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

The purpose of this lecture is to give you an (as short as possible) introduction to lists, so that you can solve a problem which can illustrate some of SML's high-level features.

This part is *not* intended as a comprehensive presentation on lists, and we will return to the topic again later.

Lists

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

$$[v_1, \ldots, v_n] \qquad ([\,] \text{ is called the empty list})$$

Lists

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

$$[v_1, \ldots, v_n] \qquad ([\,] \text{ is called the empty list})$$

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

Lists

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

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

```
- [2,3,6];
> val it = [2, 3, 6] : int list
- ["a", "ab", "abc", ""];
> val it = ["a", "ab", "abc", ""] : string list
```

Lists

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

$$[v_1, \ldots, v_n] \qquad ([\,] \text{ is called the empty list})$$

```
- [2,3,6];
> val it = [2, 3, 6] : int list
- ["a", "ab", "abc", ""];
> val it = ["a", "ab", "abc", ""] : string 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:

$$[v_1, \ldots, v_n] \qquad ([\,] \text{ is called the empty list})$$

```
- [2,3,6];
> val it = [2, 3, 6] : int list
- ["a", "ab", "abc", ""];
> val it = ["a", "ab", "abc", ""] : string list
- [Math.sin, Math.cos];
> val it = [fn, fn] : (real -> real) 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:

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

```
- [2,3,6];
> val it = [2, 3, 6] : int list
- ["a", "ab", "abc", ""];
> val it = ["a", "ab", "abc", ""] : string list
- [Math.sin, Math.cos];
> val it = [fn, fn] : (real -> real) list
- [(1,true), (3,true)];
> val it = [(1, true),(3, true)]: (int*bool) list
- [[],[1],[1,2]];
> val it = [[], [1], [1, 2]] : int list list
```

The type constructor: `list`

If τ is a type, so is τ `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

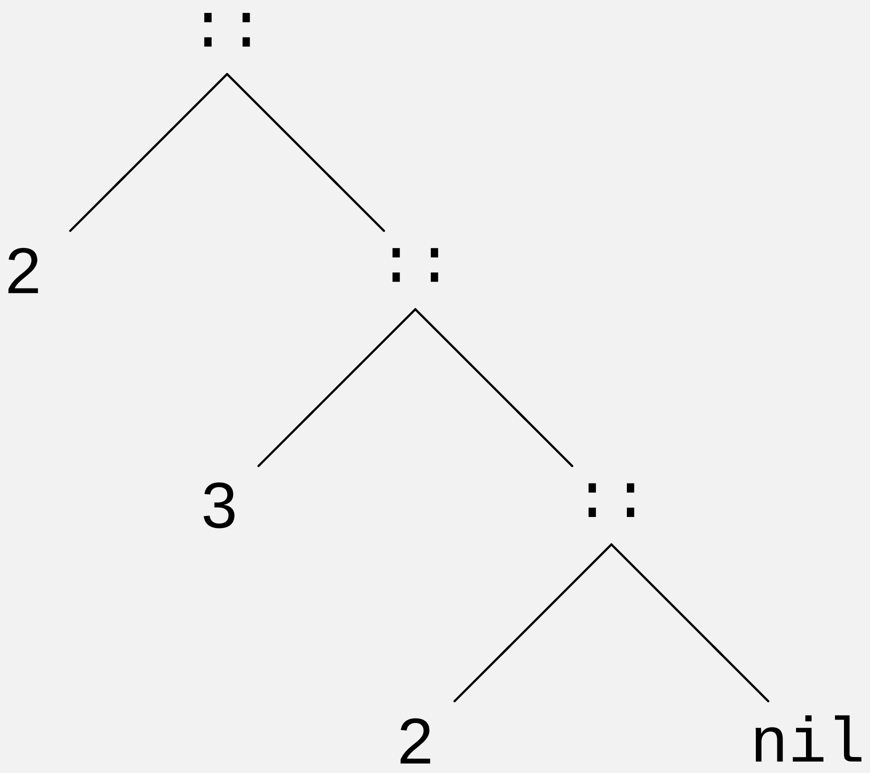
A non-empty list $[x_1, x_2, \ldots, x_n]$, $n \geq 1$, consists of

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

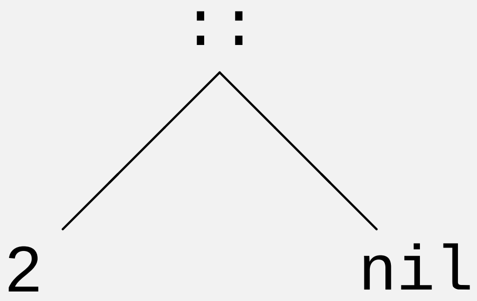
Trees for lists

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

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



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)

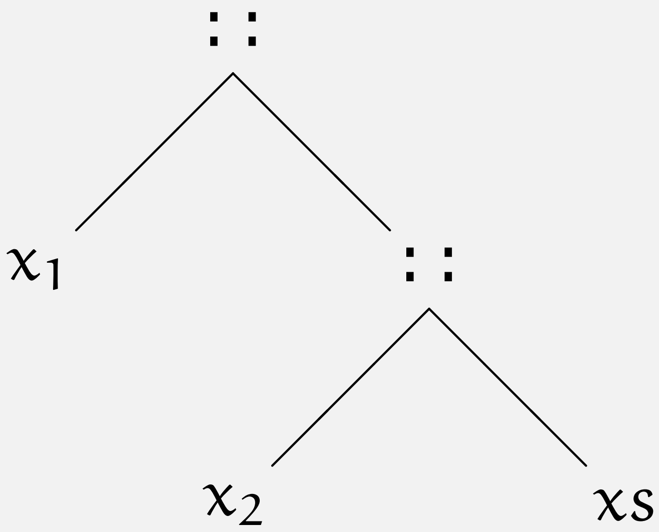
`::` associate to the right, i.e. $x_1{::}x_2{::}xs$ means $x_1{::}(x_2{::}xs)$

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

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

Constructors are used in list patterns

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

 `suml [1,2]`

$\rightsquigarrow$ `1 + suml [2]` $\quad$ ($\texttt{x} \mapsto 1$ and $\texttt{xs} \mapsto [2]$)

$\rightsquigarrow$ `1 + (2 + suml [])` $\quad$ ($\texttt{x} \mapsto 2$ and $\texttt{xs} \mapsto [\,]$)

$\rightsquigarrow$ `1 + (2 + 0)` $\quad$ (the pattern `[]` matches the value `[]`)

$\rightsquigarrow$ `1 + 2`

$\rightsquigarrow$ `3`

Recursion follows the structure of lists

Infix functions

It is possible to declare infix functions in SML, i.e. the function symbol is between the arguments.

The prefix function on lists, e.g. $[1,2,3]{<<==}[1,2,3,4] = \mathsf{true}$, is declared as follows:

```
infix 3 <<==

fun       [] <<== ys       = true
  |       xs <<== []       = false
  | (x::xs) <<== (y::ys) = x=y andalso xs <<== ys;
```

- the `infix` directive allows the function symbol to occur between the arguments.
- 3 is in this case the precedence of the symbol

Examples

- `remove`(x, ys) : removes all occurrences of x in the list *ys*
- `length` xs : the length of the list *xs* (is a predefined function).