

Introduction to SML

Lists

Michael R. Hansen

mrh@imm.dtu.dk

Informatics and Mathematical Modelling
Technical University of Denmark

Overview

- values and constructors
- recursions following the structure of lists
- useful built-in functions
- polymorphic types, values and functions

Lists

A list is a finite sequence of elements having the same type:

$[v_1, \dots, v_n]$ ($[]$ is called the empty list)

Lists

A list is a finite sequence of elements having the same type:

$[v_1, \dots, v_n]$ ($[]$ is called the empty list)

- `[2, 3, 6];`

> `val it = [2, 3, 6] : int list`

Lists

A list is a finite sequence of elements having the same type:

$[v_1, \dots, v_n]$ ($[]$ is called the empty list)

```
- ["a", "ab", "abc", ""];  
> val it = ["a", "ab", "abc", ""] : string list
```

Lists

A list is a finite sequence of elements having the same type:

$[v_1, \dots, v_n]$ ($[]$ is called the empty list)

```
- [Math.sin, Math.cos];  
> val it = [fn, fn] : (real -> real) list
```

Lists

A list is a finite sequence of elements having the same type:

$[v_1, \dots, v_n]$ (`[]` is called the empty list)

```
- [(1, true), (3, true)];  
> val it = [(1, true), (3, true)]: (int*bool) list
```

Lists

A list is a finite sequence of elements having the same type:

$[v_1, \dots, v_n]$ (`[]` is called the empty list)

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

The type constructor: `list`

If τ is a type, so is $\tau \text{ list}$

Examples:

The type constructor: `list`

If τ is a type, so is τ `list`

Examples:

- `int list`

The type constructor: `list`

If τ is a type, so is τ `list`

Examples:

- `int list`
- `(string * int) list`

The type constructor: `list`

If τ is a type, so is τ `list`

Examples:

- `int list`
- `(string * int) list`
- `((int -> string) list ) list`

The type constructor: `list`

If τ is a type, so is τ `list`

Examples:

- `int list`
- `(string * int) list`
- `((int -> string) list ) list`

`list` has higher precedence than `*` and `->`

`int * real list -> bool list`

The type constructor: `list`

If τ is a type, so is τ `list`

Examples:

- `int list`
- `(string * int) list`
- `((int -> string) list ) list`

`list` has higher precedence than `*` and `->`

`int * real list -> bool list`

means

`(int * (real list)) -> (bool list)`

Trees for lists

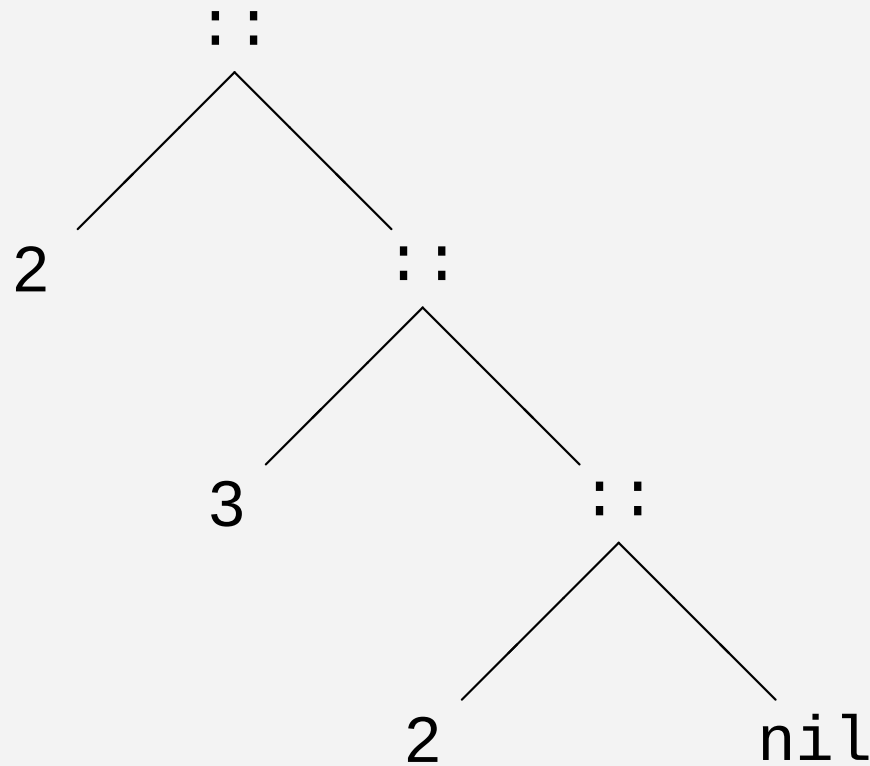
A non-empty list $[x_1, x_2, \dots, x_n]$, $n \geq 1$, consists of

- a *head* x_1 and
- a *tail* $[x_2, \dots, x_n]$

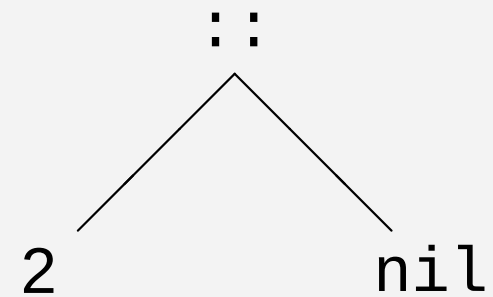
Trees for lists

A non-empty list $[x_1, x_2, \dots, x_n]$, $n \geq 1$, consists of

- a *head* x_1 and
- a *tail* $[x_2, \dots, x_n]$



Graph for $[2, 3, 2]$



Graph for $[2]$

List constructors: [], nil and ::

Lists are generated as follows:

- the empty list is a list, designated [] or nil
- if x is an element and xs is a list,
then so is $x :: xs$

(type consistency)

:: associate to the **right**, i.e. $x_1 :: x_2 :: xs$

List constructors: [], nil and ::

Lists are generated as follows:

- the empty list is a list, designated [] or nil
- if x is an element and xs is a list,
then so is $x :: xs$ (type consistency)

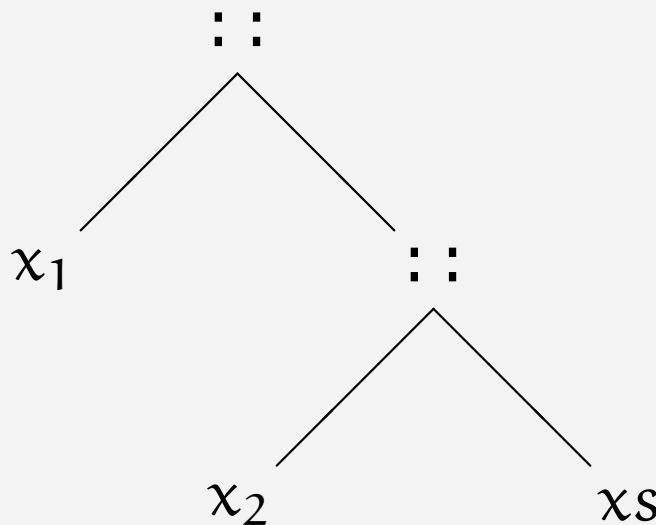
$::$ associate to the **right**, i.e. $x_1 :: x_2 :: xs$ means $x_1 :: (x_2 :: xs)$

List constructors: [], nil and ::

Lists are generated as follows:

- the empty list is a list, designated [] or nil
- if x is an element and xs is a list, then so is $x :: xs$ (type consistency)

$::$ associate to the **right**, i.e. $x_1 :: x_2 :: xs$ means $x_1 :: (x_2 :: xs)$



Graph for $x_1 :: x_2 :: xs$

Recursion on lists – a simple example

$$\text{suml } [x_1, x_2, \dots, x_n] = \sum_{i=1}^n x_i = x_1 + x_2 + \dots + x_n = x_1 + \sum_{i=2}^n x_i$$

Constructors are used in list patterns

```
fun suml [] = 0
  | suml (x::xs) = x + suml xs
> val suml = fn : int list -> int
```

```
suml [1,2]
```

```
~> 1 + suml [2]           (x ↦ 1 and xs ↦ [2])
```

```
~> 1 + (2 + suml [])      (x ↦ 2 and xs ↦ [])
```

```
~> 1 + (2 + 0)           (the pattern [] matches the value [])
```

```
~> 1 + 2
```

```
~> 3
```

Recursion follows the structure of lists

Append

The infix operator `@` (called ‘append’) joins two lists:

$$\begin{aligned} [x_1, x_2, \dots, x_m] @ [y_1, y_2, \dots, y_n] \\ = [x_1, x_2, \dots, x_m, y_1, y_2, \dots, y_n] \end{aligned}$$

Properties

$$\begin{aligned} [] @ ys &= ys \\ [x_1, x_2, \dots, x_m] @ ys &= x_1 :: ([x_2, \dots, x_m] @ ys) \end{aligned}$$

Declaration

```
infixr 5 @                               (* right associative *)
fun [] @ ys = ys
  | (x::xs) @ ys = x::(xs @ ys);
```

Append: evaluation

```
infixr 5 @                (* right associative *)
fun      [] @ ys = ys
  | (x::xs) @ ys = x::(xs @ ys);
```

Evaluation

$[1, 2] @ [3, 4]$	
$\rightsquigarrow 1 :: ([2] @ [3, 4])$	$(x \mapsto 1, xs \mapsto [2], ys \mapsto [3, 4])$
$\rightsquigarrow 1 :: (2 :: ([] @ [3, 4]))$	$(x \mapsto 2, xs \mapsto [], ys \mapsto [3, 4])$
$\rightsquigarrow 1 :: (2 :: [3, 4])$	$(ys \mapsto [3, 4])$
$\rightsquigarrow 1 :: [2, 3, 4]$	
$\rightsquigarrow [1, 2, 3, 4]$	

Append: polymorphic type

```
> infixr 5 @  
> val @ = fn : 'a list * 'a list -> 'a list
```

- 'a is a *type variable*
- The type of @ is *polymorphic* — it has many forms

'a = int: Appending integer lists

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

'a = int list: Appending lists of integer list

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

@ is a built-in function

Reverse

$$\text{rev } [x_1, x_2, \dots, x_n] = [x_n, \dots, x_2, x_1]$$

```
fun naive_rev []      = []  
  | naive_rev(x::xs) = naive_rev xs @ [x];  
val naive_rev = fn : 'a list -> 'a list
```

```
    naive_rev[1,2,3]  
  ~> naive_rev[2,3] @ [1]  
  ~> (naive_rev[3] @ [2]) @ [1]  
  ~> ((naive_rev[] @ [3]) @ [2]) @ [1]  
  ~> (([] @ [3]) @ [2]) @ [1]  
  ~> ([3] @ [2]) @ [1]  
  ~> (3::([] @ [2])) @ [1]  
  ~> ...  
  ~> [3,2,1]
```

efficient version is built-in (see Ch. 17)

Membership — equality types

$$\begin{aligned} & x \text{ member } [y_1, y_2, \dots, y_n] \\ &= (x = y_1) \vee (x = y_2) \vee \dots \vee (x = y_n) \\ &= (x = y_1) \vee (x \text{ member } [y_2, \dots, y_n]) \end{aligned}$$

Declaration

`infix member`

```
fun x member []           = false
    | x member (y::ys) = x=y orelse x member ys;
infix 0 member
val member = fn : ''a * ''a list -> bool
```

- `''a` is an equality type variable no functions
- `(1,true) member [(2,true), (1,false)]` $\rightsquigarrow$ false
- `[1,2,3] member [[1], [], [1,2,3]]` $\rightsquigarrow$?

Value polymorphism

- e is a *value expression* if no further evaluation is needed

```
- (5, []);                                (* val it = (5, []); *)  
> val 'a it = (5, []) : int * 'a list
```

```
- rev [];                                (* non-value expression *)
```

! Warning: Value polymorphism:

! Free type variable(s) at top level in value id. it

- A type is *monomorphic* if it contains no type variables, otherwise it is *polymorphic*

SML restricts the use of polymorphic types as follows: [see Ch. 18](#)

- all monomorphic expressions are OK
- all value expressions are OK
- at top-level, polymorphic non-value expressions are forbidden

Examples

- $\text{remove}(x, ys)$: removes all occurrences of x in the list ys
- $\text{prefix}(xs, ys)$: the list xs is a prefix of the list ys (ex. 5.10)
- $\text{sum}(p, xs)$: the sum of all elements in xs satisfying the predicate $p: \text{int} \rightarrow \text{bool}$ (ex. 5.15)
- From list of pairs to pair of lists:

$$\begin{aligned} \text{unzip } [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)] \\ = ([x_1, x_2, \dots, x_n], [y_1, y_2, \dots, y_n]) \end{aligned}$$

Many functions on lists are predefined, e.g. `@`, `rev`, `length`, and also the SML basis library contains functions on lists, e.g. `unzip`. See for example `List`, `ListPair`