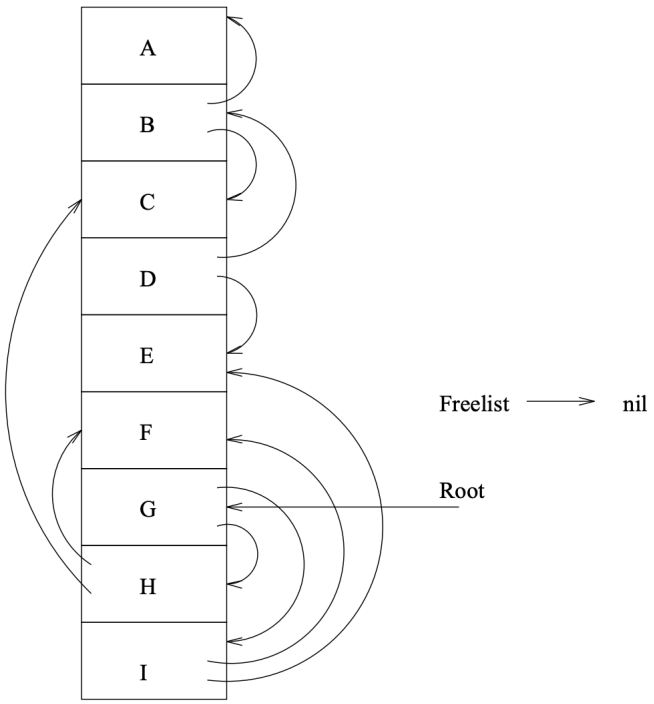
Spring 2023
Programming Languages
Homework 3

- This homework includes an ML programming assignment and short answer questions. You should use Standard ML of New Jersey (SML) for the programming portion of the assignment.
- Due Tuesday, April 11, 2023 at 11:59 PM Eastern Time. Submit two files: a PDF `<netid>-hw3.pdf` containing your solution to the short answer questions and another file `<netid>-hw3.sml` containing solutions to all of the ML questions.
- Late submissions are not accepted. **No exceptions will be made.**
- 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`. Stick to the feature set discussed in the lecture slides.
- 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. [5 points] **Garbage Collection - 1**



The above represents the heap space just before a garbage collection is invoked. Assuming the `Mark/Sweep` garbage collection:

1.1 Draw the heap after mark() step has been completed. Clearly mark all the cells (as per the algorithm mentioned in the lecture slides).

1.2 Draw the heap after the garbage collection is complete. Also, construct the free list resulting from the garbage collection.

2. [10 points] **Garbage Collection - 2**

Consider the two garbage collectors discussed in the class: `Mark/Sweep` and `Copying GC`. For each of the situations mentioned below, determine whether any of the above garbage collectors can be used or not. If yes, which garbage collection strategy is the fastest in terms of time complexity? Justify your answer. `Note:` Ignore the cost of allocation and tracing in your answers; focus on the cost of an actual garbage collection.

`Hint:` In each of the situations, suppose there is a heap containing N objects and sweeping an object costs M units of time. Using this, compute the time in both of the GC methods (keeping in mind that a particular garbage collector can not be used in some situations) and compare these times.

2.1 In a situation where the survival rate (percentage of live data after a garbage collection) of the data on heap is exactly 8% and it is 20 times more expensive to copy an object than to sweep an object.

2.2 In a situation where the survival rate (percentage of live data after a garbage collection) of the data on heap is exactly 15% and it is 5 times more expensive to copy an object than to sweep an object.

2.3 In a situation where the size of live data on heap is close to the size of the heap. No additional heap space is available.

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

For each of the following, come up with a free list (minimum 3 non-zero entries) and sequence of allocation requests (minimum 3 non-zero requests) that will result in the following outcomes (you are allowed to create different free list as well as different allocation requests in each part):

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

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

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

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

Note:

If there is a free block of size `s` available and the allocation request is for `x` size (where `x <= s`), then the free list will entry of size `s` will be reduced to size `s-x` and the pointer to the beginning of the free memory will be updated accordingly. For this problem, however, you can write the free list as a list of available block sizes and not worry about the pointer to memory.

4. [15 points] **ML Function Signature**

4.1 Write a SML function `test1` that satisfies the following type signature:

```
val test1 = fn : 'a -> 'a list -> 'a list
```

4.2 Write a SML function `test2` that satisfies the following type signature:

```
val test2 = fn : 'a -> 'b list -> 'a list
```

4.3 Write a SML function `test2` that satisfies the following type signature:

```
val test2 = fn : ( 'a * 'b -> 'b) -> 'a list -> 'b -> 'b
```

4.4 Write a SML function `test2` that satisfies the following type signature:

```
val test2 = fn : ('a * 'a list -> 'a list) -> 'a list -> 'a list -> 'a list
```

4.5 Given the following datatype:

```
datatype ('a,'b) tree = Leaf of 'a
                      | Node of 'a * 'b;
```

Write a SML function `test3` that satisfies the following type signature:

```
val test3 = fn : ('a,'a) tree -> ('a -> 'b) -> ('b -> 'a -> 'b) -> ('a,'b) tree
```

`Note:`

- You may be tempted to declare functions that satisfy a part of the signature and use these functions in the `test1`, `test2` and `test3` functions. However, in this problem, you are not allowed to do so. For example, in the function `test3`, you are not allowed to define a separate function which has signature `('c -> 'b -> 'c)` and then use this function in `test3`.
  Instead, try to think of ways you can use a function in order to restrict the signature of the function to a certain type. Thus in `test3`, try to come up with a way to use a function (say, `func3`) such that its signature is `('c -> 'b -> 'c)` without actually defining `func3`.

5. [30 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 without any special citation necessary.
- Helper functions are permitted but should be largely unnecessary in most cases. There are very clean, elegant definitions not requiring them.
- 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 seemingly 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 inside a function, any test bed you write that calls the function should handle the exception.
- 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.

5.1 Write a function `get_index : 'a list -> int -> 'a` which, given a list and index, retrieves the value of a zero-based index in the list. For example, `get_index [1,2,3] 1` evaluates to 2. Raise an exception if the index provided is not valid for the input list. To do this, declare `exception NoItem` at the top level. Then, from within your routine, write `raise NoItem`.

5.2 Write a function `listsum : int list -> int -> bool`: given an input integer list and an integer n, compute the sum of the numbers in the input list and evaluate to `true` if the sum is n, or `false` otherwise. **Do not** use recursion in your solution. See the lecture slides for ideas.

5.3 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.

5.4 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.

5.5 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]

[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 parameterized by a type variable.

(`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.

5.6 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 (if n is non-positive or n is greater than the length of list) 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
```

5.7 Write a function

```
lookup : (string * int) list -> string -> int option
```

that takes a list of pairs `(s, i)` and also a string `s2` to look up. It then goes through the list of pairs looking for the string `s2` in the first component. If it finds a match with corresponding number `i`, then it returns `SOME i`. If it does not, it returns `NONE`.

Example:

```
lookup [("hello",1), ("world", 2)] "hello";
val it = SOME 1 : int option

lookup [("hello",1), ("world", 2)] "world";
val it = SOME 2 : int option

lookup [("hello",1), ("world", 2)] "he";
val it = NONE : int option
```

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.

6. [25 points] **Multi-Level Priority Queue in ML**

1 Background

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 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.

2 Data Structure

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:

```
(* The maximum level (>= 1) supported by the multi-level queue *)
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
```