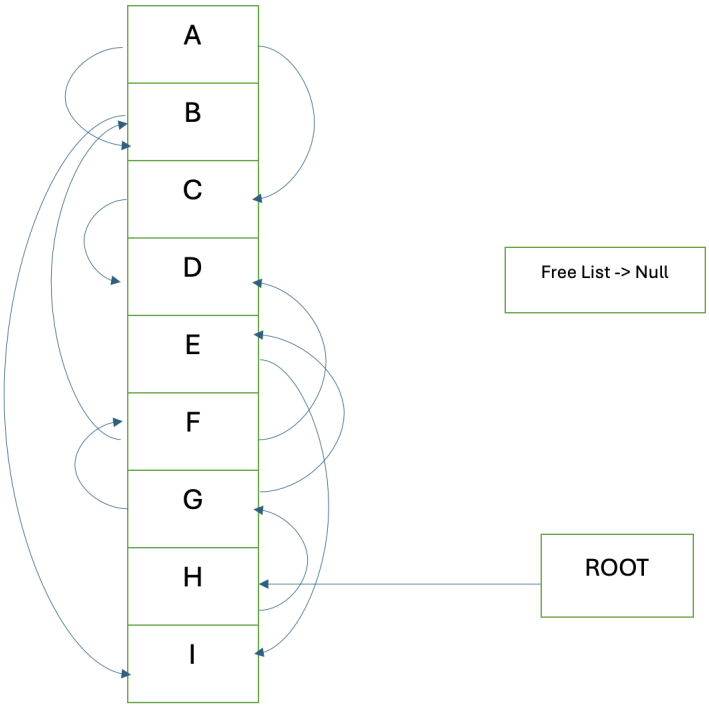
Spring 2024
Programming Languages
Homework 3

- 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 Sunday, April 14, 2024 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 following Java code snippet. This code, which runs only once and contains no loops, creates multiple Node objects that are linked to each other. Your task is to simulate the execution of this code and illustrate the state of the heap at various points in time using the copying garbage collection (GC) algorithm. Specifically, you should:

- Draw the heap before the execution of the line marked with (1). Your drawing should include all objects in the heap, the links between them, and the forwarding addresses of each object.
- Draw the heap after the garbage collection has been completed. Create a representation of the FROM and TO spaces.

```
public class Main {
    public static void main(String[] args) {
        Node a = new Node();
        Node b = new Node();
        Node c = new Node();
        Node d = new Node();
        Node e = new Node();
        Node f = new Node();
        a.next = b;
        b.next = c;
        foo(a, d);
        bar(c, e);
        bar(d, f);
        a = null;
        b = null;
        c = null; ----------------> (1)
    }

    public static void foo(Node param1, Node param2) {
        Node g = new Node();
        param1.next.next = g;
        g.next = param2;
    }

    public static void bar(Node param1, Node param2) {
        Node h = new Node();
        param1.next = h;
        h.next = param2;
    }

}

class Node {
    Node next;
}
```

Note: In this code, Node is a simple class that has a next field, which can hold a reference to another Node object.

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 First-fit and best-fit allocation can satisfy all requests, but Worst-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 * 'b list -> 'a list
```

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

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

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

```
val test3 = fn : ( 'a * 'b -> 'b) -> 'a list -> 'b -> 'b
```

4.4 Write a SML function `test4` that satisfies the following type signature:

```
val test4 = 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 `test5` that satisfies the following type signature:

```
val test5 = 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 all the `test`, 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 `('a * 'b -> 'b)` 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 `('a * 'b -> 'b)` 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 `scheme_intr = 'a list * string -> 'a list` which, given a list and command 'cdr', retrieves the remaining list. If the command given is 'car', it retrieves the first element of the list. For example, `scheme_intr([1,2,3],"cdr")` evaluates to [2,3]. `scheme_intr([1,2,3],"car")` evaluates to [1]. Raise an exception if the command provided is not valid for the input list. To do this, declare `exception Invalid Command` at the top level. Then, from within your routine, write `raise Invalid Command`.

5.2 Write a function `half_sum : int list -> int -> bool`: given an input integer list and an integer n, compute the sum of half of each element in the input list (divide each number by 2 and then sum) 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 You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, and the only constraint stopping you from robbing every house is that adjacent houses have a security system connected, and it will automatically contact the police if two adjacent houses are broken into on the same night. You must determine the maximum amount of money you can rob tonight without alerting the police.

Write a function `max_profit : int list -> int` in SML/NJ that takes a list representing the amount of money in each house and returns the maximum amount of money you can rob without alerting the police.

For example:

`max_profit [1, 2, 3, 1]` should return 4, as you can rob the first and third houses for a total of 4 without alerting the police.

`max_profit [2, 7, 9, 3, 1]` should return 12, as you can rob the first, third and fifth houses for a total of 12 without alerting the police.

5.4 Write a function `skip_num : a list * int -> 'a list`: given an input list and an integer n. It skips every n elements in the list and returns the resulting list. For example `skip_num([1,2,3,4,5] 2` evaluates to [1,4].

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] (`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
```

5.8 Write a function `spiral_sort: int list-> int list` which sorts a list of integers in spiraling order. That is, it alternatively prints the two halves of the sorted list. You may use a predefined sorting implementation—either quicksort from the slides, or some other sorting routine that you should cite to properly. Example:

```
spiral_sort [1, 2, 3, 4, 5, 6, 7];
val it = [5,1,6,2,7,3,4]

spiral_sort [3, 6, 1, 7, 5, 2, 4];
```

[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.

```
val it = [5,1,6,2,7,3,4]

spiral_sort [9, 2, 6, 5, 6]
val it = [6,2,9,5,6]

spiral_sort [1,2,3,4];
val it = [3,1,4,2]
```

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] **Circular Stack in ML**

1 Background

In the previous section, we implemented a hodgepodge of various functions serving various purposes. In this section, we will work on implementing a single data structure, organizing it appropriately, and making it generic.

A *circular stack* is an ordered collection of stacks such that each stack operation (e.g. push, pop, etc.) on the circular stack is directed toward one of the stacks in the collection, starting with the first, and continuing in a "round robin" manner for each subsequent call. Note that this circular stack concept is made up for the purposes of this assignment and is not a generally known data structure.

The size of the circular stack—that is, the number of underlying stacks—will be unique to each circular stack. Thus, there is no fixed size. All of the stacks in the circular stack must be of the same type, since there is a common interface being used for all of the individual stack operations.

2 Data Structure

Implement a circular stack ML structure which ascribes to the following slightly modified version of the STACKS interface:

```
signature STACKS =
sig
    type stack
    exception Underflow
    val empty : int -> stack
    val push : int * stack -> stack
    val pop : stack -> int * stack
    val isEmpty : stack -> bool
end
```

This is similar to the signature as shown on slide 34 of the ML lecture, with a couple exceptions:

- The type of the items that can be placed onto the stacks is fixed as 'int', as can be seen in the `push` and `pop` signatures. We will address this limitation later.
- The `empty` function in this version requires an integer, indicating the number of stacks in the stack collection. This will effectively "initialize" a new circular stack of the specified size.

With respect to the second item above, you must use this parameter to set the size of the collection. This means that each circular stack you create using the call to `empty` may have a different size than some other circular stack. You may not "hard code" the size anywhere in your implementation. Once a circular stack has been created using the `empty` function, the number of underlying stacks should not change.

One reasonable implementation for type `stack` that I use is as follows:

```
type stack = int * (int list) list
```

The first component of the tuple is the "current" stack to be operated on[2] and the second item represents the state of the internal stacks. For clarity, here is the desired behavior when creating a new circular stack of size 4 and pushing the value 5 to the stack repeatedly. I'm using *push* to demonstrate, but I could have also called any of the circular stack routines to demonstrate more interesting behavior:

[2]This number will increase after each operation and eventually cycle back to the first stack.

```
val emptyStack = Stacks.empty 3;
val stack1 = Stacks.push (1, emptyStack); (* Push 1 onto the first stack *)
val stack2 = Stacks.push (2, stack1); (* Push 2 onto the second stack *)
val stack3 = Stacks.push (3, stack2); (* Push 3 onto the third stack *)
val stack4 = Stacks.push (4, stack3);
(* Push 4 onto the first stack again, demonstrating circularity *)
(* Expected Output for stack4: (1, [[4, 1], [2], [3]]) *)
val (popped1, stack5) = Stacks.pop stack4; (* Pop from the current stack *)
(* Expected Output for popped1: 4, for stack5: (2, [[4,1], [], [3]]) *)
val (popped2, stack6) = Stacks.pop stack5; (* Pop again *)
(* Expected Output for popped2: 3, for stack6: (0, [[4,1], [], []]) *)
val (popped3, stack7) = Stacks.pop stack6; (* Pop from the first stack *)
(* Expected Output for popped3: 4, for stack7: (1, [[1], [], []]) *)
```

While implementing your routines, you may call the built in ML List functions. See any SML/NJ reference for details about the available functions, or go to `https://smlfamily.github.io/Basis/list.html`.

To ensure your implementation is correct, I recommend creating a test bed that looks similar to above, demonstrating that the functions work as expected. Your tests should naturally have far more coverage than this simple example above, by exercising all of the stack routines, on circular stacks of various sizes. You should ensure that everything works properly.

The grader will be testing your routines using his own test bed that will not be published in advance. The emphasis of the test bed will be on performing normal operations and not trying to "break" your code through the exploitation of arcane corner cases. However, in general, when performing a nonsensical operation, such as (but not limited to) creating a circular stack of size 0 and then trying to perform any operation on it, the general expectation is that an exception of some kind be raised, either from your code or from the ML List routines. You do not have to worry about handling these exceptions. You need only ensure that exceptions are raised when it makes sense to do so. The exact exception names are unimportant for this assignment.

Use common sense here. There are many ways to generate invalid inputs to the circular stack routines. Please do not flood the discussion forum with questions about each and every scenario. Handle invalid inputs using exceptions. If you find it necessary to add exceptions to your signature, please feel free to do so.

3 Functor

The circular stack implementation you wrote above is fixed to items of type `int`. We will now address this limitation by moving the code into a functor. Write a functor named `MakeCircularStack` which parametrizes the type of the circular stack items. Obviously you will utilize the structure that you already wrote when implementing the functor. Only minor tweaks to your existing code should be necessary to get this working. See slides 39-40 for more information on functors and for ideas on how to do this.

The parameter to your functor must ascribe to this STACKTYPE interface:

```
signature STACKTYPE
  type element;
end;
```

We will call your functor (see slide 41 for the syntax), and pass structures ascribing to STACKTYPE in order to create new circular stack structures. For example, we might create a circular stack that can store strings, by writing write:

```
structure String : STACKTYPE =
 struct
     type element = string;
 end;

structure StringCircularStack = MakeCircularStack(String);

val newStringStack = StringCircularStack.empty 4;
StringCircularStack.push ("Hello world", newstringStack);
...
```

Since the existing stack implementation already supports a configurable number of internal stacks, there is no need to parametrize the circular stack size inside the STACKTYPE signature, which is why you do not see any mention of it above.

First we will create the new parametrized structure and then test the routines in this structure using the same exact routines we called above before you wrote the functor. The only difference is that the grader will choose arbitrary circular stack types to create and ensure that the implementation still works properly for these new arbitrary types.

For full credit, you must turn in *both* the circular stack implementation with the fixed 'int' type and also the functor which supports creating circular stacks of arbitrary types. Do not worry that much of the code between these two are shared—we will be expecting some code duplication here, for obvious reasons.