Standard ML

lists

---

a list is an **immutable** finite sequence of elements

```
[3, 5, 9]: int list
["a", "list"]: str list
[]: 'a list
```

order matters

```
[1, 2, 3] <> [3, 2, 1];
```

and repetitions count

```
[3, 3, 3] <> [3];
```

elements may have any type

```
[(1,"One"),(2,"Two")] : (int*string) list
[[3.1],[],[5.7, ~0.6]]: real list list
```

... but all elements must have the same type

```
[5, "five"]; (*ERROR*)
```

the empty list has a polymorphic type

```
[]: 'a list
```

`nil` is a synonym of `[]`

```
nil;
```

---

building a list

a list is either *empty* or *a head followed by a tail*

`[1,2,3]` ➪ head: `1` tail: `[2,3]`

use the infix operator `::` (aka `cons`) to build a list

```
1 :: [2, 3];
```

```
1 :: 2 :: 3 :: [];
```

`::` associates to the right, so

```
x1 :: x2 :: ... :: xn :: nil
```

=

```
(x1 :: (x2 :: (... :: (xn :: nil)...)
```

`::` is a *constructor* so it can be used in patterns

```
fun replace_head (_::t) x = x :: t
  | replace_head [] _ = []
;
```

---

builtin fundamental functions

`null` - tests whether a list is empty

```
fun null [] = true
  | null (_::_) = false;
```

`hd` - evaluates to the head of a non-empty list

```
fun hd (x::_) = x;
```

```
hd[ [ [1,2], [3] ], [ [4] ] ];
```

```
hd it;
```

```
hd it;
```

`tl` - evaluates to the tail of a non-empty list

```
fun tl (_::xs) = xs;
```

```
tl ["how", "are", "you?"];
```

```
tl it;
```

```
tl it;
```

```
tl it;
```

---

example - building a list of integers

```
fun range (m, n) =
  if m = n then []
  else m :: (range (m+1, n));
```

```
range (2, 5);
```

```
infix --;
val op-- = range;
```

```
2 -- 5;
```

`take` and `drop`

$$xs = [x_1, x_2, x_3, \ldots, x_k, x_{k+1}, \ldots, x_n]$$

$$take(k, xs) = [x_1, x_2, x_3, \ldots, x_k]$$

$$drop(k, xs) = [x_{k+1}, \ldots, x_n]$$

the computation of `take`

```
fun take (0, _)     = []
  | take (i, x::xs) = x :: (take (i-1, xs));
```

```
take (3, [9,8,7,6,5,4])
9 :: take (2, [8,7,6,5,4])
9 :: (8 :: take (1, [7,6,5,4]))
9 :: (8 :: (7 :: take (0, [6,5,4])))
9 :: (8 :: (7 :: []))
```

```
9 :: (8 :: [7])
9 :: [8,7]
[9,8,7]
```

the computation of `drop`

```
fun drop (0, xs)    = xs
  | drop (i, _::xs) = drop (i-1, xs);
```

```
drop (3, [9,8,7,6,5,4])
drop (2,   [8,7,6,5,4])
drop (1,     [7,6,5,4])
drop (0,       [6,5,4])
[6,5,4]
```

---

tail recursion

normal recursion

```
fun take(0, _)     = []
  | take(i, x::xs) = x::(take(i-1, xs));
```

tail recursion

```
fun drop (0, xs)    = xs
  | drop (i, _::xs) = drop (i-1, xs);
```

normal to tail recursive

```
fun length []      = 0
  | length (_::xs) = 1 + length xs;
```

use an **accumulator** to make it iterative

```
local
  fun ilen (n, [])    = n
    | ilen (n, _::xs) = ilen (n+1, xs)
in
  fun length xs = ilen (0, xs)
end;
```

---

builtin append operator

```
[x1,...,xm] @ [y1,...,yn] = [x1,...,xm,y1,...,yn]
```

```
infix @;
fun []      @ ys = ys
  | (x::xs) @ ys = x :: (xs @ ys);
```

```
["Append", "is"] @ ["never", "boring"];
```

- is it tail recursive?
- why can't it be used in patterns?

---

side note - `orelse` and `andalso`

they are short-circuiting boolean operators

```
B1 andalso B2 = if B1 then B2  else false;
```

```
B1 orelse  B2 = if B1 then true else B2;
```

```
fun even n = (n mod 2 = 0);

fun powoftwo n =
  (n=1) orelse
  (even n andalso powoftwo (n div 2));
```

is `powoftwo` tail-recursive?

---

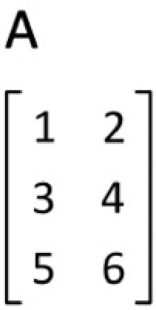
builtin function `map`

```
fun map f []      = []
  | map f (x::xs) = (f x) :: (map f xs);

val sqlist = map (fn x => x*x);

sqlist [1,2,3];
```

transposing a matrix using `map`



transp gif

```
fun transp ([]::_) = []
  | transp rows =
      (map hd rows) :: (transp (map tl rows));
```

---

builtin function `filter`

```
fun filter pred []      = []
  | filter pred (x::xs) =
       if pred x then (x:: filter pred xs)
                 else      filter pred xs;

filter (fn x => x mod 2 = 0) [1,2,3,4,5];
```

`filter` is bound as `List.filter`

---

using `map` and `filter`

a polynomial is represented as a list of `$(coeff,degree)$` pairs

$5x^3 + 2x + 7$

```
type polynomial = (int*int) list;
val a = [(5,3), (2,1), (7,0)]: polynomial;
```

taking the derivative of a polynomial

```
fun derive (p: polynomial): polynomial =
    List.filter
        (fn (coeff, deg) => deg >= 0)
        (map
            (fn (coeff, deg) => (coeff*deg, deg-1))
            p
        )
;

derive a;
```

`find`

```
fun find f [] = NONE
  | find f (x::xs) = if f x then SOME x else find f xs;
```

bound as `List.find`

`foldl` and `foldr`

builtin function `foldl`

```
fun foldl f init []      = init
  | foldl f init (x::xs) = foldl f (f (x, init)) xs;
```

calculates `$[x_1, x_2, … ,x_n] \rightarrow f(x_n, … ,f(x_2, f(x_1,init)))$`

builtin function `foldr`

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

calculates `$[x_1, x_2, … ,x_n] \rightarrow f(x1, … ,f(xn-1, f(xn,init)))$`

using `foldl` and `foldr`

let's redefine some functions…

```
fun sum l = foldl op+ 0 l;
```

```
fun reverse l = foldl op:: [] l;
```

```
fun xs @ ys = foldr op:: ys xs;
```

`exists` and `all`

builtin function `exists`

```
fun exists p []      = false
  | exists p (x::xs) = (p x) orelse exists p xs;
```

checks if the predicate `p` is satisfied by at least one element of the list

```
exists (fn x => x < 0) [1, 2, ~3, 4];
```

bound as `List.exists`

builtin function `all`

```
fun all p []      = true
  | all p (x::xs) = (p x) andalso all p xs;
```

checks if the predicate `p` is satisfied by **all** elements of the list

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

bound as `List.all`

```
fun disjoint (xs, ys) =
  all (fn x => all (fn y => x<>y) ys) xs;
```

---

equality in polymorphic functions

equality is polymorphic in a restricted sense

- defined for values constructed of integers, strings, booleans, chars, tuples, lists and datatypes
- not defined for values containing
  - functions: equality is undecidable (halting problem)
  - reals, because e.g. nan != nan
  - elements of abstract types

ML has a polymorphic equality type `''a`

```
op=;
```

somewhat like an interface/trait in other languages

---

exam questions

exercise 1

implement `map` using `foldl`

```
val foldl = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b;
val map = fn : ('a -> 'b) -> 'a list -> 'b list;

fun map f inpList = foldl
    _
    _
    inpList
;

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

exercise 2

`insSort` (insertion sort) sorts a list according to a given less-then function.

```
val foldr = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b;
val insSort : ('a * 'a -> bool) -> 'a list -> 'a list;

fun insSort lt inpList = foldr
    _
    _
    inpList
;

insSort (op<) [1, ~3, 5, 0];
```

exercise 3

```
fun upto m n = if (m > n)
    then []
    else m::(upto (m+1) n)
;

infix o;
fun f o g = fn x => f (g x);
```

what will be printed?

```
val a = map (upto 2) (upto 2 5);
```

what will be printed?

```
map
    (
        (fn f => null (f()))
        o
        (fn t => fn () => tl t)
    )
    a
;
```

what will be printed?

```
map
    (List.filter (fn t => t mod 2 = 0))
    a
;
```

exercise 4

implement a tail recursive `append`

reminder:

```
infix @;
fun []      @ ys = ys
  | (x::xs) @ ys = x :: (xs @ ys);

fun append ...
```

NOTE:

```
fun aux([], ys) = ys
  | aux(x::xs, ys) = aux (xs, x::ys);

fun append (xs, ys) = aux (aux (xs, []), ys);
```

exercise 5

implement `flatten` using `foldr`

```
flatten : 'a list list -> 'a list;

fun flatten ...

[1,2,3,4,5,6,7,8,9] = flatten [[1,2,3],[4,5,6],[],[7,8,9]];
```

NOTE:

```
fun flatten xs = foldr (op@) [] xs;
```