

A Third Look At ML

Additional information

- [Programming in ML](#)
- [A Gentle Introduction to ML](#)
- [Source level debugging in Poly/ML](#)

[Download as Power Point file for saving or printing.](#)

Outline

- More pattern matching
- Function values and anonymous functions
- Higher-order functions and currying
- Predefined higher-order functions

More Pattern-Matching

- Basic pattern-matching in function definitions:

```
□ fun f 0 = "zero"  
  |   f _ = "non-zero";
```

- Pattern-matching occurs in several other kinds of ML expressions:

```
□ fun f n =  
  case n of  
    0 => "zero" |  
    _ => "non-zero";
```

Match Syntax

- A *rule* is a piece of ML syntax that looks like this:

$$\langle rule \rangle ::= \langle pattern \rangle \Rightarrow \langle expression \rangle$$

- A *match* consists of one or more rules separated by a vertical bar, like this:

$$\langle match \rangle ::= \langle rule \rangle \mid \langle rule \rangle \mid \langle match \rangle$$

- Each rule in a match must have the same type of expression on the right-hand side
- A match is not an expression by itself, but forms a part of several kinds of ML expressions

Case Expressions

```
case 1+1 of
  3 => "three" |
  2 => "two" |
  _  => "hmm";
val it = "two" : string
```

- The syntax is
 $\langle \textit{case-expr} \rangle ::= \mathbf{case} \langle \textit{expression} \rangle \mathbf{of} \langle \textit{match} \rangle$
- This is a very powerful case construct—unlike many languages, it does more than just compare with constants

Example

```
val x = [4,5];  
  
case x of  
  _ :: _ :: c :: _ => c |  
  _ :: b :: _     => b |  
  a :: _         => a |  
  nil           => 0;
```

The value of this expression is the third element of the list **x**, if it has at least three, or the second element if **x** has only two, or the first element if **x** has only one, or 0 if **x** is empty.

Result is 5.

Examples

The following functions on left are equivalent to those at right:

```
fun fact 0 = 1
| fact n = n * fact(n-1);
```

fact 5 returns 120

```
fun fact n =
  case n of
    0 => 1
  | _ => n * fact (n-1);
```

```
fun rac [a]=a
| rac (h::t) = rac t;
```

rac [1,2,3,4] returns 4

```
fun rac a =
  case a of
    [h] => h
  | h::t => rac t;
```

Examples

The function on top is equivalent to that at bottom:

```
fun dist (a,[]) = [[a]]
| dist (a,[h]) = [[a]]@[[a,h]]
| dist (a,(h::t)) = [[a,h]]@[(a::(h::t))>@dist(a,t)];
```

dist(1,[2,3,4]) returns *[[1, 2], [1, 2, 3, 4], [1, 3], [1, 3, 4], [1], [1, 4]]*

```
fun dist (a,L) =
  case L of
    [] => [[a]]
  | [h] => [[a]]@[[a,h]]
  | (h::t) => [[a,h]]@[(a::(h::t))>@dist(a,t)];
```

Generalizes **if**

```
if  $exp_1$  then  $exp_2$  else  $exp_3$ 
```

```
case  $exp_1$  of  
  true =>  $exp_2$  |  
  false =>  $exp_3$ 
```

- The two expressions above are equivalent
- So **if-then-else** is really just a special case of **case**

Example

```
fun union ([],L) = L
|   union (h1::t1, L) =
    if member(h1,L)
    then union(t1,L)
    else (h1::union(t1,L));
```

```
fun union ([], L) = L
|   union (h1::t1,L) =
    case member(h1,L) of
        true => union(t1,L)
    |   false => (h1::union(t1,L));
```

Behind the Scenes

- Expressions using **if** are actually treated as abbreviations for **case** expressions
- This explains some odd SML/NJ error messages:

```
if 1=1 then 1 else 1.0;
```

```
Error: types of rules don't agree [literal]
  earlier rule(s): bool -> int
  this rule: bool -> real
in rule:
  false => 1.0
```

Exercise 1

Rewrite the following using *case*

a)

```
fun insert(a,L) =  
    if null L then [a]  
    else if a < hd L then a::L  
        else (hd L)::insert(a, tl L);
```

b)

```
fun member (a,[]) = false  
| member (a,h::t) =  
    if a=h  
    then true  
    else member (a,t);
```

Outline

- More pattern matching
- **Function values and anonymous functions**
- Higher-order functions and currying
- Predefined higher-order functions

Predefined Functions

- When an ML language system starts, there are many predefined variables
- Some are bound to functions:

```
ord;  
val it = fn : char -> int  
  
~;  
val it = fn : int -> int  
  
/;  
val it = fn : real * real -> real
```

Defining Functions

- **fun** notation for defining new named functions
- New names for old functions using **val** just as for other kinds of values:

```
fun add x y = x + y;
      val add = fn : int -> int -> int
val addxy = add;
      val addxy = fn : int -> int -> int
addxy 4 3; val it = 7 : int

val add4y = add 4;
      val add4y = fn : int -> int
add4y 3;   val it = 7 : int
```

Function Values

- Functions in ML *do not have names*
- Function values are bound to variables, as with other kinds of values
- The **fun** syntax does two separate things:
 - Creates a new function value
 - Binds that function value to a name

Anonymous Functions

*An unbound
function value*

- Named function: value bound to f

```
fun f x = x + 2;  
val f = fn : int -> int  
  
f 1;  
val it = 3 : int
```

- Anonymous function: unbound function value

```
fn x => x + 2;  
val it = fn : int -> int  
  
(fn x => x + 2) 1;  
val it = 3 : int
```

The **fn** Syntax

- Another use of the match syntax

$\langle \text{fun-expr} \rangle ::= \mathbf{fn} \langle \text{match} \rangle$

- **fn** yields an expression with (anonymous) function value
- Can define **fun** in terms of **val** and **fn**
- These two definitions have the same effect:
 - **fun f x = x + 2**
 - **val f = fn x => x + 2**

Examples - Anonymous Functions

- Named functions:

```
fun inc x = x + 1;  
  val inc = fn : int -> int  
inc 5;  
  
fun add (a,b) = a + b;  
  val intBefore = fn : int * int -> int  
add (3,4);
```

- Anonymous functions:

```
(fn x=>x+1) 5;  
  val it = 6 : int  
  
(fn (a,b) => a+b) (3,4);  
  val it = 7 : int
```

Examples - Anonymous Functions

Use – Rather than naming a *non-recursive* function used only once.

```
fun map (f, []) = []  
|      map (f, (h::t)) = (f h) :: (map (f, t)) ;
```

- Without **fn**:

```
fun inc x = x+1;  
map (inc, [1,2,3]); returns [2, 3, 4]
```

- With **fn**:

```
map (fn x=>x+1, [1,2,3]); returns [2, 3, 4]
```

Examples - Anonymous Functions

- Without **fn**:

```
fun add (a,b) = a + b;  
  val intBefore = fn : int * int -> int  
  
reduce add 0 [1,2,3,4];  
  val it = 10 : int
```

- With **fn**:

```
reduce (fn (a,b) => a+b) 0 [1,2,3,4];  
  val it = 10 : int  
  
reduce (fn (a,b) => a*b) 1 [1,2,3,4];  
  val it = 24 : int
```

Examples - Anonymous Functions

- Without **fn**:

```
fun intBefore (a,b) = a < b;  
  val intBefore = fn : int * int -> bool  
  
quicksort ([1,4,3,2,5], intBefore);  
  val it = [1,2,3,4,5] : int list
```

- With **fn**:

```
quicksort ([1,4,3,2,5], fn (a,b) => a<b);  
  val it = [1,2,3,4,5] : int list  
  
quicksort ([1,4,3,2,5], fn (a,b) => a>b);  
  val it = [5,4,3,2,1] : int list
```

Exercise 1.5 – Give results.

1. `fun f x = x > 3;`
2. `f 12;`
3. `val f = (fn x => x > 3);`
4. `(fn x => x > 3) 12;`
5. `map(f, [1,2,3,4,5]);`
6. `map (fn x => x > 3, [1,2,3,4,5]);`
7. `filter(f, [1,2,3,4,5]);`
8. `filter (fn x => x > 3, [1,2,3,4,5]);`
9. `fun f2(x,y) = x+y;`
10. `f2(2,3);`
11. `(fn (x,y)=>x+y) (2,3);`

Exercise 1.5 – Give as anonymous function and call.

12. `fun f12 x = x + 3;`

13. `fun f13 x = 3;`

14. `fun f14 x = true;`

15. `fun f15 (x, y) = x :: y;`

The **op** keyword

- Binary operators are special functions with operands before and after:

4 + 3

- To treat them as plain functions: to pass **<**, for example, as an argument of type:

int * int -> bool

- The keyword **op** before an operator gives the underlying function

- Call binary function as a regular function:

(op +) (4, 3);

Example - The **op** keyword

```
op *;
```

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

```
(op <) (3,4);
```

```
val it = true : bool
```

```
quicksort ([1,4,3,2,5], op <);
```

```
val it = [1,2,3,4,5] : int list
```

```
fun reduce f a []=a
```

```
| reduce f a [b]=f (a, b)
```

```
| reduce f a (h::t) = f (h, (reduce f a t));
```

```
reduce (op +) 0 [1,2,3,4];
```

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

Exercise 1.6 - The `op` keyword

```
fun map2 (f, [], []) = []  
|   map2 (f, (h1::t1), (h2::t2)) =  
      (f (h1, h2))::map2(f, t1, t2);
```

- Give vector addition for `[1,2,3]` and `[5,6,7]` using `map2`.

Functions returning functions

- Functions can return functions as a result:

```
fun inc x = x + 1;  
fun dubal x = x * 2;  
  
fun pick 1 = inc  
  |    pick 2 = dubal;  
  
pick 1;  
val it = fn : int -> int
```

- The function returned can then be called:

```
pick 1 4;  
val it = 5: int
```

Functions returning functions

- Functions can return functions as a result:

```
fun inc x = x + 1;
fun dubal x = x * 2;

fun pick 1 = inc
  | pick 2 = dubal;

val p = pick 1;
val p = fn : int -> int
```

- The function returned can then be called:

```
p 4;
val it = 5: int
```

Functions returning Anonymous functions

- Functions can return functions as a result:

```
fun pick 1 = fn x => x + 1
|   pick 2 = fn x => x * 2;

pick 1;
val it = fn : int -> int
```

- *pick* returns functions having same signature.
- Both functions have one int parameter and return int.
- The function returned can then be called:

```
pick 1 4;
val it = 5: int
```

Exercise 1.7 - Give results

1. `fun f1 x = if x > 0 then fn y => y - 1
 else fn y => y + 1;`

a) `f1 3;`

b) `f1 3 8;`

c) `f1;`

2. `fun f16 x = fn y => x + y;`

a) `f16 3;`

b) `f16 3 4;`

Outline

- More pattern matching
- Function values and anonymous functions
- Higher-order functions and currying
- Predefined higher-order functions

Higher-order Functions

- Every function has an *order*:
 - A function with no functions as a *parameter*, and does not *return* a function value, has *order 1*
 - A function with a function as a parameter or returns a function value has *order n+1*, where *n* is the order of its highest-order parameter or returned value
- The **quicksort** we just saw is a second-order function because it has one *function* parameter
- **fun eq (x:int,y:int) = x=y;** order 1
val eq = fn : int * int -> bool
- **fun comp (f,g,x) = f (g x);** order 3
val comp=fn : ('a->'b) * ('c->'a) * 'c->'b

Exercise 2

What is the order and an example.

int*real->bool parameters int & real tuple returns bool

fun f (x,y) = x > y; order 1

int->real->bool function of int returns function of real returns bool

fun h y = fn x => real y < x; order 2

(int->'a) -> int -> 'a (function of int that returns 'a) and an int; returns an 'a.
fun f g (x:int)=g x; order 3

a) **int -> int**

b) **int * int -> bool**

c) **int list -> int list**

d) **(int -> int) * (int -> int) -> (int -> int)**

e) **int -> bool -> real -> string**

f) **('a -> 'b) * 'a list -> 'b list**

fun f (fab,h)::t)=(fab h)::[];

What can you say about the function with this type?

('a -> 'b) * ('c -> 'a) -> 'c -> 'b

Exercise 2 Continued

Give the anonymous functions equivalent to:

a) `fun dubal x = 2 * x;`

b) `fun mult x y = x * y;`

Duplicate using anonymous functions.

a) `map dubal [1, 2, 3, 4];`

b) `reduce mult 1 [1, 2, 3, 4];`

What is the signature for:

a) `fun map f L`

b) `fun map (f, L)`

c) `fun reduce f a L`

d) `fun reduce (f, a, L)`

Currying – Functions return Functions

- Returning functions termed *currying* in ML
- Generally used to bind parameters of the returned function
- Function g returns an anonymous function with a bound:

```
fun g a = fn b => a+b;  
  val g = fn : int -> int -> int  
fun g a b = a+b;  
  val g = fn : int -> int -> int
```

- $g\ 3$ binds parameter a to 3:

```
g 3 returns fn b => 3+b  
  val it = fn : int -> int
```

- $g\ 3\ 4$ binds parameter a to 3 and b to 4 :

```
g 3 4 returns 3+4  
  val it = 7 : int
```

Curried Addition

```
fun f (a,b) = a+b;
```

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

```
fun g a = fn b => a+b;
```

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

```
f(2,3);
```

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

```
g 2 3;
```

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

- Remember that function application is left-associative
- So `g 2 3` means `((g 2) 3)`

Curried vs Uncurried Examples

- Uncurried – One parameter (i.e. may be a tuple)
 - `fun f x = x + 4;`
 - `fun f (x, y) = x + y;`
 - `fun f (x, y, z) = x + y + z;`
- Curried – One or more parameters (i.e. the addition of each parameter returns a different function).
 - `fun f x = x + 4;`
 - `fun f x y = x + y;`
 - `fun f x y z = x + y + z;`

Advantages Example

- No tuples: write `g 2 3` instead of `fn (2, 3)`
- Main advantage: specialize functions by binding particular initial parameters

```
fun g a = fn b => a+b;  
val g = fn : int -> int -> int
```

```
val add2 = g 2;  
val add2 = fn : int -> int
```

```
add2 3;  
val it = 5 : int
```

```
add2 10;  
val it = 12 : int
```

Advantages: Example

- Convert *binary* function to *unary* with fixed first parameter. Useful with *map*, *reduce*, ...

```
fun map f []=[]
|   map f (h::t) = (f h)::(map f t);

fun g a = fn b => a+b;
map (fn x=>x+1) [1,2,3];   returns [2, 3, 4]
map (g 1) [1,2,3];       returns [2, 3, 4]

fun h a = fn b => [a,b];
map (fn x=>[1,x]) [1,2,3];
                                returns [[1,1], [1,2], [1,3]]
map (h 1), [1,2,3]; returns [[1,1], [1,2], [1,3]]
```

Advantages: Example

- Like the previous **quicksort**
- But now, the comparison function is a first, curried parameter

```
quicksort (op <) [1,4,3,2,5];  
val it = [1,2,3,4,5] : int list  
  
val sortBackward = quicksort (op >);  
val sortBackward = fn : int list -> int list  
  
sortBackward [1,4,3,2,5];  
val it = [5,4,3,2,1] : int list
```

Advantages Example

- Compute function at runtime
- Signatures of computed functions must be identical

```
fun sortorder "ascending"  x y = x >= y
|   sortorder "descending" x y = x <= y;
val sortorder = fn:string -> int -> int -> bool
```

```
val orderup = sortorder "ascending";
val orderup = fn : int -> int -> bool
```

```
orderup 3 4;
val it = false : bool
```

```
orderup 4 3;
val it = true  : bool
```

Advantages: Example

- Convert *n-ary* function to *lower order* with fixed parameters with *map*, *reduce*, ...

```
fun sortorder "ascending"  x y = x >= y
|  sortorder "descending"  x y = x <= y;
val sortorder=fn:string -> int -> int -> bool

val ge = sortorder "ascending";
val orderup = fn : int -> int -> bool

val L = map (ge 3) [1,2,3,4];
val L = [true, true, true, false] : bool list

reduce (fn a => fn b => a andalso b) true L;
val it = false : bool
```

Multiple Curried Parameters

- Currying generalizes to any number of parameters

```
fun f (a, b, c) = a+b+c;  
val f = fn : int * int * int -> int  
  
fun g a = fn b => fn c => a+b+c;  
val g = fn : int -> int -> int -> int  
  
f (1,2,3);  
val it = 6 : int  
  
g 1 2 3;  
val it = 6 : int
```

Easier Notation for Currying

- Instead of writing intermediate anonymous functions explicitly:

```
fun g a = fn b => b>a;
```

- Write:

```
fun g a b = b>a;
```

- Generalizes for any number of curried arguments

```
fun g a b = b>a;  
    val g = fn : int -> int -> bool  
val gt1 = g 1;  
    val gt1 = fn : int -> bool  
gt1 5;  
    val it = true : bool  
map (g 1, [0,1,2]);  
    val it = [false,false,true] : bool list
```

Exercise 3

```
fun f a = fn b => a div b;  
fun g a b = a div b;
```

What is the result of:

1. **f;**
2. **g;**
3. **f 13;**
4. **g 13;**
5. **f 13 4;**
6. **val x = map (g 12);**
7. **map (g 12) [2,3,4];**
8. **x [2, 3, 4];**

Exercise 3 Continued

9. Write a curried function of two parameters that subtracts integer parameter 1 from integer parameter 2.
10. Use the previous function and *map* to subtract integer 10 from each integer of the list [1,2,3,4,5].
11. Write a curried function of two parameters that subtracts integer parameter 2 from integer parameter 1.
12. Use the previous function and *map* to subtract each integer of the list [1,2,3,4,5] from integer 10.
13. Write a curried function of two parameters that returns true when parameter 1 is greater than parameter 2.
14. Use the previous function and *filter* to return a list of integers greater than 3, similar to the following:

```
fun p x = x > 3;  
filter p [1,2,3,4,5]; returns [4,5]
```

Computing Anonymous Functions

- *map* has two parameters, a unary function *f* and list
- Returns function where *map* function has parameter *f* bound to *inc* and one unbound list parameter.
- *mapINC* is computed to be an unary function that increments every element of an integer list.

```
fun map f [] = []  
|   map f (h::t) = (f h)::(map f t);  
  
fun inc x = x + 1;  
  
val mapINC = map inc;      (* f bound to inc *)  
    val mapINC = fn : int list -> int list  
  
mapINC [1,2,3];          (* returns [2,3,4] *)
```

Automating Computing Functions

- *bu* has two parameters, a binary function *f* and *f*'s first parameter
- Returns unary function where *f* and *first* parameter are bound and second parameter is unbound.
- *bu* stands for *binary to unary*, since *bu* takes a binary function and returns a unary function.

```
fun bu f a = fn b => f(a, b);  
    val bu = fn: ('a*'b -> 'c) -> 'a->'b->'c  
bu (op +) 4 5;  
    val it = 9 : int  
val INC = bu (op +) 1;  
    val INC = fn : int -> int  
map INC [1,2,3]; (* returns [2,3,4] *)  
map (bu (op +) 1) [1,2,3]; (*returns [2,3,4]*)  
bu (op ::) 4 [5,6];  
    val it = [4,5,6] : int
```

Automating Computing Functions

- *rev* has one parameter, a binary function *f*
- Returns binary function where *f* is bound and the first and second parameters are reversed.
- *rev* stands for *reverse* the binary parameters.

```
fun bu f a = fn b => f(a, b);  
fun rev(f) = fn(a,b) => f(b,a);  
    val rev = fn : ('a*'b->'c) ->'b*'a ->'c  
(op -) (10,4);  
    val it = 6 : int  
rev (op -) (10, 4);  
    val it = ~6 : int  
bu (rev (op -)) 10 4;  
    val it = ~6 : int  
map (bu (rev (op -)) 4) [1,2,3,4,5];  
    val it = [~3, ~2, ~1, 0, 1] : int list
```

Exercise 3 Continued

```
fun bu f a = fn b => f a b;  
fun rev f = fn a => fn b => f b a;  
fun cons a b = a::b;  
fun map _ [] = []  
|   map f (h::t) = (f h) :: (map f t);  
fun add x y = x + y;
```

Give the results.

15. `bu cons 3 [4, 5];`

16. `bu cons 3;`

17. `map (bu cons 3) [[4, 5], [6], []];`

18. `val a1 = bu add 1;`

19. `val m = bu map a1;`

20. `m [1, 2, 3];`

21. `map m [[1, 2], [3, 4, 5], [6]];`

22. `map (bu map (bu add 1)) [[1, 2], [3, 4, 5], [6]];`

Computing Anonymous Functions

- Old *reduce* has parameters *f* and *a* fixed but repeatedly passed
- *staged_reduce* computes anonymous reduction function of one parameter
- Returns *red* function of one unbound parameter.

```
fun staged_reduce (f, a) =  
  let  
    fun red nil = a  
      |   red (h::t) = f (h, (red t))  
  in  
    red      (* return function red      *)  
  end;      (* with 1 unbound parameter *)  
  
staged_reduce (op +, 0) [1,2,3];      returns 6
```

Exercise 3 Continued

23. Give the results.

```
fun staged_reduce (f, a) =  
  let  
    fun red nil = a  
      |   red (h::t) = f h (red t)  
  in  
    red      (* return function red *)  
  end;  
  
fun DIV x y = x div y;
```

a. `val x = staged_reduce (DIV, 1);`

b. `x [100, 25, 5];`

Exercise 3 Continued

24. Give a staged version for *filter*.

```
fun staged_reduce (f, a) =
  let
    fun red nil = a
      |   red (h::t) = f h (red t)
  in
    red      (* return function red *)
  end;
```

```
fun filter p nil = nil
  |   filter p (h::t) =
      if (p h) then h::(filter p t)
      else filter p t;
fun lt3 x = x < 3;

filter lt3 [1,2,3,4];      returns [1,2]
```

Outline

- More pattern matching
- Function values and anonymous functions
- Higher-order functions and currying
- **Predefined higher-order functions**

Predefined Higher-Order Functions

- Three important predefined higher-order functions:
 - `map`
 - `foldr`
 - `foldl`
- Actually, `foldr` and `foldl` are very similar, as you might guess from the names

The **map** Function

Used to apply a *unary* function to every element of a list, and collect a list of results

```
map ~ [1,2,3,4];
```

```
val it = [~1,~2,~3,~4] : int list
```

```
map (fn x => x+1) [1,2,3,4];
```

```
val it = [2,3,4,5] : int list
```

```
map (fn x => x mod 2 = 0) [1,2,3,4];
```

```
val it = [false,true,false,true] : bool list
```

```
map (op +) [(1,2), (3,4), (5,6)];
```

```
val it = [3,7,11] : int list
```

A definition of **map** function

Repeated mapping of a *unary* function to every element of *one* list

```
fun map f [] = []  
|   map f (h::t) = (f h)::(map f t);  
    val map = fn : ('a -> 'b) -> 'a list -> 'b list
```

```
fun less a b = a<b;  
    val less = fn : int -> int -> bool  
less 3;  
    val it = fn : int -> bool  
map (less 3) [1,2,3,4];  
    val it = [false,false,false,true] : bool list  
fun list a L = a::L;  
    val list = fn : 'a -> 'a list -> 'a list  
list 1 [3];  
    val it = [1, 3] : int list  
list 1;  
    val it = fn : int list -> int list  
map (list 1) [[1],[2],[3]];  
    val it = [[1, 1], [1, 2], [1, 3]] : int list list
```

The `map` Function Is Curried

```
map;
```

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

```
val f = map (op +);
```

```
val f = fn : (int * int) list -> int list
```

```
f [(1,2), (3,4)];
```

```
val it = [3,7] : int list
```

```
map (op +) [(1,2), (3,4)];
```

```
val it = [3, 7] : int list
```

```
map (op @) ([1,2,3], [4,5,6]), ([7,8], [9]);
```

```
val it = [[1,2,3,4,5,6], [7,8,9]] : int list list
```

The **foldr** Function

- Nearly identical to **reduce**.
- Used to combine all the elements of a list
- For example, to add up all the elements of a list **x**, we could write **foldr (op +) 0 x**
- It takes a function f , a starting value c , and a list $x = [x_1, \dots, x_n]$ and computes:

$$f(x_1, f(x_2, \dots f(x_{n-1}, f(x_n, c)) \dots))$$

- So **foldr (op +) 0 [1, 2, 3, 4]** evaluates as $1+(2+(3+(4+0)))=10$

Definition of **foldr** and Examples

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

```
foldr (op -) 0 [1,2,3,4];
```

```
val it = ~2 : int (1-(2-(3-(4-0))))
```

```
foldr (op div ) 1 [100,100,10];
```

```
val it = 100 : int (1000 div(100 div(10 div 1)))
```

```
foldr (fn (x,y)=> if x<y then x else y) 9999 [4,2,8,5];
```

```
val it = 2 : int
```

```
foldr (op ::) [5] [1,2,3,4];
```

```
val it = [1,2,3,4,5] : int list
```

The `foldr` Function Is Curried

```
foldr;  
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b  
  
foldr (op +);  
val it = fn : int -> int list -> int  
  
foldr (op +) 0;  
val it = fn : int list -> int  
  
val addup = foldr (op +) 0;  
val addup = fn : int list -> int  
  
addup [1,2,3,4,5];  
val it = 15 : int
```

The `foldl` Function

- Used to combine all the elements of a list
- Same results as `foldr` in some cases

```
foldl (op +) 0 [1,2,3,4];  
val it = 10 : int
```

```
foldl (op * ) 1 [1,2,3,4];  
val it = 24 : int
```

The **foldl** Function

- To add up all the elements of a list **x**, we could write **foldl (op +) 0 x**
- It takes a function f , a starting value c , and a list $x = [x_1, \dots, x_n]$ and computes:

$$f(x_n, f(x_{n-1}, \dots f(x_2, f(x_1, c)) \dots))$$

- So **foldl (op +) 0 [1, 2, 3, 4]** evaluates as $4+(3+(2+(1+0)))=10$
- Remember, **foldr** did $1+(2+(3+(4+0)))=10$

The `foldl` Function

- `foldl` starts at the **left**, `foldr` starts at the **right**
- Difference does not matter when the function is *associative* and *commutative*, like `+` and `*`
- For other operations, such as `-` or `div`, it does matter

```
foldr (op ^) "" ["abc", "def", "ghi"];
```

```
val it = "abcdefghi" : string
```

```
foldl (op ^) "" ["abc", "def", "ghi"];
```

```
val it = "ghidefabc" : string
```

```
foldr (op -) 0 [1,2,3,4];
```

```
val it = ~2 : int           (1 - (2 - (3 - (4 - 0))))
```

```
foldl (op -) 0 [1,2,3,4];
```

```
val it = 2 : int           (4 - (3 - (2 - (1 - 0))))
```

Compare **foldl** **foldr**

- **foldl** $f(x_n, f(x_{n-1}, \dots f(x_2, f(x_1, c)) \dots))$
- **foldr** $f(x_1, f(x_2, \dots f(x_{n-1}, f(x_n, c)) \dots))$

```
fun cons (a,L) = a::L;
```

```
val cons = fn : 'a * 'a list -> 'a list
```

```
foldl cons [] [1,2,3];
```

```
val it = [3, 2, 1] : int list  
      cons (3, cons (2, cons (1, [])))
```

```
foldr cons [] [1,2,3];
```

```
val it = [1, 2, 3] : int list  
      cons (1, cons (2, cons (3, [])))
```

Exercise 4

1. Implement **foldl**. Use `rac` and `rdc`.

```
fun foldr f y [ ] = y
|   foldr f y (x::xs) = f(x, (foldr f y xs));

fun rac [a] = a
|   rac (_::t) = rac t;

fun rdc [ _ ] = [ ]
|   rdc (h::t) = h::rdc t;
```

Exercise 4

- **foldl** $f(x_n, f(x_{n-1}, \dots f(x_2, f(x_1, c)) \dots))$
- **foldr** $f(x_1, f(x_2, \dots f(x_{n-1}, f(x_n, c)) \dots))$

2. Evaluate:

foldl (op ::) [] [1,2,3];

foldr (op ::) [] [1,2,3];

.

• **foldr** (op div) 5 [100,50,10];

evaluates as $100 \text{ div } (50 \text{ div } (10 \text{ div } 5)) = 4$

• **foldl** (op div) 5 [10,50,100];

evaluates as $100 \text{ div } (50 \text{ div } (10 \text{ div } 5)) = 4$

Exercise 4

3. Evaluate:

```
foldl snoc [] [1,2,3];
```

```
foldr snoc [] [1,2,3];
```

.

- **foldr add 0 [1,2,3,4]** evaluates as `add (1,(add (2, add(3, add(4,0))))=10`
- **foldl add 0 [1,2,3,4]** evaluates as `add (4,(add (3, add(2, add(1,0))))=10`

```
fun snoc (a, []) = [a]  
| snoc (a, h::t) = h::snoc (a, t);
```

Debugging - Trace function entry and exit values.

```
PolyML.Compiler.debug := true;
open PolyML.Debug;
trace true;
fun fac 0 = 1
|   fac n = n * fac (n-1);

fac 3;
fac entered val n = 3
  fac entered val n = 2
    fac entered val n = 1
      fac entered
        fac returned 1
      fac returned 1
    fac returned 2
  fac returned 6
val it = 6 : int
```