

## CS 3721: Programming Languages Lab

Lab #08 Solutions: Higher-Order Functions and Parametric Polymorphism in ML

**Due date: March 25, 3:30pm.** At the beginning of the next recitation.

Goals of this lab:

- Be able to read/write ML expressions utilizing `let`, `fn`, and lists to create higher-order polymorphic functions in ML.
1. *Lists*. A *list* is a finite sequence of elements. The elements of a list may have any type, including tuples and even other lists. Every element of a list must have the same type. That is, if the type of a list is  $\tau$  ( $\tau$  list), then all of its elements are of type  $\tau$ . Examples of empty list:

```
- []; (* Empty list. Its type is polymorphic *)
      (* because it contains a type variable. *)
      (* It can be regarded as having any type of elements. *)
val it = [] : 'a list
- nil; (* Synonym for the empty list. []. *)
val it = [] : 'a list
```

Please, **write down** the values and types of the following expressions.

```
[1, 2, 3]; ==> val it = [1,2,3] : int list
[3.3 + 4.2, (5.3 - 3.1) / 1.8, real(3)];
==> val it = [7.5,1.222222222222,3.0] : real list
[(10, "ten"), (20, "twenty"), (30, "thirty")];
==> val it = [(10,"ten"),(20,"twenty"),(30,"thirty")] : (int * string) list
[[3], [2, 7], [], [4, ~1]]; (* ~ is the unary negative operator. *)
==> val it = [[3],[2,7],[],[4,~1]] : int list list
```

*Building a list*. The infix operator `::` is used to construct a list. It is similar to the prefix `cons` operator in Scheme. Every list is either *nil*, or has the form  $x :: l$  where  $x$  is its *head* and  $l$  its *tail*. The tail itself is a list. Examples:

```
- 2 :: [];
val it = [2] : int list
- 3.1 :: [2.4];
val it = [3.1,2.4] : real list
- "spurs" :: ["nets", "rockets", "lakers"];
val it = ["spurs","nets","rockets","lakers"] : string list
```

Using the operator `::`, write a function *fromto* that builds the list of integers from  $m$  to  $n$ . Test your function with the following.

```
- fromto(3, 8);
val it = [3,4,5,6,7,8] : int list
```

**Solution:**

```
- fun fromto(m, n) = if m>n then [] else m :: fromto(m + 1, n);
val fromto = fn : int * int -> int list
```

*Fundamental list functions.* The function **null** tests whether a list is empty. The function **hd** returns the first element (head) of a non-empty list. The function **tl** returns the tail of a non-empty list. Please, **write down** the return values and types of the following expressions. If an error occurs, comment on its reason.

```

null([]); ==> type: true, return value: true
hd([1, 2, 3]); ==> type: int, return value: 1
hd(1.1 :: [2.2]); ==> type: real, return value: 1.1
tl([1, 2, 3]); ==> type: int list, return value: [2, 3]
tl(1 :: [2, 3]); ==> type: int list, return value: [2, 3]
null(tl(["spurs"])); ==> type: bool, return value: true
null(tl([2, 3])); ==> type: bool, return value: false
null(hd(["cs", "utsa"])); ==> "hd" function returns a string, not a list.
And, since "null" should take a list
as its argument, an error occurs.

```

2. *Local declarations.* ML provides *let* expression for local declarations. It has the following form:

```
let D in E end;
```

where D is a sequence of value declarations:  $D_1; D_2; \dots; D_n$ . And, each  $D_i$  has the form:

```
val <pattern> = <expression>
```

During evaluation, the declaration D is evaluated first. And, the environment created is only visible inside the *let* expression. Finally, the expression E is evaluated, and its value is returned. Now, using the *let* operator, convert the following Scheme code to ML. (Use pattern matching to access the head and tail of  $x$  and  $y$ .)

```
(let ((x '(1 2)) (y '(4 6)))
  (cons (+ (car x) (car y)) (cdr y)))
```

**Solution:**

```

(* Using functions: hd, tl: *)
let
  val x = [1, 2]
  val y = [4, 6]
in
  (hd(x) + hd(y)) :: tl(y)
end;

val it = [5,6] : int list (* Compiler output *)
(* Using pattern matching: *)
let
  val xHead::xTail = [1, 2]
  val yHead::yTail = [4, 6]
in
  (xHead + yHead) :: yTail
end;

```

3. *Function declarations (multiple-clause definition).* Last lab, we discussed single-clause defini-

tion. Now, it's time to talk about multiple-clause definition which has the form:

```
fun f( <pattern1>) = <expression1>
  | f( <pattern2>) = <expression2>
  ...
  | f( <patternN>) = <expressionN>
```

An example to multiple-clause definition is the following function which computes the product of a list of integers:

```
- fun prod(nil) = 1
  | prod(x::xs) = x * prod(xs);
val prod = fn : int list -> int    (* compiler output *)
- prod [3, 5, 8];    (* an application of the function *)
val it = 120 : int
```

When the function *prod* is applied to an argument, the clauses are matched in the order they are written. If the argument is an empty list, it matches the first clause and then the function returns the value 1. Otherwise, the argument is matched against the pattern given in the second clause (*x::xs*), and the code for the second branch is executed.

Now, write the *count* function which takes two arguments, a primitive value and a list. The function returns the number of times the value occurs in the list. Test your function with the following.

```
- count(5, [4, 3, 2, 5, 1, 5, 6]);
val it = 2 : int
```

**Solution:**

```
(* version1: *)
fun count(elem, nil) = 0
  | count(elem, x::xs) = if x=elem then (1 + count(elem, xs))
                        else count(elem, xs);

val count = fn : 'a * 'a list -> int    (* compiler output *)

(* version2: *)
fun count(elem, nil) = 0
  | count(elem, x::xs) = ((if x=elem then 1 else 0) + count(elem, xs));
```

4. *Anonymous functions.* The form of anonymous functions in ML is the following:

```
fn <pattern> => <expression>
```

and, it is like

```
(lambda (<parameters>) (<expression>))
```

in Scheme. Example:

```
- (fn (x, y) => x + y) (2,3);
val it = 5 : int
```

Now, translate the higher-order function *maplist* from lab2 into ML.

```
(define maplist (lambda (f myList)
  (cond
    ((null? myList) '())
    (else
     (cons (f (car myList)) (maplist f (cdr myList)))))))
```

**Solution:**

```
fun maplist(f, nil) = nil
  | maplist(f, x::xs) = f(x) :: maplist(f, xs);
```

Also, translate the following (call *maplist*) into ML.

```
(maplist (lambda (x) (* x x)) '(1 2 3 4 5))
```

**Solution:**

```
maplist((fn (x) => x * x), [1,2,3,4,5]);

val it = [1,4,9,16,25] : int list (* Compiler Output*)
```

What is the type of the function *maplist*?

**Solution:**

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

*maplist* function takes a function ('a -> 'b), **and** a list **of type** 'a; **and** returns a list **of type** 'b.

5. *Higher-order functions.* In this question, we will be reimplementing the higher-order functions *insert* and *sort* from lab4, but this time in ML. You may need to have a look at the solutions for lab4, <http://www.cs.utsa.edu/~vonronne/classes/cs3723-s08/lab04-solutions.pdf>.

- (a) Translate the *insert* function into ML. What is the type of *insert*?

**Solution:**

```
fun insert(func, x, nil) = x::[]
  | insert(func, x, l::ls) = if func(x, l) then x::l::ls
                             else l::insert(func, x, ls);

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

- (b) Translate the *sort* function into ML. What is the type of *sort*?

**Solution:**

```
fun sort(func, nil) = nil
  | sort(func, l::ls) = let
                          val sortedTail = sort(func, ls)
                        in
                          insert(func, l, sortedTail)
                        end;

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

- (c) Using *sort* function, sort the list [2, 9, 6, 7] in ascending and descending order. (*Hint*: Translate the functions *lte* and *gte* into ML.)

**Solution:**

```
fun lte(x, y) = x<=y;
fun gte(x, y) = x>=y;

(* Ascending order: *)
- sort(lte, [2, 9, 6, 7]);
val it = [2,6,7,9] : int list

(* Descending order: *)
- sort(gte, [2, 9, 6, 7]);
val it = [9,7,6,2] : int list
```

- (d) Using *sort* function, sort the list [[3, 0], [5, 8], [4, 9], [1, 3]] on the first element of each component. (*Hint*: Translate the function *sortOnFirst* into ML.)

**Solution:**

```
fun sortOnFirst(x::xs, y::ys) = x<=y;

- sort(sortOnFirst, [[3, 0], [5, 8], [4, 9], [1, 3]]);
val it = [[1,3],[3,0],[4,9],[5,8]] : int list list
```