15-150 Summer 2016
Homework 11

Out: Tuesday, 21 Jun 2016
Due: Wednesday, 22 Jun 2016 at 23:59 EDT

1 Introduction

This assignment will practice imperative programming.

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

Submissions will be handled through Autolab, at `https://autolab.andrew.cmu.edu`.

To submit your solutions, run `make` from the `hw/11` directory (that contains a `code` folder). This should produce a file `hw11.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 `hw11.tar` file via the "Handin your work" link.

This homework will be partially autograded. When you submit your code some *very* basic tests are run. These are the public tests, and you will immediately see your score on these. After the final submission deadline, we will run a more comprehensive suite of private tests on your code. Your final score will be a function of your public score, private score, and any manual grading we perform. Non-compiling code will automatically receive a 0.

All the code that you want to have graded for this assignment should be contained in `imperative.sml`, 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, your code will not compile in our environment.

1.3 Due Date

This assignment is due on Wednesday, 22 Jun 2016 at 23:59 EDT. You have no late days. You may submit as many times as you would like until the due date.

1.4 Methodology

Since you are implementing modules, we give you the signatures for the modules. The specs are in the signatures, so you don't need to write specs. However you still need to write testcases.

For example, for the factorial function presented in lecture: Please write tests in a separate testing structure(s).

For example:

```
signature FOO =
sig
  val fact : int -> int
end

structure Foo : FOO =
struct
  (*  fact : int -> int
   *  REQUIRES:  n >= 0
   *  ENSURES: fact(n) ==> n! *)
  fun fact (0 : int) : int = 1
    | fact (n : int) : int = n * fact(n-1)
end

structure FooTests =
struct
  val 1 = Foo.fact 0
  val 6 = Foo.fact 3
end
```

2 The Only Constant is Change

Arrays are a commonly used mutable datastructure, 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.

Representation

We will represent arrays as the type

```
datatype 'a array = A of int * (int -> 'a ref)
```

A value `A(n,f)` of type `t array` represents an array of length `n` using a partial function `f` which maps any integer `0` to `n-1` to an `'a ref` reference cell containing the element at that index.

If we wish to change an element of `A(n,f)` at index i to the value `x`, we can use the following code: `f i := x`. Likewise, if we want the i^{th} element of `arr`, we can use `!(f i)`. If an index i is out of bounds, `f i` can do whatever it wants. We provide an exception `OutofBounds` that you can raise in this case.

Note: Built-in arrays have constant-time access, but because we are implementing arrays with functions, we will allow accessing an element to take $O(\log n)$ time instead.

Implementation

Task 2.1 (10 pts). Implement the function

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

`init_array n a` is a new array defined over the indices 0 through $n-1$, with all elements initialized to `a`. Beware of *aliasing*: different elements need to be in different reference cells. Random access must have $O(\log(n))$ work at most.

Task 2.2 (1 pts). Implement the function

```
get :  'a array -> int -> 'a ref option
```

`get a i` returns `SOME(` the i'th element of `a)` or NONE if it is out of bounds.
Task 2.3 (3 pts).

Implement the function

```
double_len:  'a array -> 'a -> 'a array
```

`double_len arr n a` is an array of length $2n$ with the exact same first n elements and ref-cells as `arr` (such that modifying an element in one will be reflected in the other). The last n elements should be initialized to `a`. This new larger array should also have logarithmic work random access if `arr` has logarithmic work random access.

Task 2.4 (3 pts). Implement the function

```
foldl:  ('a ref * 'b -> 'b) -> 'b -> ('a array) -> 'b
```

`foldl f b a` returns the value produced by using f to combine all the *references* in the array from left-to-right (in the same order as `List.foldl`). This should not change the state of the array unless `f` does.

Task 2.5 (3 pts). Implement the function

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

`expand a n x` expands `a` to a size of exactly n, filling in the new elements with x (hint: You know how to make arrays longer, and making them shorter is easy.)

Memo

Task 2.6 (8 pts). Storing previous results in an array can make code faster. Implement the function

```
memo_fib:  unit -> int -> IntInf.int
```

where `memo_fib ()` returns a function (let's call that function `f`), where `f i` is the i^{th} Fibonacci number (starting at $0, 1$). For all `i:int`, `f` should only compute the value of `f i` once, storing the result in a array. After that,

calling `f i` should result in looking up the appropriate value in the array. You should test that your function is not exponential time by running a test that would take too long if it was exponential-time. **Hint:** `memo_fib ()` should return a new function that uses some local state.

Note: This function returns `IntInf.int` which is like `int` except it can represent arbitrarily large numbers (which is useful for testing `fib`). This type uses the exact same type as `int` and supports the same operations (you can write out numerals, write `+` and `*`, etc.) To use `IntInf.int` you just need to add at least one type annotation somewhere, such as `(0 : IntInf.int)` so that it doesn't accidentally use `int` instead.

Searching Arrays

Let's practice different ways of searching an array by writing `Finds`

Task 2.7 (2 pts). Implement

```
findr:  ('a ref -> bool) -> 'a Array.array -> 'a ref option
```

`findr p a` returns (`SOME` applied to) a reference `r` from the array `a` such that `p r == true` if one exists, `NONE` otherwise.

Task 2.8 (1 pts). Implement

```
findv:  ('a -> bool) -> 'a Array.array -> 'a ref option
```

`findv p a` returns (`SOME` applied to) a reference `r` from the array `a` such that `p x == true` for the value `x` currently stored in `r`, if one exists, `NONE` otherwise.

Task 2.9 (2 pts). Implement

```
findv_imp('a -> bool) -> 'a Array.array -> 'a ref ref -> bool
```

`findv_imp p a out` searches for a reference `r` from the array `a` such that `p x == true` for the value `x` currently stored in `r`. If such an element exists, `findv` stores it in `out`. `findv_imp` returns `true` iff it modified `out`, `false` otherwise.

Task 2.10 (2 pts). Implement

```
containsr :  'a ref > 'a Array.array -> bool
```

`containsr a r` evaluates to `true` iff the reference `r` is in the array `a`, `false` otherwise.

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