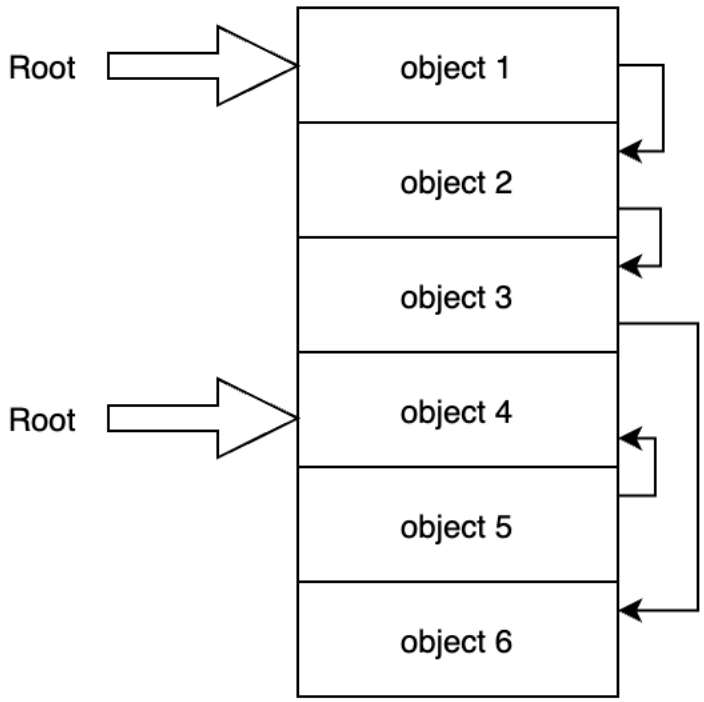
Fall 2021
Programming Languages
Homework 3

- 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 Monday, November 22, 2021 at 11:59 PM Eastern Standard 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 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. [15 points] **Garbage Collection**



The above represents the FROM heap for a program that utilizes copy collection. Assume that the roots point to objects 1 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. 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.

2. [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.

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.

3. [20 points] **Virtual Functions** [This topic will be covered in November 16th lecture]

This problem examines the difference between two forms of object assignment. In C++, local variables are stored on the run-time stack, while dynamically allocated data (created using the new keyword) is stored on the heap.

```
class Vehicle {
public:
    int x;
    virtual void f();
    void g();
};

class Airplane : public Vehicle {
public:
    int y;
    virtual void f();
    virtual void h();
};

void inHeap() {
    Vehicle *b1 = new Vehicle; // Allocate object on the heap
    Airplane *d1 = new Airplane; // Allocate object on the heap
    b1->x = 1;
    d1->x = 2;
    d1->y = 3;
    b1 = d1; // Assign derived class object to base class pointer
}

void onStack() {
    Vehicle b2; // Local object on the stack
    Airplane d2; // Local object on the stack
    b2.x = 4;
    d2.x = 5;
    d2.y = 6;
    b2 = d2; // Assign derived class object to base class variable
}

int main() {
    inHeap();
    onStack();
}
```

Answer the following questions:

Note: In the questions below, `vtable pointer` refers to the pointer of an object to its vtable

1. Draw a picture of the stack, heap and vtables that result after objects `b1` and `d1` have been allocated (but before the assignment `b1=d1`) during the call to `inHeap`. Be sure to indicate where the instance variables and `vtable pointers` of the two objects are stored before the assignment `b1=d1`, and to which vtable(s) the respective `vtable pointers` point.
2. Re-draw your diagram from `(1)`, showing the changes that result after the assignment `b1=d1`. Be sure to clearly indicate where `b1`'s vtable pointer points after the assignment `b1=d1`. Explain why `b1`'s vtable pointer points where it does after the assignment `b1=d1`.

3. Draw a picture of the stack, heap and vtables that result after objects `b2` and `d2` have been allocated (but before the assignment `b2=d2`) during the call to `onStack`. Be sure to indicate where the instance variables and `vtable pointers` of the two objects are stored before the assignment `b2=d2`, and to which vtables the respective `vtable pointers` point.
4. Re-draw your diagram from `(3)`, showing the changes that result after the assignment `b2=d2`. Be sure to clearly indicate where `b2`'s vtable pointer points after the assignment `b2=d2`. Explain why `b2`'s vtable pointer points where it does after the assignment `b2=d2`.
5. We have used assignment statements `b1=d1` and `b2=d2`. Why are the opposite statements `d1=b1` and `d2=b2` not allowed?

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 without any special citation necessary.
- 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.

1. Write a function

   ```
   alternate : int list -> int
   ```

   that takes a list of numbers and adds them with alternating sign (alternating between + and - sign, starting with +). For example alternate [1,2,3,4] = 1 - 2 + 3 - 4 = -2

   Example:

   ```
   alternate [1,2,3,4];
   val it = -2 : int
   ```

   Explanation:

   ```
   1 - 2 + 3 - 4 = -2
   1 has + sign
   2 has - sign
   3 has + sign
   4 has - sign
   ```

2. Write a function

   ```
   alternate2 : int list -> (int * int -> int) -> (int * int -> int) -> int
   ```

   that takes a list of numbers, two functions f and g as input and returns a number after alternating application of these functions to the list, i.e., given the list `[x1, x2, x3, x4, x5]`, the return value should be `g(f(g(f(x1, x2), x3), x4), x5)`.

   Example:

   ```
   alternate2 [1,2,3,4] op+ op-;
   val it = 4 : int;
   ```

3. Write a function

   ```
   splitup : int list -> int list * int list
   ```

   that given a list of integers creates a tuple of two lists of integers, first one containing the non-negative entries, the second containing the negative entries. Relative order must be preserved: All non-negative entries must appear in the same order in which they were on the original list, and similarly for the negative entries. Example:

```
- splitup [1,-2,-4,0,1,3];
val it = ([1,0,1,3],[-2,-4]) : int list * int list
```

4. 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
```

5. 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;
```

6. Write a function

```
zipRecycle: 'a list * b list -> ('a * 'b) list
```

   that takes a pair of lists (not necessarily of equal length) and processes them as follows:
   - when `length(first list) = length(second list)` : functionality same as `zip`
   Example:

```
zipRecycle ([1,2,3], ["a","b","c"]);
val it = [(1,"a"),(2,"b"),(3,"c")] : (int * string) list
```

   - when `length(first list) > length(second list)`: start processing the lists as in `zip` above and once the second list is exhausted, ignore the remaining elements
   Example:

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

   - when `length(first list) > length(second list)`: start processing the lists as in `zip` above and when first list is empty it starts recycling from its start until the other list completes.
   Example:

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

zipRecycle ([1,2,3], ["a","b","c","d","e","f","g"]);
val it = [(1,"a"),(2,"b"),(3,"c"),(1,"d"),(2,"e"),(3,"f"),(1,"g")] : (int * string) list
```

7. 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.

8. 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
   ```

9. 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
   ```

10. 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) [];
    stdIn:251.1-251.20 Warning: type vars not generalized because of
       value restriction are instantiated to dummy types (X1,X2,...)
    ```

[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.

```
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);
```

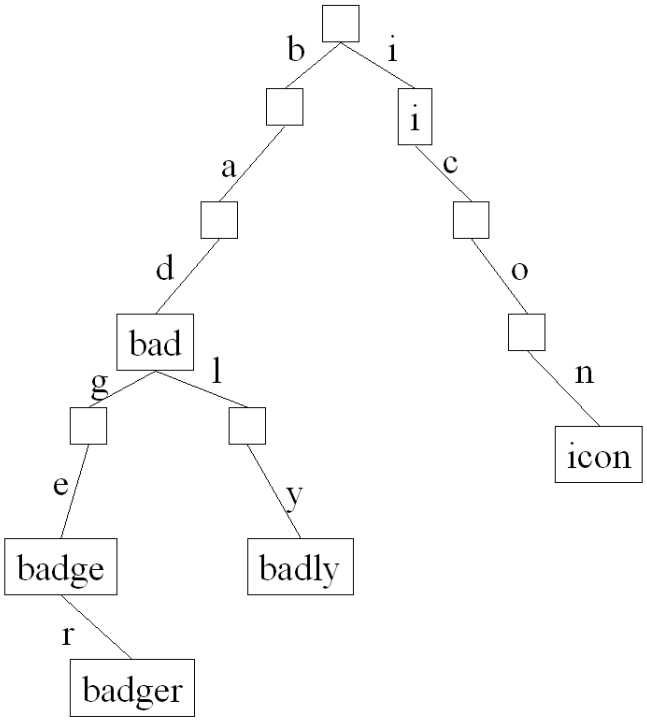
For those not familiar with this style of programming, `Using` evaluates to a value (in an object-oriented language, typically an object), upon which the `Select` method is invoked. This method evaluates to a new value (object), which is then used to call `From`, and so on. The functions are therefore 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] **Trie in ML**

A `trie` is an efficient data structure for indexing data based on lists of ordered keys. In this example, we are concerned with an implementation of a dictionary where keys are always string and values can be any arbitrary types or datatypes. A particularly useful instance of this idea considers a string as a list of characters.

Entries are found in a `trie` by starting at the root of the tree and following the appropriate branches until a labeled node is found. As an example, consider a `trie` indexed by lists of characters, where the data entry stored at a labeled node is the string representation of the word. For example, the string "badge" is found in the following tree by taking the path #"b", #"a", #"d", #"g", #"e".



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. We will use these features of ML in our `trie` implementation. Tries can implement all the operations of dictionaries. Consider the below signature of a `dictionary`:

```
signature DICT =
sig
  type key                          (* concrete type *)
  type 'a entry = key * 'a          (* concrete type *)

  type 'a dict                      (* abstract type *)

  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end;
```

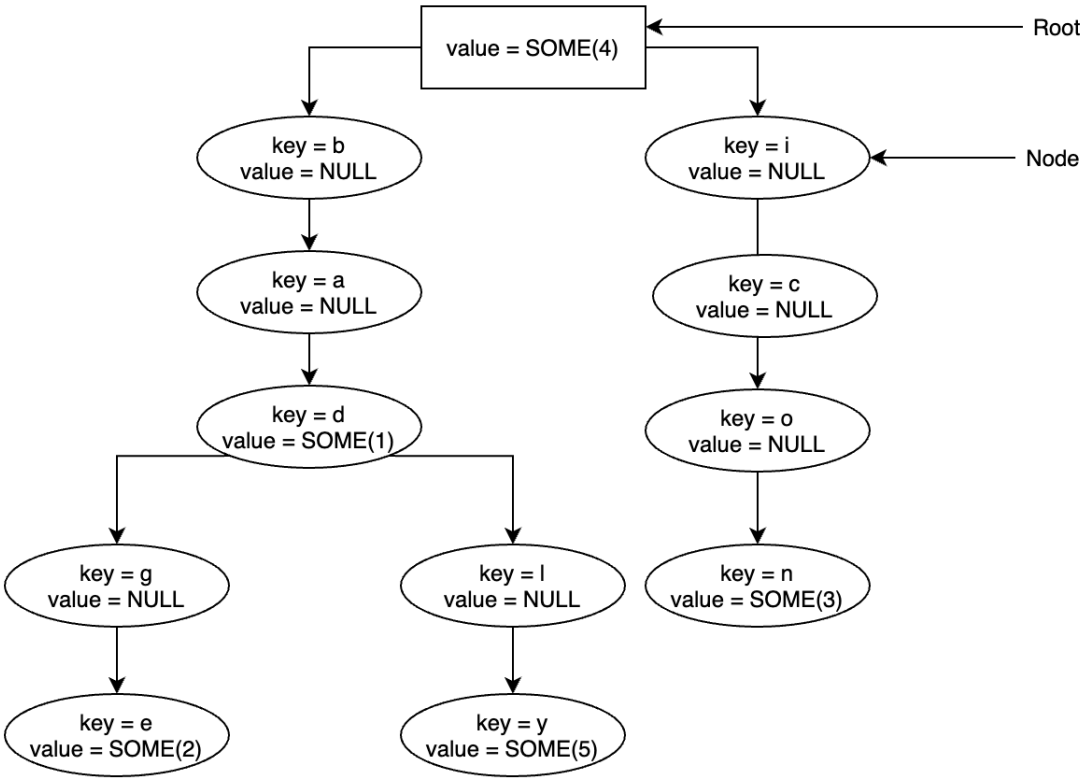
In this question, we will try to implement a `trie` using the signature of the above `dictionary`. Here is a partial implementation of the `Trie` structure:

```
structure Trie :> DICT =
struct
  type key = string
  type 'a entry = key * 'a

  datatype 'a trie =
    Root of 'a option * 'a trie list
  | Node of 'a option * char * 'a trie list

  type 'a dict = 'a trie

  val empty = Root(NONE, nil)

  (* val lookup: 'a dict -> key -> 'a option *)
  (* tries to find the key in the trie,
  * returns NONE if key is not found in the trie, otherwise
  * returns a SOME(value) corresponding to this key *)
  fun lookup trie key = (* TODO - function implementation here *)

  (* val insert: 'a dict * 'a entry -> 'a dict *)
  (* Inserts the key and value in the trie *)
  (* If the key is nil, assume that the Root is the destination *)
  fun insert (trie, (key, value)) = (* TODO - function implementation here *)

end
```

For example:

```
{
    "bad"     :    1,
    "badge"   :    2,
    "icon"    :    3,
    ""        :    4,  (* empty/nil key *)
    "badly".  :    5
}
```

This `trie` corresponding to the above dictionary can be diagrammatically represented as:



Your task is to:

- Provide an implementation of the functions, `lookup` and `insert`. Making helper functions is advised for a cleaner, elegant and easier to debug implementation.
- Write a test bed that tests the implementation. Make sure to test all the functionalities to receive full credits. Add these tests at the end of your `<netid>-hw3.sml` file and comment these out.