 2.1 (5%). Implement the case of `match` for the one-character wildcard _, that is, `Wild`.

Task 2.2 (5%). Complete the following case of soundness:

Case for `Wild`:

To show: For all cs : `char list` and k : `char list` $\to$ `bool`,
if `match Wild` cs $k \cong$ `true`
then $\exists p$, s such that $p@s \cong cs$ and $p \in L(\_)$ and $k\ s \cong$ `true`

Complete this case.

Solution 2.2

Proof. Assume `match Wild` cs $k \cong$ `true`. By stepping

```
  match Wild cs k
≅ case Wild of ...   | Wild => ...   | ...
≅ case cs of [] => false | c::cs' => k cs'
```

Therefore, by transitivity,

```
(case cs of [] => false | c::cs' => k cs') ≅ true
```

Because `cs` is a value of type `char list`, we have two cases to consider: either `cs` is `[]` or `cs` is `c::cs'`.

In the former, `(case cs of [] => false | c::cs' => k cs')` $\cong$ `false`, so `false` $\cong$ `true`, which is a contradiction.

In the latter, `cs` must have the form `c :: cs'` where `c` is some character. By stepping,

```
(case c::cs' of [] => false | c::cs' => k cs') ≅ k cs'
```

so by transitivity $k\ cs' \cong$ `true`. We take p to be `[c]` and s to be `cs'`. Then $p\ @\ s \cong [c]@cs' \cong cs$ (by stepping `@`), $[c] \in L(\_)$ by definition, and $k\ cs' \Longrightarrow$ `true`. $\square$

Task 2.3 (10%). Implement the case of `match` for intersection $r_1 \& r_2$, that is, `Both`(r_1, r_2).

Task 2.4 (10%). Complete the following case of soundness:

Case for $\texttt{Both}(r_1, r_2)$:

To show: for all cs : `char list` and k : `char list` $\to$ `bool`,
if `match` $(\texttt{Both}(r_1, r_2))$ cs $k \cong$ `true` then
$\exists p$, s such that $p@s \cong cs$ and $p \in L(\texttt{Both}(r_1, r_2))$ and k $s \cong$ `true`

Complete this case.

Solution 2.4

Proof. Assume cs, k such that `match` $(\texttt{Both}(r_1, r_2))$ cs $k \cong$ `true`. We need to show that $\exists p, s$ such that $p@s \cong cs$ with $p \in L(\texttt{Both}(r_1, r_2))$ and k $s \cong$ `true`.

Inductive Hypotheses (Soundness on r_1 and r_2):

1. $\forall cs_1$:`char list`, k_1 :`char list -> bool`, if `match` r_1 cs_1 $k_1 \cong$ `true`, then $\exists p_1, s_1$ *s.t.* $p_1@s_1 \cong cs_1, p_1 \in L(r_1), k_1$ $s_1 \cong$ `true`
2. $\forall cs_2$:`char list`, k_2 :`char list -> bool`, if `match` r_2 cs_2 $k_2 \cong$ `true`, then $\exists p_2, s_2$ *s.t.* $p_2@s_2 \cong cs_2, p_2 \in L(r_2), k_2$ $s_2 \cong$ `true`

By stepping:

```
  match (Both (r1, r2)) cs k
≅ case (Both (r1, r2)) of ...  | Both (r1, r2) => ...  | ...
≅ match r1 cs (fn s => match r2 cs (fn s' => charlisteq(s,s') andalso k s'))
```

By assumption, `match (Both (r1, r2)) cs k` $\cong$ `true`. So by transitivity

(i) `match r1 cs (fn s => match r2 cs (fn s' => charlisteq(s,s') andalso k s'))` $\cong$ `true`

We can then apply IH1, taking k_1 to be

```
(fn s => match r2 cs (fn s' => charlisteq(s,s') andalso k s'))
```

and cs_1 to be cs, and using (i) to satisfy the premise. Then we know that $\exists p_1, s_1$ such that $p_1@s_1 \cong cs$, $p_1 \in L(r_1)$, and `(fn s => match r2 cs (fn s' => charlisteq(s,s') andalso k s'))` $s_1 \cong$`true`.

By stepping

```
  (fn s => match r2 cs (fn s' => charlisteq(s,s') andalso k s'))s1
≅ match r2 cs (fn s' => charlisteq(s1,s') andalso k s')
```

Thus, by transitivity,

(ii) `match r2 cs (fn s' => charlisteq(`s_1`,s') andalso k s')` $\cong$ `true`

We can then apply IH2, taking k_2 to be

`(fn s' => charlisteq(`s_1`,s') andalso k s')`

and cs_2 to be `cs` and using (ii) to satisfy the premise. Then we know that $\exists p_2, s_2$ such that $p_2@s_2 \cong cs$, $p_2 \in L(r_2)$, and `(fn s' => charlisteq(`s_1`,s') andalso k s')` s_2 $\cong$`true`.

By stepping

$$\begin{aligned} & \texttt{(fn s' => charlisteq(}s_1, s_2\texttt{) andalso k s')}s_2 \\ \cong\ & \texttt{charlisteq(}s_1, s_2\texttt{) andalso k } s_2 \end{aligned}$$

Thus, by transitivity, `charlisteq(`s_1, s_2`) andalso k` s_2 evaluates to `true`. By inversion on `andalso`, `charlisteq(`s_1, s_2`)` $\cong$ `true` and `k` $s_2 \cong$ `true`. By correctness of `charlisteq`, $s_1 \cong s_2$. By lemma, since $p_1@s_1 \cong cs$, $p_2@s_2 \cong cs$, and $s_1 \cong s_2$, it must be the case that $p_1 \cong p_2$—if there are two splittings of cs with the same suffix, then the prefixes must be the same.

So, take p to be p_1 and s to be s_1. Then $p = p_1 \in L(r_1)$ and $p = p_1 \cong p_2 \in L(r_2)$. So $p@s \cong p_1@s_1 \cong cs$, $p \in L(\texttt{Both}(r_2, r_2))$ by definition, and $k\ s \cong k\ s_1 \cong$ `true`. $\square$

Do this proof carefully! There is a plausible-looking, but incorrect, implementation of `Both`; this case of the proof will fail if your code has this bug.

Note: we will go over the code for ***`Star`*** *in lecture on Tuesday; you may want to wait until after then for the next two tasks.*

As in the case with `Star`, the case for `Any` should use a helper function called `matchany`. This function should do all the work of matching `Any` with the given continuation, `k`.

Task 2.5 (10%). Define the function `matchany` so that it evaluates to true if and only if there is some suffix (possibly the whole list) of its argument that satisfies the given continuation.

To prove the soundness of the `Any` case, we will prove the following lemma about `matchany` in the scope of a given continuation k:

Lemma 1. *For all* cs : `char list`*, if* `matchany` $cs \cong$ `true` *then* $\exists p, s$ *such that* $p@s \cong cs$ *with* $k\ s \cong$ `true`*.*

Task 2.6 (10%). Prove Lemma 1 by structural induction on cs.

Solution 2.6 We use the following definition of `matchtop`:

```
fun matchtop cs' =
   case cs' of
      [] => k []
    | _::cs'' => k cs' orelse matchtop cs''
```

Proof. **Case for `[]`:** Assume `matchtop` $[] \cong$ `true`.

To Show: $\exists p, s$ such that $p@s \cong$ `[]` with $k\ s \cong$ `true`.

```
  matchtop []
≅ case [] of [] => k [] | ...
≅ k []
```

Therefore, by transitivity, `k []` evaluates to `true`. Take p and s to be `[]`. Thus, $p@s \cong []@[] \cong []$ and $k\ s \cong k\ [] \cong$ `true`, completing the base case.

Case for $c :: cs'$:

Inductive Hypothesis: If `matchtop cs'` $\cong$ `true`, then $\exists p', s'$ such that $p'@s' \cong cs'$ with $k\ s' \cong$ `true`.

To Show: Assume `matchtop` $(c :: cs') \cong$ `true`; we must show $\exists p, s$ such that $p@s \cong c :: cs'$ with $k\ s \cong$ `true`.

By stepping

```
  matchtop (c::cs')
≅ case c::cs' of ...  | c::cs' => k (c::cs') orelse matchtop cs'
≅ k (c::cs') orelse matchtop cs'
```

By transitivity, `(k (c::cs') orelse matchtop cs')` $\cong$ `true`. By inversion on `orelse`, we have two cases to consider, one where each disjunct evaluates to `true`:

Case 1: `k (c::cs')` $\cong$ `true`

We take p to be `[]`, and s to be $c :: cs'$. Observe that `[]`$@(c :: cs') \cong c :: cs'$, so $p@s \cong c :: cs'$, and that $k\ s \cong$ `true`.

Case 2: `matchtop cs'` $\cong$ `true`

This fact allows us to use the inductive hypothesis to show that $\exists p', s'$ such that $p'@s' \cong cs'$ with $k\ s' \cong$ `true`. Take p to be $c :: p'$ and s to be s'. Then $p@s \cong (c :: p')@s' \cong c :: (p'@s')$ (by associativity) $\cong c :: cs'$. And $k\ s \cong k\ s' \cong$ `true`.

3 File Systems

3.1 Structural Recursion on Trees

As a first example, consider structural recursion on trees, as in `size`:

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


fun size (t : 'a tree) : int =
    case t of
        Leaf x => 1
      | Empty => 0
      | Node (l, r) => size l + size r
```

This pattern can be abstracted into a function `mapreduce`:

```
fun mapreduce (f : 'a -> 'b) (e : 'b) (n : 'b * 'b -> 'b) (t : 'a tree) : 'b =
    case t of
        Leaf x => f x
      | Empty => e
      | Node (l, r) => n (mapreduce f e n l, mapreduce f e n r)
```

`mapreduce` takes three arguments: `f` says what to do in the `Leaf` case, as a function of the data stored at the leaf; `e` says what to do in the `Empty` case; `n` says how to combine the recursive results in the `Node` case. It returns a function `'a tree -> 'b` that applies this process to the tree.

For example,[1]

```
fun size (t : 'a tree) : int = mapreduce (fn _ => 1) 0 (op+) t
```

The name `mapreduce` comes from the fact that

`mapreduce f e n` ≅ `(reduce n e) o map f`

for `map` and `reduce` defined in Lecture 10.

[1] Note that `op` allows an infix operator to be used as a function (`+` by itself, without arguments or `op`, doesn't parse); `op+` is equivalent to `fn (x,y) => x + y`.

3.2 Structural Recursion on a File System

In this section we will use the following simple representation of objects in a filesystem:

```
datatype fsobject =
    File of string * int
  | Dir of (string * fsobject) tree
```

A `fsobject` is either a file consisting of the **contents** and size of the file (represented by a `string` and `int`, respectively) or a directory consisting of a collection of `fsobject`'s, each paired with a name represented by a `string`. We represent this collection as a `tree`, so that if a directory had lots and lots of entries in it, we could process them in parallel. Note that individual `File`s do not have a name—the filename is part of the enclosing directory.

Analogous to `reduce` on trees and lists, we define `fsreduce` to operate on values of type `fsobject`:

```
val fsreduce :    ((string * int) -> 'a)
               -> ((string * 'a) tree -> 'a)
               -> (fsobject -> 'a)
```

Its first argument is a function that computes the result for a single file, from the file's contents and size. The second argument is a function that computes the result for a directory, from a tree corresponding to the contents of a directory, where each recursive occurence of an `fsobject` has been replaced by the result of a recursive call on it. The third argument is just the `fsobject` itself. As an example of defining functions using `fsreduce`, we consider the `count_rec` function that computes the number of names (of both files and directories) that satisfy the given predicate (*i.e.*, function of type `string` → `bool`). The code is given here for reference:

```
fun count_rec (match : string -> bool) (fso : fsobject) : int =
    let val case_for_leaf = fn (name : string, subcount : int) =>
             subcount + (case match name of true => 1 | false => 0)
    in
      case fso of
          File _ => 0
        | Dir t =>
              let fun loop t =
                      case t of
                          Node (t1, t2) => loop t1 + loop t2
                        | Leaf (n, fso') =>
                          case_for_leaf (n, count_rec match fso')
                        | Empty => 0
              in
                  loop t
              end
    end
```

This function definition exhibits a pattern of recurring over a `fsobject` that can be used to define many functions. It defines the result for a single `File` (*i.e.*, `0`), and then defines the result for a `Dir` with a recursive function that traverses the tree of `fsobject`'s in the directory. For each `Leaf` of the tree, it performs some action on the name and the recursive result of the function on the nested `fsobject`. These traversals of `fsobject`'s and `tree`'s are represented more concisely using `fsreduce` and `mapreduce` in the following alternative definition:

```
val count_reduce : (string -> bool) -> (fsobject -> int) =
  fn match =>
    let
      val case_for_leaf = ... same as above ...
    in
      fsreduce (fn _ => 0) (mapreduce case_for_leaf 0 op+)
    end
```

The `File` branch of the outer `case` in the recursive version corresponds to the first function argument to `fsreduce`. The `Leaf` branch of `loop` corresponds to the first argument to `mapreduce`. The `Empty` branch of the `loop` corresponds to the second argument to `mapreduce`. Finally, the `Node` branch of `loop` corresponds to the third argument to `mapreduce`. Observe that we only partially apply `mapreduce`, so that the expression is of type `int tree` → `int`. Similarly, we only partially apply `fsreduce` so the expression is of type `fsobject` → `int`.

There are a few really important benefits of rewriting code in this style:

1. It's shorter (1 line instead of 11!).

2. It's easier to read: much of reading code is about finding familiar patterns that you understand, and using them to understand new code. In the first version, you have to puzzle out the fact that the outer recursion defines an `fsreduce` and the inner `loop` a `mapreduce`. The second version tells you what pattern it is using, which we can do because the pattern is expressed as a higher-order function!

In the next two tasks we will ask you to take some other examples of recursive code, extract the pattern of recursion, and express it using `fsreduce` with `mapreduce`.

3.3 du

We begin with the function `totsize_rec`, which computes the total size of all files in the given `fsobject` (cf. the unix command `du`). Here is the recursive version:

```
fun totsize_rec (fso : fsobject) : int =
    case fso of
        File (_, sz) => sz
```

```
        | Dir t =>
              let fun loop t =
                  case t of
                      Node (t1,t2) => loop t1 + loop t2
                    | Leaf (_ , fso') => totsize_rec fso'
                    | Empty => 0
              in
                  loop t
              end
```

Task 3.1 (10%). Define the function

```
totsize_reduce : fsobject -> int
```

to compute the total size of all the files in the given `fsobject` using `fsreduce` with `mapreduce`. The function definition should be no more than a couple lines.

3.4 grep

The unix command `grep` finds all files matching a given pattern, and prints their full paths along with their contents. We model this by the following recursive function, `all_matches_rec`, which computes a tree of all the names that match a given predicate along with their absolute paths:

```
fun all_matches_rec (match : string -> bool) (fso : fsobject)
    : (string * string) tree =
    let
      fun case_for_leaf (name : string, t' : (string * string) tree)
          : (string * string) tree =
          let
            val submatches =
                treemap (fn (name', path) => (name', "/" ^ name ^ path)) t'
          in
            case match name of
                true => Node (Leaf (name, "/" ^ name), submatches)
              | false => submatches
          end
    in
      case fso of
          File _ => Empty
        | Dir t =>
            let fun loop t =
                case t of
```

```
                 Node (t1, t2) => Node (loop t1, loop t2)
               | Leaf (n, fso') =>
                     case_for_leaf (n, all_matches_rec match fso')
               | Empty => Empty
        in
            loop t
        end
end
```

Task 3.2 (10%). Define the function

```
all_matches_reduce : (string -> bool) -> fsobject -> (string * string) tree
```

to compute all the `fsobject` names that satisfy the given predicate along with the path to the `fsobject`. You should still use the `case_for_leaf` function for the leaves of the tree. Your code should be no more than a couple lines. You may wish to use your regular expression matcher to construct predicates on names for test cases.

4 Patterns of Recursion On Lists

4.1 Structural Recursion on Lists

Recall the following two functions from Homework 5:

```
fun ap (l1 : 'a list, l2 : 'a list) : 'a list =
    case l1
     of [] => l2
      | x::xs => x::ap(xs,l2)

fun concatap (l : 'a list list) : 'a list =
    case l
     of [] => []
      | x::xs => ap(x,concatap(xs))
```

Both functions are structurally recursive, in that they have the form

```
  fun f (l : 'a list) : 'b =
    case l
      of [] => <some expression e1>
       | x::xs => <some expression e2 involving x and a recursive call on xs>
```

We can capture this pattern by lifting the `e1` from the `[]`-branch and the `e2` from the `x::xs`-branch to be arguments. This is a new higher order function called `fold`, which has type

$$(\alpha * \beta \to \beta) \to \beta \to \alpha\ \mathit{list} \to \beta$$

The first argument is the function that we use to combine the elements with the recursive call; the second argument is the result for when there are no more elements; the third argument is the list to combine in this way.

We can implement `fold` by following the above pattern with some new arguments:

```
fun fold (f : 'a * 'b -> 'b) (b : 'b) (l : 'a list) : 'b =
    case l
     of [] => b
      | x::xs => f(x, fold f b xs)
```

We can then use `fold` to implement the two functions above very simply, by turning each branch of the `case` into a function argument to `fold`:[2]

```
fun ap (l1 : 'a list, l2 : 'a list) : 'a list = fold (op ::) l2 l1
val concatap : 'a list list -> 'a list = fold ap []
```

The SML basis library provides an implementation of `fold` as `List.foldr`

4.2 Tasks

For each task below, implement the specified function using `List.foldr`. You may not write recursive solutions to any of these tasks. You may use `List.map` and non-recursive helper functions unless otherwise indicated, but you may not use other built in library functions on lists.

If you get stuck on a task, we suggest that you implement the function recursively using the above template for structural recursion, and then turn the branches of the `case` *into arguments to* `List.foldr`*.*

Task 4.1 (2%). Implement a function `foldmap : ('a -> 'b) -> 'a list -> 'b list` such that

$$\texttt{foldmap} \cong \texttt{List.map}$$

You may not use `List.map` for this task.

Task 4.2 (2%). Implement a function `foldid : 'a list -> 'a list` that is the identity function for lists. Specifically,

$$\texttt{foldid} \cong \texttt{(fn x => x)}$$

Your implementation must run in linear, not constant, time.

Task 4.3 (3%). Implement a function `foldfilter : ('a -> bool) -> 'a list -> 'a list` that keeps all and only the elements of the list that satisfy the predicate, in their original order. For example,

$$\texttt{foldfilter (fn x => x > 9) [1,2,3,4,10,11,12]} \cong \texttt{[10,11,12]}$$

Specifically,

$$\texttt{foldfilter} \cong \texttt{List.filter}$$

[2] The `op` keyword is an extremely convenient way of taking an infix identifier, like `::`, and using it as a normal function—if `I` is some infix identifier, `op I` $\cong$ `fn (x,y) => x I y`. So, for example,

$$\texttt{(op ::)} \cong \texttt{(fn (x,y) => x::y)}$$

Task 4.4 (7%). Implement a function `prodofsum : int list list -> int` that, given a list of lists, computes the sum of each inner list, and then the product of those sums.[3] For example,

`prodofsum [[1,2,3],[],[],[~6]]` $\cong$ `6 * 0 * 0 * ~6` $\cong$ `0`

Task 4.5 (8%). Implement a function `cntchar : char list -> char -> int * int` with *one* call to `List.foldr`. If there are `n` characters in a list of characters `l` and a particular character `c` appears in the list `k` times, then `cntchar l c` $\cong$ `(k,n)`.[4] For example,

`cntchar (explode "curry") #"r"` $\cong$ `(2,5)`
`cntchar (explode "howard") #"z"` $\cong$ `(0,6)`

4.3 Tail Recursion on Lists

Another common pattern of recursion on lists is tail recursion with an accumulator parameter. For example, recall the following tail recursive function from Lecture 11:

```
fun sumlist (l : int list) : int =
    let
      fun sumtc (l : int list) (acc : int) : int =
          case l
           of [] => acc
            | x::xs => sumtc xs (acc + x)
    in
      sumtc l 0
    end
```

This is similar to `fold` above, in that we're combining every element of a list with some function and with some base case—here, integer addition and zero—but different in that we pass ourselves the partial results in an accumulator argument to make the function tail recursive. We can abstract that pattern out as another higher order `foldtc` function:

```
fun foldtc (f : 'a * 'b -> 'b) (b : 'b) (l : 'a list) : 'b =
    case l
     of [] => b
      | x::xs => foldtc f (f (x,b)) xs
```

[3] Recall that the empty sum is 0 and the empty product is 1.
[4] Recall that `#"x"` is the SML notation for character literals.

We can then use this new `foldtc` to recapture our original tail-recursive `sumlist` function:

```
val sumlist : int list -> int = foldtc (op +) 0
```

The SML basis library provides an implementation of `foldtc` as `List.foldl`. It's important to note that while both functions have the same type, they encapsulate profoundly different—if related—patterns of recursion on lists.

4.4 More Tasks

For each task below, implement the specified function using `List.foldr` or `List.foldl` as appropriate. You may not write recursive solutions to these tasks. You may use `@` but you may not use other built in functions on lists.

Task 4.6 (4%).
Implement a function `foldrevslow : 'a list -> 'a list` that reverses its argument in time quadratic in its length. Specifically,

$$\texttt{foldrevslow} \cong \texttt{List.rev}$$

Task 4.7 (4%).
Implement a function `foldrevfast : 'a list -> 'a list` that reverses its argument in time linear in its length. Specifically,

$$\texttt{foldrevfast} \cong \texttt{List.rev}$$