

CS 3721: Programming Languages Lab

Lab #08: 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 τ (τ list), then all of its elements are of type τ . 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]; ==>
[3.3 + 4.2, (5.3 - 3.1) / 1.8, real(3)];
==>
[(10, "ten"), (20, "twenty"), (30, "thirty")];
==>
[[3], [2, 7], [], [4, ~1]]; (* ~ is the unary negative operator. *)
==>
```

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
```

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([]); ==>
hd([1, 2, 3]); ==>
hd(1.1 :: [2.2]); ==>
tl([1, 2, 3]); ==>
tl(1 :: [2, 3]); ==>
null(tl(["spurs"])); ==>
null(tl([2, 3])); ==>
null(hd(["cs", "utsa"])); ==>

```

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)))

```

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

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)))))))
```

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

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

What is the type of the function *maplist*?

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