15-150 Summer 2017
Homework 10

Out: Tuesday, 27 June 2017
Due: Wednesday, 28 June 2017 at 23:59 EST

0 Introduction

This assignment will focus on writing using lazy programming techniques in SML as well as coverage of the imperative features in SML.

0.1 Getting The Homework Assignment

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

0.2 Submitting The Homework Assignment

Code submissions will be handled through Autolab, at

```
https://autolab.andrew.cmu.edu
```

Written submissions will be handled through Gradescope, at

```
https://gradescope.com
```

To submit your code, run `make` from the `hw/10` directory (that contains a `code` folder. This should produce a file `handin.tar`, containing the files that should be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `handin.tar` file via the “Handin your work” link.

To submit your written solutions (which *must* be in PDF form), log in to Gradescope and submit your PDF to the assignment **Homework 10**. If you are unable to do this, make a Piazza post or find a TA to help.

The Autolab handin script does some basic checks on your submission: making sure file names are correct; making sure that no files are missing; making sure your code compiles cleanly; making sure your functions have the right types. In addition, the script tests each function that you submit individually using public tests. These public tests are in no way comprehensive, and will in most cases consist of a simple base case test. In order to receive full points, you must pass private tests that will not be run until after the submission deadline has passed, so you will still have to write your own tests to ensure correctness.

The script indicates a correct submission via a score on the Autolab website. If all files are present, your submission will receive a score of “0.0” in the “files” category. If your code compiles, it will receive a score of “0.0” in the “compile” category. If files are missing or the code does not compile, you will receive a highly negative score for the corresponding category. For each function, you will receive a score that represents the number of public tests passed for that function. If you do not receive full points for the public tests for a particular function, you will receive a score of zero for that function.

Your `hw10.sml` file must contain all the code that you want to have graded for this assignment, and must 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.

0.3 Due Date

This assignment is due on Wednesday, 28 June 2017 at 23:59 EST. Remember that you ***do not get grace days this semester***, so we will not be accepting late submissions without a documented, university-approved excuse.

0.4 Methodology

You must use the five step methodology discussed in class for writing functions, for **every** function you write in this assignment. Recall the five step methodology:

1. In the first line of comments, write the name and type of the function.

2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments passed to the function.

3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns).

4. Implement the function.

5. Provide testcases, generally in the format
   `val <return value> = <function> <argument value>`.

For example, for the factorial function:

```
(*  fact : int -> int
 *  REQUIRES:  n >= 0
 *  ENSURES: fact(n) ==> n!
*)

fun fact (0 : int) : int = 1
  | fact (n : int) : int = n * fact(n-1)

(* Tests: *)

val 1 = fact 0
val 720 = fact 6
```

0.5 Style

For this assignment, we will be grading your submissions based on your coding style. There are several ways that you can learn what is good style and what isn't:

- Your returned and graded homework submissions have been graded for style so use the markups for a reference.
- We have published solution code for the previous assignments, labs, and lectures.
- We have published a style guide at

  `https://www.cs.cmu.edu/~15150/resources/handouts/style.pdf`.
- You can ask your TAs about specific examples, or post on Piazza asking general questions.

If any code you submit for a problem violates our style guidelines or is otherwise difficult to understand, you can lose up to 4 points for that problem (that is, you can lose up to 4 points on Task 3.1, 4 points on Task 3.2, etc.). You can lose up to 10 style points overall on a homework assignment.

If you lose style points, you may present to a TA a compelling argument as to why your code has proper style, which, if successful, will grant you the points you lost due to style. Note that the TA you ask will have the final say in whether your code meets the style guidelines sufficiently.

Lazy finite and infinite lists

Here is a datatype whose values can represent finite or infinite lazy lists, with some useful functions which you can use in the homework.

Make sure your code works properly with finite and infinite lazy lists.

```
(* datatype definition *)
datatype 'a lazylist =
  Nil
| Cons of 'a * (unit -> 'a lazylist)

(* show : int -> 'a lazylist -> 'a list *)
fun show n xs =
  case (n, xs) of
    (0, _) => []
  | (_, Nil) => []
  | (_, Cons (x, xs')) => x :: show (n - 1) (xs' ())

(* singleton : 'a -> 'a lazylist *)
fun singleton x =
  Cons (x, fn () => Nil)

(* nats : int -> int lazylist *)
fun nats n =
  Cons (n, fn () => nats (n + 1))

(* lazymap : ('a -> 'b) -> 'a lazylist -> 'b lazylist *)
fun lazymap f xs =
  case xs of
    Nil => Nil
  | Cons (x, xs') => Cons (f x, fn () => lazymap f (xs' ()))

(* lazyfilter : ('a -> bool) -> 'a lazylist -> 'a lazylist *)
fun lazyfilter p xs =
  case xs of
    Nil => Nil
  | Cons (x, xs') =>
      if p x
      then Cons (x, fn () => lazyfilter p (xs' ()))
      else lazyfilter p (xs' ())
```

```
(* union : ('a * 'a -> order) -> 'a lazylist * 'a lazylist -> 'a lazylist *)
fun union cmp (xs, ys) =
  case (xs, ys) of
    (_, Nil) => xs
  | (Nil, _) => ys
  | (Cons (x, xs'), Cons (y, ys')) => (
      case cmp (x, y) of
        LESS => Cons (x, fn () => union cmp (xs' (), ys))
      | EQUAL => Cons (x, fn () => union cmp (xs' (), ys' ()))
      | GREATER => Cons (y, fn () => union cmp (xs, ys' ()))
    )
```

1 Comparison functions of various kinds

1.1 How to compare

For a type `t`, a function `cmp : t * t -> order` is a *comparison function* if it is total and, for all `x, y, z` of type `t`:

- `cmp (x, y) = LESS` iff `cmp (y, x) = GREATER`
- `cmp (x, y) = EQUAL` iff `cmp (y, x) = EQUAL`
- `cmp (x, y) = LESS` and `cmp (y, z) <> GREATER` implies `cmp (x, z) = LESS`
- `cmp (x, y) = GREATER` and `cmp (y, z) <> LESS` implies `cmp (x, z) = GREATER`
- `cmp (x, y) = EQUAL` and `cmp (y, z) = EQUAL` implies `cmp (x, z) = EQUAL`

Remember "iff" is short for "if and only if," i.e., logical biimplication.

A value `L` of type `t lazylist` is `cmp`-sorted if no value occurring in `L` is `cmp-GREATER` than any value that occurs later. Similarly, `L` is `cmp`-increasing if every value in `L` is `cmp-LESS` than all values that occur later.

A `cmp`-increasing lazy list is also `cmp`-sorted.

When `t` is an *equality type*, a comparison `cmp : t * t -> order` is *linear* if for all values `x, y` of type `t`, `cmp (x, y) = EQUAL` implies `x = y`.

When `cmp` is linear, a `cmp`-increasing lazy list has no repeated elements.

The built-in SML function

```
String.compare : string * string -> order
```

compares strings based on their "dictionary" ordering.

For example,

```
String.compare ("a", "b") = LESS
String.compare ("aardvark", "abba") = LESS
String.compare ("arthur", "alfie") = GREATER
```

`string` is an equality type, and `String.compare` is a linear comparison.

1.2 Lazy merge

Recall `union`, a higher-order function for merging lazy lists:

```
(* union : ('a * 'a -> order) -> 'a lazylist * 'a lazylist -> 'a lazylist *)
fun union cmp (xs, ys) =
  case (xs, ys) of
    (_, Nil) => xs
  | (Nil, _) => ys
  | (Cons (x, xs'), Cons (y, ys')) => (
      case cmp (x, y) of
        LESS => Cons (x, fn () => union cmp (xs' (), ys))
      | EQUAL => Cons (x, fn () => union cmp (xs' (), ys' ()))
      | GREATER => Cons (y, fn () => union cmp (xs, ys' ()))
    )
```

This function is intended to satisfy the following specification:

```
REQUIRES:
  - cmp is linear
  - xs and ys are cmp-increasing
ENSURES:
  union cmp (xs, ys) = all values in xs and ys, in cmp-increasing order
```

This function *does not* satisfy this specification, unless we assume an extra property for the `cmp` function, or additional properties for `xs` and `ys`.

Let's explore the reasons for this failure.

Task 1.1 (2 pts).

```
doctor : string -> string lazylist
```

is a function such that for all string values `x`, `doctor x` represents the lazy list

```
"", x, x^x, x^x^x, ...
```

Some examples:

- `doctor "a"` represents `"", "a", "aa", "aaa", ...`

- `doctor "b"` represents `"", "b", "bb", "bbb", ...`
- `doctor "foo"` represents `"", "foo", "foofoo", "foofoofoo", ...`
- `doctor ""` represents `"", "", "", ...`

Determine the value of

```
show 10 (union String.compare (doctor "a", doctor "b"))
```

Task 1.2 (4 pts).

Define a linear comparison function

```
stringcmp : string * string -> order
```

such that

```
  show 10 (union stringcmp (doctor "a", doctor "b"))
= ["","a","b","aa","bb","aaa","bbb","aaaa","bbbb","aaaaa"]
```

Hint: You can use the built-in SML function `size : string -> int`.

Task 1.3 (2 pts).

A comparison function `cmp : t * t -> order` is *accessible* if for all values `x` of type `t`, there is no infinite lazy list of values $\mathtt{y}_0, \mathtt{y}_1, \ldots$ such that for all $i \geq 0$, $\mathtt{cmp}(\mathtt{y}_i, \mathtt{y}_{i+1}) = \mathtt{LESS}$ and $\mathtt{cmp}(\mathtt{y}_i, \mathtt{x}) = \mathtt{LESS}$.

Determine which of the following comparison functions, if any, are accessible.

- `String.compare`
- `stringcmp`

Task 1.4 (3 pts).

Explain briefly why `union` satisfies the following specification:

```
REQUIRES:
  - cmp is linear and accessible
  - xs and ys are cmp-increasing
ENSURES:
  union cmp (xs, ys) = all values in xs and ys, in cmp-increasing order
```

1.3 Lazy pairwise combination

Let `t` be a type, let `cmp : t * t -> order` be a comparison function, and let `f : t * t -> t` be a function. We say that `f` is `cmp`-monotone if it is total, and for all values `x, y, x', y'` of type `t`:

- `cmp (x, x') = LESS` implies `cmp (f (x, y), f (x', y)) = LESS`
- `cmp (y, y') = LESS` implies `cmp (f (x, y), f (x, y')) = LESS`

For example, `op *` and `op +` are `Int.compare`-monotone functions.

Task 1.5 (10 pts).

Define a recursive SML function

```
pairwith :
('a * 'a -> order) -> ('a * 'a -> 'a) ->
'a lazylist * 'a lazylist -> 'a lazylist
```

that satisfies the following specification:

```
REQUIRES:
  - cmp is linear and accessible
  - f is cmp-monotone
  - xs and ys are cmp-increasing
ENSURES:
  pairwith cmp f (xs, ys) = all values f (x, y), in cmp-increasing order,
  where x in xs, y in ys
```

For example,

```
pairwith stringcmp (op ^) (doctor "a", doctor "b")
```

represents

```
"", "a", "b", "aa", "ab", "bb", "aaa", "aab", "abb", "bbb", ...
```

Hint: It's OK to use `union` here.

2 Randomness Two Ways

As you may know, computers are really bad at being random. So what do we do when we need randomness for our algorithms? We turn to *pseudo-random* numbers.

A pseudo-random number generator (or PRNG, for short), is an algorithm which takes an initial value, called a *seed*, and produces a lazy list of random-looking numbers from that. [1]

Most PRNGs will use the initial seed to generate a random number and a new seed, and then use that new seed to generate the next random number and the next seed.

Depending on your point of view, you may have read that sentence and thought "Oh, it's a lazy list", or "Hey, they're just updating the seed at each iteration." In fact, both views are used in functional languages!

SML and OCaml use benign effects for their PRNGs, and Haskell uses something similar to lazy lists (the State monad). In this section, you'll implement a PRNG both ways.

2.1 The Blum-Blum-Shub PRNG

In this question, you'll implement the Blum-Blum-Shub PRNG. This PRNG is quite good, given its simplicity.

Suppose x_0 is your seed to the PRNG, and $m = pq$, where p and q are large primes. The i^{th} random number in the sequence (1-indexed) is defined as

$$x_i = x_{i-1}^2 \mod m$$

In practice, we would take some transformation of this number, perhaps the `xor` of its bits, or some subset of its digits. But for this assignment, we'll take x_i as-is.

Task 2.1 (6 pts).

In the structure `LazyRandom`, define the function

```
bbs : int -> int -> int LazyList.lazylist
```

`bbs m seed` should evaluate to a lazy list of integers representing a Blum-Blum-Shub PRNG with modulus `m` and initial seed `seed` when given an `m` that is the product of two large primes.

[1] What is "random-looking", you ask? Good question. There are several tests, but we can never *really* know if a sequence was randomly generated.

Task 2.2 (6 pts).

In the structure `RefRandom`, define the type `randstate`, and the functions:

```
init :  int -> int -> randstate
next :  randstate -> int
```

`init m seed` should evaluate to a value `r` such that the i^{th} call to `next r` produces the i^{th} random number from a Blum-Blum-Shub PRNG with modulus `m`, a product of two large primes, and initial seed `seed`.

Here, as in the definition of the Blum-Blum-Shub algorithm, we 1-index.

3 Expecting Stuff to Change

Arrays are a commonly used mutable data structure, especially in imperative languages. While SML has its own array implementation in the standard basis library, we will task you to implement your own version of arrays.

3.1 Representation

We will represent arrays as the type

```
type 'a array = int -> 'a ref
```

A value `arr` of type `t array` is a function from some contiguous set of integers (say 0 through 10) representing the indices to a set of `'a ref` cells representing the elements.

If we wish to change an element of `arr` at index i, we can simply modify the ref cell at the ith index.

Likewise, if we want to get the value at index i, we can simply bang the ref cell at the ith index.

If the array is not defined for an index i, applying the array to `i` will result in an exception `OutofBounds` being raised.

Notice that traditionally, arrays have constant time random access. However, this requires pointer arithmetic and direct access to memory, neither of which we have, so we will make do with logarithmic random access time.

3.2 Implementation

In the followings tasks, you will implement the `ARRAY` signature, which includes all functions involved with interacting with arrays represented in the way described in the previous section.

You may *not* use any predefined SML types such as sequences or SML's own built-in arrays (obviously). You may, however, define your own `datatype`s.

Task 3.1 (20 pts).

Implement the function:

```
init_array:  int -> 'a -> 'a array
```

`init_array n v` is a new array defined over the indices 0 through $n - 1$, with all elements initialized to `v`.

It is important that all ref cells be newly defined.

Make sure to have your array raise the appropriate error for out-of-bounds access.

This function should have $O(n)$ work and $O(\log n)$ span, where n is the argument `n`.

Task 3.2 (3 pts).

Implement the function:

```
update:  'a array -> int * 'a -> unit
```

`update a (i, v)` updates the `i`th element of `a` to be `v`.

This function should raise the appropriate out-of-bounds error if called on an index that is not inside the given array.

This function should have $O(\log n)$ work and span, where n is the number of elements in `a`.

Task 3.3 (3 pts).

Implement the function:

```
lookup:  'a array -> int -> 'a
```

`lookup a i` evaluates to the `i`th element of `a`.

This function should raise the appropriate out-of-bounds error if called on an index that is not inside the given array.

This function should have $O(\log n)$ work and span, where n is the number of elements in `a`.

See you at the final! Good luck, have fun!