Spring 2021
Programming Languages
Homework 3

- This homework includes an ML programming assignment and short answer questions. You should use Standard ML of New Jersey for the programming portion of the assignment. A link is available on NYU Classes.
- Due Wednesday, April 7, 2021 at 11:55 PM Eastern Daylight Time. Submit two files: a PDF `<netid>-hw3.pdf` containing your solution to the questions in section 1 and another file `<netid>-hw3.sml` containing solutions to all of the ML questions.
- Late submissions are highly discouraged. Nonetheless, the following provision exists for late submissions. A late penalty of 15 points per day applies for submissions received during the first 24 hour period after the deadline. An additional 15 point penalty (total 30 points) applies for the subsequent 24 hour period. Submissions are not accepted for credit more than 48 hours after the deadline. **No exceptions will be made.** Please note that partial credit may be awarded, so take this into consideration if you decide to submit late.
- For the ML portion of the assignment, do not use imperative features such as assignment `:=`, references (keyword `ref`), or any mutable data structure, such as `Array`.
- You may use any published ML references to learn the language. In particular, the book *Elements of ML Programming* by Jeffrey Ullman is highly recommended reading for the newcomer to ML. You may call any functions that are either used or defined in the lecture slides without citing them. Otherwise, all homework solutions including algorithmic details, comments, specific approaches used, actual ML code, etc., **must** be yours alone. Plagiarism of any kind will not be tolerated.
- There are 100 possible points. For the ML question, you will be graded primarily on compliance with all requirements. However, some portion of the grade will also be dedicated to readability, succinctness of the solution, use of comments, and overall programming style.
- Please see `http://www.smlnj.org/doc/errors.html` for information on common ML errors. Look in this document first to resolve any queries concerning errors before you ask someone else.

1 Memory Allocation and Deallocation

1. [20 points] **Garbage Collection 1**

   Consider the following C# code:

```
static IEnumerable<Student> GetBestStudents()
{
   Student s1 = new Student { Name = "Jane", Id = 28127, GPA = 3.6M };
   Student s2 = new Student { Name = "John", Id = 17273, GPA = 2.8M };

   Course c = new Course
     {
        Name = "Programming Languages",
        Prof = "Plock"
     };

   c.Add(s1);
   c.Add(s2);

   Department d = new Department { Name = "CS" };
   d.Add(c);

   // Where produces a new IEnumerable<Student>
   return c.Where(i => i.GPA > 3.5M).OrderBy(i => i.Name);
}

 static void Main(string[] args)
{
    Course c2 = new Course { Name = "Honors PL", Prof = "Goldberg" };
    c2.AddRange(GetBestStudents()); // add multiple items at once

    // GC here
}
```

   C# is based on Java and is therefore very similar. Variables of a non-primitive type are references to objects in the heap. For example, $c2$, $s1$, $s2$, and d are all stack-allocated variables which reference heap objects by means of the **new** operator.

   Assume the heap pointer method is used, in conjunction with copy collection. Assume there exist TO and FROM spaces (not generational, not G1) and that the roots are processed in the order listed in the program text, from top down. Draw a diagram showing the state of heap memory as it exists:

   1. Before copy collection begins at the line with comment "GC here," while `Main` is still executing.
   2. After copy collection executes and fully completes at the line with comment "GC here" while `Main` is still executing.

   When it comes to container objects, we are going to make some simplifying assumptions that deviate from reality. Specifically, assume that container objects (`Department`, `Course`, `IEnumerable<Student>`) are all singly-linked (i.e. "one way") list data structures. There will be a memory allocation for the container itself (regardless of the contents), one allocation for the linked list cell each time an item is added, and another allocation for the object being stored. (The container is not responsible for allocating the object to be stored—the programmer is expected to do that before adding the item and passing the reference to the allocated object to the `Add` method.) Since it is a linked list, each cell will contain a

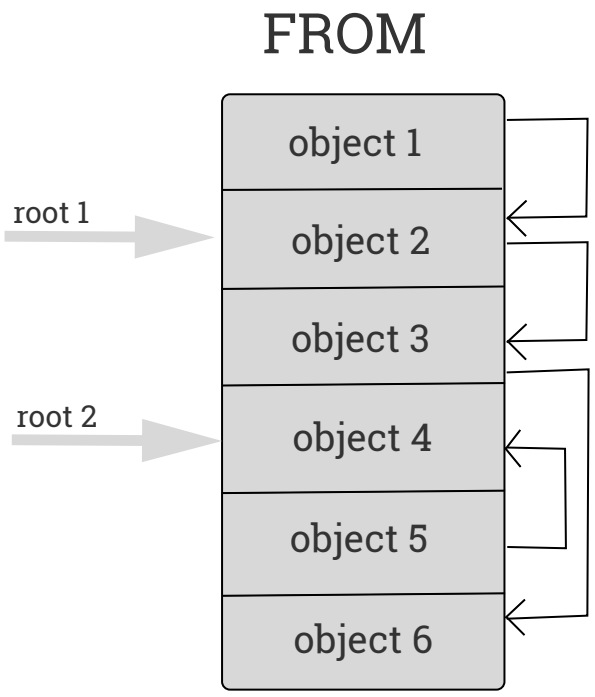
data field and pointer field which will point to the next item, if any. If the data field is of a primitive type, it will contain the value itself. Otherwise, the data field will be a pointer to heap memory.

The `Add` method adds one item to the container by creating a linked list cell in which the new item is referenced, and pointing the previous last element to the new cell, whereas `AddRange` adds a collection of items—think of it as a method that makes individual calls to `Add`.

In both diagrams, show:

- All objects in FROM and TO space—in the correct order from left-to-right (or top-to-bottom, depending on how your heap diagram is oriented), where the far left (top, respectively) object is the oldest object.
- For each object in the heap above, draw all outbound references to other objects.
- The roots pointing into the heap. If there is more than one root, label the order which you assumed they were processed by the copy collector.
- The outbound references between objects. If there is more than one outbound reference, label the order in which you assumed they were processed by the copy collector.
- You do not have to show the forwarding addresses.

2. [15 points] **Garbage Collection 2**



The above represents the FROM heap for a program that utilizes copy collection. Assume that the roots point to objects 2 and 4, as depicted. Draw the FROM and TO space after copy collection runs. Assume the roots are processed in order from top-to-bottom above. Also draw all forwarding address pointers using arrows, similarly to the question above. You may draw these heaps using graphics or freehand (e.g. scanner, taking a picture), as long as it is legible and is contained within the same PDF document as the rest of the solutions.

3. [15 points] **Memory Allocation**

For each of the following, come up with a free list (minimum 3) and sequence of allocation requests (minimum 3) that will result in the following outcomes:

- Best-fit allocation can satisfy all requests, but First-fit and Worst-fit cannot.
- First-fit allocation can satisfy all requests, but Best-fit and Worst-fit cannot.
- Worst-fit can satisfy all requests, but First-fit and Best-fit cannot.
- Worst-fit and best-fit allocation can satisfy all requests, but First-fit cannot.

2 ML

4. [25 points] **Getting Started with ML**

Implement each of the functions described below, observing the following points while you do so:

- You may freely use any routines presented in the lecture slides.
- Make an effort to avoid unnecessary coding by making your definitions as short and concise as possible. Most functions for this question should occupy a few lines or less.
- Make sure that your function's signature *exactly* matches the signature described for each function below.
- You will likely encounter bizarre errors while you write your program and most of the time they will result from something quite simple. The first page of this assignment contains a link to a page which discusses the most common ML errors and an English translation of what each of them mean. Consult this before approaching anyone else. Google also exists.
- If the question asks you to raise an exception, you are not required to turn in any code to catch it. On the other hand, it isn't a bad idea to write a test bed to test your program, in which case you might want to catch it.
- Some of the questions below require a fairly clear understanding of datatype `option`, which was discussed in the slides. You are encouraged to review the slides and experiment on your own with the use of `option` before attempting the questions below.

1. Implement `reverse = fn : 'a list -> 'a list` using the Standard Basis Library function `foldl`. Do not use recursion directly.

   Example:

   ```
   - reverse [1,2,3];
   val it = [3,2,1] : int list
   ```

2. Write a function `composelist = 'a -> ('a -> 'a) list -> 'a` which, given an initial value v and a list of unary functions $f_1, \ldots, f_n$, computes $f_n(\ldots(f_2(f_1(v))))$.

   Example:

   ```
   composelist 5 [ fn x => x+1, fn x => x*2, fn x => x-3 ];
   val it = 9 : int


   composelist "Hello" [ fn x => x ^ " World!", fn x => x ^ " I love", fn x => x ^ " PL!"];
   val it = "Hello World! I love PL!" : string
   ```

3. Write a function `scan_left : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b list` that returns a list of each value taken by the accumulator during the processing of a fold.
   For example:
   `scan_left (fn x => fn y => x+y) 0 [1, 2, 3]` would return [0, 1, 3, 6].
   Hint: try starting with this curried definition of foldl:

   ```
   fun myfoldl f y []     = y
     | myfoldl f y (x::xs) = myfoldl f (f x y) xs;
   ```

4. Write a function `zip: 'a list * b list -> ('a * 'b) list` that takes a pair of lists (of equal length) and returns the equivalent list of pairs. Raise the exception `Mismatch` if the lengths don't match. Example:

   ```
   - zip ([1,2,3,4,5], ["a","b","c","d","e"]);
   val it = [(1,"a"),(2,"b"),(3,"c"),(4,"d"),(5,"e")] : (int * string) list
   ```

```
- zip ([1,2,3,4,5], ["a","b","c","d"]);

uncaught exception Mismatch
  raised at: stdIn:xx.xx-xx.xx
```

5. Write a function `unzip : ('a * 'b) list -> 'a list * 'b list` that turns a list of pairs (such as generated with `zip` above) into a pair of lists. Example:

```
- unzip [(1,"a"),(5,"c"),(3,"e")];
val it = ([1,5,3],["a","c","e"]) : int list * string list
```

6. Write a function `bind = fn : 'a option -> 'b option -> ('a -> 'b -> 'c) -> 'c option` which, given two `option` arguments x and y, evaluates to `f x y` on the two arguments, provided neither x nor y are `NONE`. Otherwise, the function should evaluate to `NONE`. Examples:

```
(* Define a method that operates on ordinary int arguments
   We choose add purely for the sake of example. *)
fun add x y = x + y;
val add = fn : int -> int -> int

bind (SOME 4) (SOME 3) add;
val it = SOME 7 : int option

bind (SOME 4) NONE add;
val it = NONE : int option
```

   Functions like `bind` are examples of the *monad* design pattern, discussed in further detail immediately following this list of questions. Specifically, the bind function accepts a monadic type[1] (`option`), invokes an ordinary function (e.g. `add`) on the underlying type (`int`) and then evaluates to the monadic type. Any irregularities (e.g. `NONE`) that are passed in are passed right back out.

7. Write a function `getitem = fn :int -> 'a list -> 'a option` which, given an integer n and a list, evaluates to the nth item in the list, assuming the first item in the list is at position 1. If the value v exists then it evaluates to `SOME v`, or otherwise evaluates to `NONE`. Examples:

```
getitem 2 [1,2,3,4];
val it = SOME 2 : int option

getitem 5 [1,2,3,4];
val it = NONE : int option
```

8. Write a function `getitem2 = fn : int option -> 'a list -> 'a option`. This is similar to above, but instead of accepting an `int` as the first argument, it accepts `int option`. The function should evaluate to `NONE` if `NONE` is passed as an argument, and behave as above otherwise. Examples:

```
getitem2 (SOME 2) [1,2,3,4];
val it = SOME 2 : int option

getitem2 (SOME 5) [1,2,3,4];
val it = NONE : int option

getitem2 NONE [1,2,3];
val it = NONE : int option

getitem2 (SOME 5) [];
```

[1] A monadic type is a type that wraps some underlying type and provides additional operations. Datatype `option` is a monadic type because in this example because it wraps the underlying type, `int`. In general, it can wrap any type since `OPTION` is parametrized by a type variable.

```
stdIn:251.1-251.20 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val it = NONE : ?.X1 option

(* Oops. Let's try this instead *)
getitem2 (SOME 5) ([] : int list);
val it = NONE : int option
```

Hint: this should follow the same idea as the *bind* function, but should fix the underlying routine as `getitem`.

Why do we care about monads?

Monads originate from category theory, but were popularized in the field of programming languages by Haskell, and more generally by the functional paradigm. Over time, the pattern has crept into imperative languages too and is now fairly universal, although one may not hear the term *monad* used to describe it. Most readers have experienced some type of monad prior to now. For example, the equivalent of `datatype 'a option` in the .NET Framework is `Nullable<T>`. In Java, it is `Optional<T>`. Many other monadic types exist as well besides expressing the presence or absence of a value—this one just happens to be very common.

One property of the monad design pattern is that irregularities are handled within the *bind* function, rather than through more traditional means such as exception handling. The monad design pattern gives rise to another now-popular syntactic pattern seen just about everywhere: *function chaining*, used to create *fluent interfaces*. This pattern is known for its readability while at the same time performing non-trivial operations where choices must be made along the way. Consider this example:

```
Using("db1").Select(x => x.FirstName).From("People").Where(x => x.Age < 20).Sort(Ascending);
```

For those not familiar with this style of programming, `Using` evaluates to a value (in an object-oriented language, typically an object), upon which the `Select` method is invoked. This method evaluates to a new value (object), which is then used to call `From`, and so on. The functions are therefore called in sequence from left-to-right. What confuses most newcomers is that these functions don't pass values to the next function through arguments, but rather through the object each routine evaluates to. In a functional language, this pattern would show up as a curried function. As we already know, passing parameters to curried functions creates bindings which remain visible to later calls, making functional languages ideal for monads.

Monads are not *necessary* to chain functions in general and function chaining is only one use case for monads. However, they are incredibly helpful for the following reason: during a chain of calls such as above, it is desirable for irregularities occurring early in the chain to be gracefully passed to the subsequent calls. For example, if the table "People" does not exist in the database, the function `From` might evaluate to a monadic value such as TABLE_NO_EXIST, which would then be passed seamlessly through each of the remaining calls without "blowing up" the rest of the expression. Without monads, programmers would typically rely on exceptions. The problem with exceptions is that they are computationally expensive, can happen just about anywhere, can be difficult to trace, and must be properly handled or else other parts of the code may also break. It is much easier to learn about and deal with irregularities after the entire expression has fully evaluated.

5. [25 points] **Multi-Level Priority Queue in ML**

A priority queue is a sorted container of pairs, with each pair consisting of the *priority* and *value* of the item being stored. The priority queue works much like an ordinary queue and has similar operations, such as `enqueue` and `dequeue`. However, one difference is that a priority queue will always dispense the highest priority item, whereas an ordinary queue dispenses items on a first-in, first-out basis.

A *multi-level* priority queue consists of several priority queues, one at each level. Items are enqueued at a particular level. Two or more items enqueued at the same priority and level should be maintained in FIFO order. Retrieving an element from such a queue yields the element with highest priority from the non-empty queue with highest level. The structure also supports an operation that moves elements from one level to the level below if certain conditions are met.

For instance, such a structure is used for scheduling processes in operating systems. New processes are inserted in the queue with highest level. As processes age, they are moved to the lower level queues. Since the scheduler always chooses processes from higher-level queues, this allows short-lived processes to complete faster.

Implement a multi-level priority queue using a list that contains all the elements of all queues. Each element is a triple consisting of: its priority, the level of the queue it belongs to, and the data item. Priorities and levels are represented as integer numbers:

```
type 'a mlqueue = (int * int * 'a) list
```

The first integer is the level, the second is the priority. Smaller numbers represent higher priorities and levels. The list is maintained sorted, such that elements of a queue with level n occur before all the elements of queue with level $n+1$. Furthermore, within the same level, elements with higher priorities occur at the front of the list. Using this representation, implement the following functions to support such a data structure:

```
val maxlevel : int

(* enqueue q l p e = inserts an element e with priority p
 * at level l in the multi-level queue q. Raise LevelNoExist
 * if the level doesn't exist.  *)
val enqueue : 'a mlqueue -> int -> int -> 'a -> 'a mlqueue

(* dequeue q = a pair containing: the element with highest
 * priority from the highest-level queue in q; and the
 * remaining queue.
 * Raises Empty if the queue is empty. *)
val dequeue : 'a mlqueue -> 'a * 'a mlqueue

(* move pred q = moves all elements that satisfy the predicate
 * pred to a lower level queue within q, provided a lower level
   exists. *)
val move : ('a -> bool) -> 'a mlqueue -> 'a mlqueue

(* atlevel q n = a list of all elements at level n. The list
 * contains (priority, data item) pairs, and is sorted by
 * priorities.  Raises LevelNoExist if level n does not
 * exist.  If no elements exist at a particular level,
 * the function should output an empty list. *)
val atlevel : 'a mlqueue -> int -> (int * 'a) list
```

```
(* lookup pred  q = looks up the first element in q that
 * satisfies pred, and returns its level and priority.
 * Raises NotFound if no such element exists. *)
val lookup : ('a -> bool) -> 'a mlqueue -> int * int

(* isempty q = true iff the queue q contains no elements. *)
val isempty : 'a mlqueue -> bool

(* flatten q = outputs the data items of the entire mlqueue
 * as a list. *)
flatten : 'a mlqueue -> 'a list
```

You will utilize the function definitions you create above inside the functor in next question, so you do not need to turn anything in for this question.

Like many other languages, ML provides facilities for modularization, using *signatures* and *structures*. Encapsulation is achieved by placing all related functions and types into a single module. Information is exposed to the outside world by placing it in the signature. Additionally, a signature may impose a narrower type definition in the signature than it might in the structure.

Using these modularization constructs, execute the following:

- Consider the following signature, `MLPQUEUE`. All of the function names and types visible to the outside world are declared here. Take note of the signature for `new` (not discussed earlier) which should evaluate to an empty queue:

```
signature MLPQUEUE =
sig
  type 'a mlqueue

exception Overflow
exception Empty
exception LevelNoExist
exception NotFound

val maxlevel : int
val new: 'a mlqueue
val enqueue : 'a mlqueue -> int -> int -> 'a -> 'a mlqueue
val dequeue : 'a mlqueue -> 'a * 'a mlqueue
val move : ('a -> bool) -> 'a mlqueue -> 'a mlqueue
val atlevel : 'a mlqueue -> int -> (int * 'a) list
val lookup : ('a -> bool) -> 'a mlqueue -> int * int
val isempty : 'a mlqueue -> bool
val flatten : 'a mlqueue -> 'a list
end;
```

- Write a functor `MakeQ` which takes a structure implementing MLQPARAM (shown below) as input and outputs a structure implementing MLPQUEUE with the `maxlevel` field fixed at "max" and the underlying queue data type set to "element." The functor should contain all of the function definitions from the previous section. See slides 34-36 of the lecture for a similar example.

```
signature MLQPARAM = sig
   type element;
   val max : int;
end;
```

- Write a test bed that invokes the library functions above. Your code should perform the following actions:
  1. Use the functor above to instantiate a structure with maximum level set to 2 (i.e., 0, 1, 2) and `int` as the underlying type.
  2. Enqueue the following elements (in the given sequence) in the queue (where `q` is the queue). The queue resulting from any given `enqueue` call should be passed to the next expression so that the operations are cumulative. For this example we assume your structure is named `maxLevel2PQueue`:
     (a) `maxLevel2PQueue.enqueue q 1 1 2`
     (b) `maxLevel2PQueue.enqueue q 0 0 3`
     (c) `maxLevel2PQueue.enqueue q 2 0 5`
     (d) `maxLevel2PQueue.enqueue q 2 2 1`

(e) `maxLevel2PQueue.enqueue q 1 0 4`

(f) `maxLevel2PQueue.enqueue q 2 1 6`

3. Raise an exception when element is enqueued at a level exceeding the maxlevel of the queue.
4. Move all the elements which are greater than 3 to the next lower level.
5. Dequeue two elements from the queue.
6. Query the priority queue for level 1.
7. Find the first element whose value is less than 5.

Frequently Asked Questions

1. **Q:** With respect to the `move` function, by "lowering" the level, you mean incrementing the current level in the tuple by 1 to make it "less" important?
   A: Yes
2. **Q:** The function should raise levelnoexist if level n does not exist and output empty list if no elements exist at a particular level. These requirements seem contradictory: if there's no level n then there are no elements that exist at level n?
   A: f max level is 5, for example, and you query what is at level 6, then the level does not exist and the exception is raised. If you query level 4, for example, and there are no items, the level exists but it will give you an empty list.
3. **Q:** Do the level and priority have a minimum like a min level is 1 or 0?
   A: Lowest level should be 0.
4. **Q:** Can I perform tests that are more exhaustive than the ones shown above?
   A: Yes, and you are encouraged to do so.