

Introduction to SML

Higher-Order Functions

Michael R. Hansen

mrh@imm.dtu.dk

Informatics and Mathematical Modelling
Technical University of Denmark

Overview

Contents

- Higher-order functions
- Anonymous functions
- Higher-order list functions (in the library)
 - map
 - exists, all, filter
 - foldl, foldr
- Many recursive declarations follows the same schema.
 - Succinct declarations using higher-order functions.
- Parameterization of program modules

Higher-order functions

A function $f : \tau_1 \rightarrow \tau_2$ is a *higher-order function*, if a function type $\tau \rightarrow \tau'$ occurs in either τ_1 or τ_2 or both.

Functions are first class citizens

Higher-order functions

A function $f : \tau_1 \rightarrow \tau_2$ is a *higher-order function*, if a function type $\tau \rightarrow \tau'$ occurs in either τ_1 or τ_2 or both.

```
fun f x = let fun g y = x+y in g end;
```

```
> val f = fn : int -> int -> int
```

```
- f 2;
```

```
> val it = fn : int -> int
```

```
- it 3;
```

```
> val it = 5 : int
```

```
fun h g = g(g 2);
```

```
> val h = fn : (int -> int) -> int
```

```
- h (f 4);
```

```
> val it = 10 : int
```

Functions are first class citizens

Anonymous functions

Expressions denoting functions can be written using **fn** expressions:

$$\text{fn } pat_1 \Rightarrow e_1 \mid pat_2 \Rightarrow e_2 \mid \cdots \mid pat_n \Rightarrow e_n$$

Yields the value obtained by evaluation of the expression:

```
let fun f x = case x of
    pat1 => e1 | pat2 => e2 | ... | patn => en
in f end
```

Examples:

```
fn n => 2 * n;
```

```
fn 0 => false | _ => true;
```

```
fn r => Math.pi * r * r;
```

Exercise

Simplify

```
fun complement(Q, S, d, q0, F) =  
  let  
    fun F' q = not (F q)  
  in  
    (Q, S, d, q0, F')  
  end;
```

Declarations having the same structure

```
fun posList [] = []  
  | posList (x::xs) = (x > 0)::posList xs;  
val posList = fn : int list -> bool list
```

```
posList [4, ~5, 6];  
> val it = [true,false,true] : bool list
```

Applies the function `fn x => x > 0` to each element in a list

```
fun addElems [] = []  
  | addElems ((x,y)::zs) = (x + y)::addElems zs;  
> val addElems = fn : (int * int) list -> int list
```

```
addElems [(1,2),(3,4)];  
> val it = [3, 7] : int list
```

Applies the sum function `op+` to each pair of integers in a list

The function: `map`

Applies a function to each element in a list

$$\text{map } f [v_1, v_2, \dots, v_n] = [f(v_1), f(v_2), \dots, f(v_n)]$$

Declaration

Library function

```
fun map f = fn [] => []  
           | (x::xs) => f x :: map f xs;  
> val map = fn : ('a -> 'b) -> 'a list -> 'b list
```

Succinct declarations can be achieved using `map`, e.g.

```
val posList = map (fn x => x > 0);  
> val posList = fn : int list -> bool list
```

```
val addElems = map op+  
- val addElems = fn : (int * int) list -> int list
```

Declaration of higher-order functions

Commonly used form

```
fun map f []          = []  
    | map f (x::xs) = f x :: map f xs;  
> val map = fn : ('a -> 'b) -> 'a list -> 'b list
```

General form

```
fun  f pat11 pat12 ... pat1n = e1  
    | f pat21 pat22 ... pat2n = e2  
    | ...  
    | f patk1 patk2 ... patkn = ek
```

Exercise

Declare a function

$$g [x_1, \dots, x_n] = [x_1^2 + 1, \dots, x_n^2 + 1]$$

Remember

$$\text{map } f [v_1, v_2, \dots, v_n] = [f(v_1), f(v_2), \dots, f(v_n)]$$

Higher-order list functions: `exists`

$$\text{exists } p \text{ } xs = \begin{cases} \text{true} & \text{if } p(x) = \text{true} \text{ for some } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Declaration

Library function

```
fun exists p []          = false
  | exists p (x::xs) = p x orelse exists p xs;
> val exists = fn: ('a -> bool) -> 'a list -> bool
```

Example

```
exists (fn x => x>=2) [1,3,1,4];
> val it = true : bool
```

Exercise

Declare `member` function using `exists`.

```
infix member;
```

```
fun x member ys = exists      ????? ;  
> val member = fn : 'a * 'a list -> bool
```

Remember

$$\text{exists } p \text{ } xs = \begin{cases} \text{true} & \text{if } p(x) = \text{true for some } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Higher-order list functions: `all`

$$\text{all } p \text{ } xs = \begin{cases} \text{true} & \text{if } p(x) = \text{true}, \text{ for all elements } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Declaration

Library function

```
fun all p [] = true
  | all p (x::xs) = p x andalso all p xs;
> val all = fn: ('a -> bool) -> 'a list -> bool
```

Example

```
all (fn x => x>=2) [1,3,1,4];
> val it = false : bool
```

Exercise

Declare a function

`disjoint(xs, ys)`

which is true when there are no common elements in the lists `xs` and `ys`, and false otherwise.

Declare a function

`subset(xs, ys)`

which is true when every element in the lists `xs` is in `ys`, and false otherwise.

Remember

$$\text{all } p \text{ } xs = \begin{cases} \text{true} & \text{if } p(x) = \text{true, for all elements } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Higher-order list functions: `filter`

`filter p xs` is the list of those elements `x` of `xs` where `p(x) = true`.

Declaration

Library function

```
fun filter p []          = []  
  | filter p (x::xs) = if p x then x :: filter p xs  
                      else filter p xs;  
> val filter = fn: ('a -> bool) -> 'a list -> 'a list
```

Example

```
filter Char.isAlpha ["1", "p", "F", "-"];  
> val it = ["p", "F"] : char list
```

where `Char.isAlpha c` is true iff $c \in \{"A", \dots, "Z"\} \cup \{"a", \dots, "z"\}$

Exercise

Declare a function

`inter(xs,ys)`

which contains the common elements of the lists `xs` and `ys` — i.e. their intersection.

Remember `filter p xs` is the list of those elements `x` of `xs` where `p(x) = true`.

Higher-order list functions: **foldr** (1)

foldr 'accumulates' a function f from a 'start value' b over the elements of a list $[x_1, x_2, \dots, x_n]$ (from right to left):

$$\text{foldr } f \ b \ [x_1, x_2, \dots, x_{n-1}, x_n] = f(x_1, \underbrace{f(x_2, \dots, f(x_{n-1}, f(x_n, b)) \dots)}_{\text{foldr } f \ b \ [x_2, \dots, x_{n-1}, x_n]})$$

Declaration

Library function

```
fun foldr f b []          = b
  | foldr f b (x::xs) = f(x, foldr f b xs);
> val foldr =
    fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

Example: the length function

```
fun length xs = foldr (fn (_, y) => y+1) 0 xs;
> val length = fn : 'a list -> int
length [4,5,6];
> val it = 3 : int
```

Higher-order list functions: `foldr` (2)

Accumulation of an **infix operator** $\oplus$. Evaluation is as follows

$$\text{foldr } op \oplus b [x_1, x_2, \dots, x_n] \rightsquigarrow x_1 \oplus (x_2 \oplus \dots \oplus (x_n \oplus b) \dots)$$

Examples: Addition and Append

```
fun sumr xs = foldr op+ 0 xs;  
> val sumr = fn : int list -> int
```

```
sumr [1,2,3,4];  
> val it = 10 : int
```

```
fun append(xs,ys) = foldr op:: ys xs;  
> val append = fn : 'a list * 'a list -> 'a list
```

```
append([1,2,3],[4,5]);  
> val it = [1,2,3,4,5] : int list
```

Exercise: union of sets

Let an insertion function be declared by

```
fun insert(x, ys) = if x member ys then ys else x::ys
```

Declare a union function on sets.

Remember:

$$\text{foldr } op \oplus b [x_1, x_2, \dots, x_n] \rightsquigarrow x_1 \oplus (x_2 \oplus \dots \oplus (x_n \oplus b) \dots)$$

Higher-order list functions: `foldl` (1)

`foldl` 'accumulates' a function `f` from a 'start value' `b` over the elements of a list $[x_1, x_2, \dots, x_n]$ (from left to right):

$$\text{foldl } f \ b \ [x_1, x_2, \dots, x_{n-1}, x_n] = \underbrace{f(x_n, f(x_{n-1}, \dots, f(x_2, \overbrace{f(x_1, b)}^{b'})) \dots)}_{\text{foldl } f \ b' \ [x_2, \dots, x_{n-1}, x_n]}$$

Declaration

Library function

```
fun foldl f b [] = b
  | foldl f b (x::xs) = foldl f (f(x,b)) xs;
> val foldl =
    fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

Higher-order list functions: `foldl` (2)

Accumulation of an **infix operator** $\oplus$. Evaluation is as follows

$$\text{foldl } \text{op} \oplus b [x_1, x_2, \dots, x_n] \rightsquigarrow (x_n \oplus \dots \oplus (x_2 \oplus (x_1 \oplus b)) \dots)$$

Examples

```
fun rev xs = foldl op:: [] xs;  
> val rev = fn : 'a list -> 'a list
```

```
rev [1,2,3];  
> val it = [3, 2, 1] : int list
```