CSC/MAT-220: Lab 5

Due: 10/29/2018

In Lab 3, we reviewed aggregate data structures: a type of data that can be referenced as a single entity and yet consist of more than one piece of data. Examples where tuples and lists in SML. Tuples allowed for mixed types, but they have a fixed size. Lists, on the other hand, have a dynamic size, but cannot have mixed types. More precisely, a list is an example of a *type constructor*, or *parametrized type*, where the type of the list reveals the type of its elements. For example, consider the example below.

```
(* SML Example 1 *)
val x = [1,2,3];
val y = [#"A",#"B",#"C"];
val z = [true,false];
```

Run this example in SML, you will see that x is a list of type *int list*, y is a list of type *char list*, and z is a list of type *bool list*. Furthermore, the list is an example of a recursive data structure defined as follows:

i. The empty list *nil* is a value of type *typ list*,

ii. If h is a value of type *typ*, and t is a value of type *typ list*, then $h :: t$ is a value of type *typ list*

iii. Nothing else is a value of type *typ list*.

Note that *typ* denotes an arbitrary data type such as *int*, *char*, or *bool*. Recall that the binary constructor :: constructs a non-empty list from a head h of type *typ* and a tail t of type *typ list*. In general, all lists are of the form $[h_0, h_1, \ldots, h_{n-1}]$ and can be written recursively as follows

$$h_0 :: (h_1 :: (\cdots \ :: \ (h_{n-1} :: nil))),$$

for $n \geq 1$.

Computing with Lists

Since values of the list type are defined recursively, it is natural that functions that deal with lists be defined recursively. Moreover, we can do this recursion on the list itself, rather than on the size of the list. Finally, we can often write functions of an arbitrary type, which is denoted in SML by $'a$, rather than only working with int lists. Consider the example below that computes the length of a list.

```
(* SML Example 2 *)
val rec length:'a list->int =
fn nil => 0
| (_::t) => 1+length(t);
```

Since this function is the same no matter which type of list is chosen, it is said to be *polymorphic*. Verify this functions works by computing the length of the lists from SML Example 1. Note that the length function is built into SML's list structure. Similarly, the append function, which SML denotes by the infix notation as $l_1 @ l_2$, can be written as follows.

```
(* SML Example 3 *)
val rec append:'a list*'a list->'a list =
fn (nil,l) => l
| (h::t,l) => h::append(t,l);
```

Next, we use the append function to define another function *rev* for reversing the entries of a list.

```
(* SML Example 4 *)
val rec rev:'a list->'a list =
fn nil => nil
| (h::t) => append(rev(t),[h]);
```

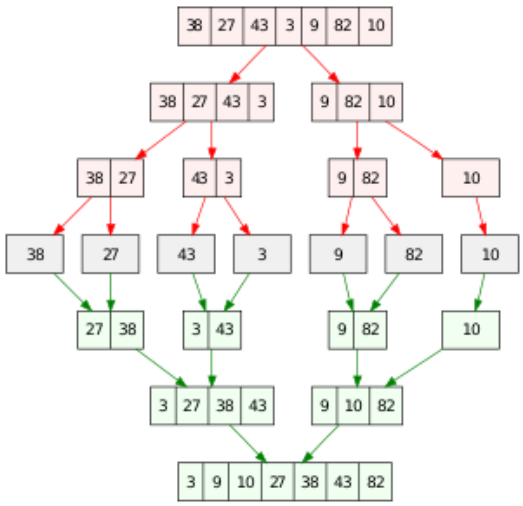
Verify this function works by reversing the lists from SML Example 1. While this seems to be the most natural way to define the *rev* function, it is not the most efficient. Below is a more efficient implementation of this function.

```
(* SML Example 5 *)
local
    val rec helper : 'a list*'a list -> 'a list =
    fn (nil,l) => l
    | (h::t,l) => helper(t,h::l)
in
    val rev : 'a list -> 'a list =
    fn l => helper(l,nil)
end;
```

Again, verify this function works by reversing the lists from SML Example 1. Note that the *helper* function is appending the head of the first argument to the second argument, rather than appending the entire tail of a list to its head as is done in SML Example 4.

Merge Sort

Merge sort is a divide and conquer algorithm developed by John von Neumann in 1945. This algorithm is capable of sorting a list in $O(n \log n)$ time. In essence, the algorithm recursively splits a list in half, sorts the smaller halves, and then merges them back together (see the figure below). Throughout this section, for simplicity, we will assume we are working with lists of type *int list*. However, all functions can be written in a polymorphic form.



In order to perform a merge sort, we need to be able to accomplish two tasks: a split and a merge. Below is a function that performs a split.

```
(* SML Example 6 *)
local
    val rec helper : int*int list*int list -> int list*int list =
    fn (0,x,y) => (rev(x),y)
    | (n,x,h::t) => helper(n-1,h::x,t)
in
    val split : int list -> int list*int list =
    fn l =>
    let
        val n = length(l) div 2
    in
        helper(n,nil,l)
    end
end;
```

Note that the function *split* determines the middle of the list l (rounding down if the length is odd) and then uses helper to return a tuple of the firs half and second half of the list. Below is a function that performs the merge.

```
(* SML Example 7 *)
val rec merge : int list*int list -> int list =
fn (nil,r) => r
| (l,nil) => l
| (l as h1::t1,r as h2::t2) =>
if h1 <= h2 then h1::merge(t1,r) else h2::merge(l,t2);
```

Note how we are able to use the *as* keyword to represent the left and right lists as both l and r, respectively, and as $h1 :: t1$ and $h2 :: t2$, respectively. Furthermore, note that *merge* only returns a sorted list when each list l and r is individually sorted.

Assignment

At the top of your file include a commented line with your name, lab number, and date. In addition, be sure to comment generously throughout so I can follow your steps. Finally, when you are done upload your source code to Dropbox. The file you you turn in should be labeled as follows: *yourname_labnumber.sml*.

I. Determine the time complexity of the *append* function from SML Example 3. Use this information to determine the time complexity of the *rev* function from SML Example 4. Show your work by setting up and solving a recursive formula and report your answer in big O notation.

II. Determine the time complexity of the *rev* function from SML Example 5. Again, show your work by setting up and solving a recursive formula and report your answer in big O notation. How much faster is *rev* from Example 5 than *rev* from Example 4?

III. Perform an evaluation trace of the function *split* from SML Example 6 on the list $x = [1, 2, 3, 4, 5]$.

IV. Perform an evaluation trace of the function *merge* from SML Example 7 on the tuple $([1, 3], [2, 4])$.

V. Complete the following code for the merge sort function, then test your function on several lists of length at least 10.

```
val rec merge_sort : int list -> int list =
fn nil => nil
| h::nil => [h]
| x =>
```