Standard ML

Lists
A list is a finite sequence of elements.
- [3, 5, 9]
- ["a", "list"]
- []

ML lists are immutable.

Elements may appear more than once, and the order matters:
- [3, 4, 3]
- [3, 3, 4]

Elements may have any type...

...But all elements of a list must have the same type:
- [(1, "One"), (2, "Two")]: (int*string) list
- [[3.1], [], [5.7, ~0.6]]: real list list

The empty list [] has the polymorphic type ‘a list
Building a List

◆ Every list is either
  ● Empty
  or
  ● An element followed by a list

◆ Examples:
  ● [1, 2, 3, 4] <=> Head = 1, Tail = [2,3,4]
  ● [1, 2] <=> Head = 1, Tail = [2]
  ● [1] <=> Head = 1, Tail = []
  ● [] <=> []

The tail is a list!
The empty list cannot be disassembled
Building a List (cont.)

- \texttt{nil} is a synonym for the empty list \texttt{[]}.
  - They are completely interchangeable.

- The operator \texttt{::} (also called \texttt{cons}) makes a new list by combining an element and an existing list.
  - \texttt{1 :: [2,3];}
  - \texttt{val it = [1,2,3] : int list}

- Every list is either \texttt{nil} or has the form \texttt{x::xs} where \texttt{x} is its head and \texttt{xs} is its tail (which is a list itself).

- The \texttt{infixr} operator \texttt{::} groups to the \texttt{right}.

- It is a \texttt{constructor}, which is a special kind of function.
  - Basically, it means it can be used in patterns.

- The notation \texttt{[x1,x2,...,xn]} stands for \texttt{x1 :: x2 :: ... :: xn :: nil}
  - \texttt{3::(5::(9::nil))} is \texttt{[3,5,9]}
• **Testing lists and taking them apart**

  • **null** - tests whether a list is empty. Could have been defined like that:
    
    ```
    fun null [] = true
    | null (_::_) = false;
    val null = fn : 'a list -> bool
    ```

  • **hd** – evaluates to the head of a non-empty list
    
    ```
    fun hd (x::_) = x;
    **Warning: Patterns not exhaustive**
    val hd = fn : 'a list -> 'a
    ```

  • **tl** - evaluates to the tail of a non-empty list
    
    ```
    fun tl (_::xs) = xs;
    **Warning: Patterns not exhaustive**
    val tl = fn : 'a list -> 'a list
    ```

Note how list constructors are used in patterns.
Building the list of integers \([m,m+1,...,n-1]\)

- The implementation:
  
  ```ml
  fun range (m, n) = if m=n then []
                  else m :: (range (m+1, n));
  val range = fn : int * int -> int list
  range (2, 5);
  val it = [2,3,4] : int list
  ``

- Or as an operator:
  
  ```ml
  infix --;
  val op-- = range;
  val -- = fn : int * int -> int list
  2 -- 5;
  val it = [2,3,4] : int list
  ```
**take and drop**

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

\[\text{take (k, xs)} \quad \text{drop (k, xs)}\]

- **take (k, xs)** contains the first \(k\) elements of \(xs\)
  ```ml
take = fn : 'a list * int -> 'a list
  fun take (0, _) = []
  | take (i, x::xs) = x :: take (i-1, xs);
  val take = fn : 'a list * int -> 'a list
  - take (5, explode "Throw Pascal to the dogs!");
  val it = [#"T", #"h", #"r", #"o", #"w"] : char list
  ```

- **drop (k, xs)** contains all but the first \(k\) elements of \(xs\)

*explosion*: converts a string to a list of chars
The Computation of \texttt{take}

\begin{verbatim}
fun take (0, _)   = []
  | take (i, x::xs) = x :: take (i-1, xs);

- take (3, [9,8,7,6,5,4])
  \implies 9::take (2, [8,7,6,5,4])
  \implies 9::(8::take (1, [7,6,5,4]))
  \implies 9::(8::(7::take (0, [6,5,4])))
  \implies 9::(8::(7::[]))
  \implies 9::(8::[7])
  \implies 9::[8,7]
  \implies [9,8,7]
\end{verbatim}
The Computation of \texttt{drop}

\begin{verbatim}
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]
\end{verbatim}
Tail Recursion

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

- **Tail recursion (also called an iterative function)**
  - fun drop (0, xs) = xs
  - | drop (i, _::xs) = drop (i-1, xs);

- In tail recursion there is no need to "go up" in the recursion, since there is no computation left to do.

- Tail recursion can be implemented more efficiently using loops
  - General tail-call elimination (TCO) : using GOTO.
Transforming Normal to Tail Recursion

◆ The built-in length function:

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

`val length = fn : 'a list -> int`

◆ In order to transform it into an iterative function, we will use a helper variable:

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

`val length = fn : 'a list -> int`

➢ In general, not necessarily a single variable.
➢ It is called “accumulator”, or “acc”.
The Built-in Append Operation

- Puts the elements of one list after those of another list
- \([x_1,\ldots,x_m] \@ [y_1,\ldots,y_n] = [x_1,\ldots,x_m,y_1,\ldots,y_n]\)
  - infix \(@\);
  - fun [] \@ ys = ys
    | (x::xs) \@ ys = x::(xs@ys);
  
  val \@ = fn : 'a list * 'a list -> 'a list

- Examples
  - ["Append","is"] \@ ["never","boring"];
  - [[2,4,6,8],[3,9]] \@ [[5],[7]];

- Is it a tail recursion?
- Why can’t it be used in patterns?
Side Note: `orelse` and `andalso`

- They are **short-circuit** OR and AND boolean operations.

\[
B_1 \text{ andalso } B_2 \iff \text{if } B_1 \text{ then } B_2 \text{ else false}
\]

\[
B_1 \text{ orelse } B_2 \iff \text{if } B_1 \text{ then true else } B_2
\]

- `B_2` is evaluated only if needed.

- **Is the following** `powoftwo` **function a tail recursion?**
  - `fun even n = (n mod 2 = 0);`
  - `val even = fn : int -> bool`

  - `fun powoftwo n = (n=1) orelse`  
    `\text{(even n andalso powoftwo (n div 2))};`
  - `val powoftwo = fn : int -> bool`
**map**

- Applying a function to all the elements in a list

\[ \text{map} \ f \ [x_1, \ldots, x_n] = [f \ x_1, \ldots, f \ x_n] \]

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

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

- Usage:
  - val sqlist = map (fn x => x*x);
    val sqlist = fn : int list -> int list
  - sqlist [1,2,3];
    val it = [1,4,9] : int list

- Transposing a matrix using map
  - fun transp ([]:_ = []
    | transp rows =
      (map hd rows) :: transp (map tl rows);

val transp = fn: 'a list list -> 'a list list

*Note: a curried function*
filter

- **filter** returns all elements satisfying a predicate
  ```
  fun filter pred [] = []
  | filter pred (x::xs) =
    if pred x then (x::filter pred xs)
    else filter pred xs;
  ```
  
  val filter = fn : ('a -> bool) -> 'a list -> 'a list

- **Example**
  ```
  filter (fn x => x mod 2 = 0) [1,2,3,4,5];
  ```
  
  val it = [2,4] : int list

*filter is built-in but bound as List.filter (this is also the case for some of the other functions in this slides)*
Using map and filter

- Polynomial is represented as a list of \((\text{coeff}, \text{degree})\) pairs
  - \(5x^3 + 2x + 7\) is represented by
  
  ```
  val a = [(5,3),(2,1),(7,0)];
  ```

- Taking the derivative - we need to take each pair \((a, n)\) and convert it to the pair \((a \cdot n, n - 1)\). Then we need to remove elements with negative rank or zero coeff.

  ```
  fun derive p = 
    List.filter (fn (a,n) => n>=0) 
    (map (fn (a,n) => (a*n, n-1)) p);
  ```

  ```
  val derive = fn : (int * int) list -> (int * int) list
  - derive a;
  ```

  ```
  val it = [(15,2),(2,0)] : (int * int) list
  ```
foldl and foldr

- **foldl** $f$ $init$ $[x_1, x_2, \ldots, x_n]$ calculates $f(x_n, \ldots, f(x_2, f(x_1, init)))$

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

- **foldr** $f$ $init$ $[x_1, x_2, \ldots, x_n]$ calculates $f(x_1, \ldots, f(x_{n-1}, f(x_n, init)))$

  **Possible implementation:**
  ```ml
  fun foldr f init [] = init
  | foldr f init (x::xs) = f (x, foldr f init xs);
  
  val foldr = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
  ```
Using foldl and foldr

◆ Summarize list of integers
  - fun sum list = foldl op+ 0 list;
  
  val sum = fn : int list -> int

◆ Reverse a list (alternative for List.rev)
  - fun reverse list = foldl op:: [] list;

  val sum = fn : 'a list -> 'a list

◆ Redefine the append operator (@)
  - fun xs @ ys = foldr op:: ys xs;

  val @ = fn : 'a list * 'a list -> 'a list
exists and all

- **Inside List:** List.exists, List.all
- Checks if pred is satisfied for an element or the whole list
  - fun exists p [] = false
  - | exists p (x::xs) = (p x) orelse exists p xs;
  - val exists = fn:('a -> bool)-> 'a list -> bool
  - fun all p [] = true
  - | all p (x::xs) = (p x) andalso all p xs;
  - val forall = fn:('a -> bool) -> 'a list -> bool

- Useful for converting a predicate over type 'a to a predicate over type 'a list
  - fun disjoint (xs, ys) = all (fn x => all (fn y => x<>y) ys) xs;
  - val disjoint = fn : "a list * "a list -> bool
Equality Test 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
- Standard ML has equality type variables ranging over the equality types
  - `op= ;
    val it = fn : ("a * "a) -> bool
- Somewhat similar to an interface in other languages
A list of functions is a perfectly legitimate value

- `[fn x => 2*x, fn x => 3*x];
val it = [fn,fn] : (int -> int) list
- map (fn f => f 3) it;
val it = [6,9] : int list

Example from exam

- fun upto m n = if m>n then []
  else m:(upto (m+1) n);
- map upto (upto 1 4);
val it = [fn,fn,fn,fn] : (int -> int list) list
- map (fn f => f 4) it;
val it = [[1,2,3,4],[2,3,4],[3,4],[4]] : int list list
שאלה מבחן (1)

בכל אוחז מתוחה תחביר יסודי בראון על יציר光纤ית א' באנacağı א' באנضرورة פונקציה ב'.

פונקציה א': פונקציה ב': foldl

fun map f inpList = foldl __________________________________________________________
                                                                                           inpList;

fun map f inpList = foldl (fn (x,a) => a@[f x]) [] inpList;

foldr: פונקציה א': insSort (פונ',crc מפשים מינווי תכשitta לרשימה). פונקציה ב':

הנפטרת - הפונקציה מלבשת פונקציה מינווי רשימה, ומיומנות אוחזת על ידי כל שוהים מוכרים של איבר

לרשימה بمוקס 함ונד ול.


fun insSort f inpList = foldr __________________________________________________________
                                                                                           inpList;

fun insAux f (x, []) = [x]

| insAux f (x, y::ys) = if f(x,y) then x::y::ys else y::(insAux f (x, ys));
| - fun insSort f list = foldr (insAux f) [] list;
שאלה ממבחן

מה يوجد במקראים הבאים, אכזרים הם הוכנסו אחד אחריוحسن למשרשי:

במידה ויש ש鹣ואה, יש לאז אוז הוסביר את סיבת השיאוה. אין צורלכסה את תגובת המשרשי במידה.

בになりました הבאים נשתתמו בפוקציותו התש.

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

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

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

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

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

val it = [[2],[2],[2,4],[2,4]] : int list list
## ML’s List-Library Functions seen

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>null L</td>
<td>'a list -&gt; bool</td>
<td>Checks whether L is empty</td>
</tr>
<tr>
<td>hd L</td>
<td>'a list -&gt; 'a</td>
<td>Returns the head of L</td>
</tr>
<tr>
<td>tl L</td>
<td>'a list -&gt; 'a list</td>
<td>Returns the tail of L</td>
</tr>
<tr>
<td>take L n</td>
<td>'a list * int -&gt; 'a list</td>
<td>Returns the first n elements of L</td>
</tr>
<tr>
<td>drop L n</td>
<td>'a list * int -&gt; 'a list</td>
<td>Returns L without its first n elements</td>
</tr>
<tr>
<td>length L</td>
<td>'a list -&gt; int</td>
<td>Returns the length of L</td>
</tr>
<tr>
<td>L @ M</td>
<td>'a list * 'a list -&gt; 'a list</td>
<td>Returns a “glue” of L and M</td>
</tr>
<tr>
<td>map f L</td>
<td>('a -&gt; 'b) -&gt; ('a list -&gt; 'b list)</td>
<td>Returns L, when each element is mapped by f</td>
</tr>
<tr>
<td>filter pred L</td>
<td>('a -&gt; bool) -&gt; ('a list -&gt; 'a list)</td>
<td>Returns the elements of L which satisfy the predicate pred</td>
</tr>
<tr>
<td>exists pred L</td>
<td>('a -&gt; bool) -&gt; ('a list -&gt; bool)</td>
<td>Checks if any of the elements of L satisfies the predicate pred</td>
</tr>
<tr>
<td>all pred L*</td>
<td>('a -&gt; bool) -&gt; ('a list -&gt; bool)</td>
<td>Checks if all of the elements of L satisfies the predicate pred</td>
</tr>
</tbody>
</table>