Introduction to SML

Basic Types, Tuples, Lists, Trees and Higher-Order Functions

Michael R. Hansen

`mrh@imm.dtu.dk`

Informatics and Mathematical Modelling

Technical University of Denmark

Basic Types: Integers

A data type comprises

- a set of values and
- a collection of operations

Basic Types: Integers

A data type comprises

- a set of values and
- a collection of operations

Integers

Type name : `int`

Values : `~27`, `0`, `1024`

Operations: (A few selected)

Operator	Type	Precedence	Association
`~`	`int -> int`	Highest	
`* div mod`	`int * int -> int`	7	Left
`+ -`	`int * int -> int`	6	Left
`= <> < <=`	`int * int -> bool`	4	Left

See also the library `Int`

Reals

Type name : `real`

Values : `~27.0`, `0.0`, `1024.71717`, `23.4E~11`

Operations: (A few selected)

Operator	Type	Precedence	Association
`abs`	`real -> real`	Highest	
`* /`	`real*real -> real`	7	Left
`+ -`	`real*real -> real`	6	Left
`= <> < <=`	`real*real -> bool`	4	Left

See also the libraries `Real` and `Math`

Reals

Type name : `real`

Values : `~27.0`, `0.0`, `1024.71717`, `23.4E~11`

Operations: (A few selected)

Operator	Type	Precedence	Association
`abs`	`real -> real`	Highest	
`* /`	`real*real -> real`	7	Left
`+ -`	`real*real -> real`	6	Left
`= <> < <=`	`real*real -> bool`	4	Left

See also the libraries `Real` and `Math`

Some built-in operators are *overloaded*. `*`:

```
real*real -> real
int * int -> int
```

Default is `int`

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
`fun square x = x * x`	`int -> int`	Default

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
`fun square x = x * x`	`int -> int`	Default

A squaring function on reals: `square:  real -> real`

Declaration	

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
`fun square x = x * x`	`int -> int`	Default

A squaring function on reals: `square:  real -> real`

Declaration	
`fun square(x:real) = x * x`	Type the argument

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
`fun square x = x * x`	`int -> int`	Default

A squaring function on reals: `square: real -> real`

Declaration	
`fun square x:real = x * x`	Type the result

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
`fun square x = x * x`	`int -> int`	Default

A squaring function on reals: `square: real -> real`

Declaration	
`fun square x = x * x: real`	Type expression for the result

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
`fun square x = x * x`	`int -> int`	Default

A squaring function on reals: `square: real -> real`

Declaration	
`fun square x = x:real * x`	Type a variable

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
`fun square x = x * x`	`int -> int`	Default

A squaring function on reals: `square:  real -> real`

Declaration	
`fun square x = x:real * x`	Type a variable

Choose any mixture of these possibilities

Characters

Type name `char`

Values `#"a"`, `#" "`, `#"\""` (escape sequence for `"`)

Operator	Type	
`ord`	`char -> int`	ascii code of character
`chr`	`int -> char`	character for ascii code
`= < <= ...`	`char*char -> bool`	comparisons by ascii codes

Examples

```
- ord #"a";
> val it = 97 : int

- ord #"A";
> val it = 65 : int
```

```
-  #"a" < #"A";
> val it = false : bool;

- chr 88;
> val it = #"X" : char
```

Strings

Type name `string`

Values `"abcd"`, `" "`, `""`, `"123\"321"` (escape sequence for `"`)

Operator	Type	
`size`	`string -> int`	length of string
`^`	`string*string -> string`	concatenation
`= < <= ...`	`string*string -> bool`	comparisons
`Int.toString`	`int -> string`	conversions

Examples

```
- "auto" < "car";
> val it = true : bool

- "abc"^"de";
> val it = "abcde": string
```

```
- size("abc"^"def");
> val it = 6 : int

- Int.toString(6+18);
> val it = "24" : strin
```

Booleans

Type name `bool`

Values `false`, `true`

Operator	Type	
`not`	`bool -> bool`	negation

```
not true = false
not false = true
```

Expressions

e_1 `andalso` e_2 "conjunction $e_1 \wedge e_2$"

e_1 `orelse` e_2 "disjunction $e_1 \vee e_2$"

— are lazily evaluated, e.g.

```
1<2 orelse 5/0 = 1
⇝ true
```

Precedence: `andalse` has higher than `orelse`

Tuples

An ordered collection of n values $(v_1, v_2, \ldots, v_n)$ is called an n-tuple

Examples

`- ();` `> val it = () : unit`	0-tuple
`- (3, false);` `> val it = (3, false) : int * bool`	2-tuples (pairs)
`- (1, 2, ("ab",true));` `> val it = (1, 2, ("ab", true)) : ?`	3-tuples (triples)

Selection Operation: $\#i(v_1, v_2, \ldots, v_n) = v_i$. `#2(1,2,3) = 2`

Equality defined componentwise

```
- (1, 2.0, true) = (2-1, 2.0*1.0, 1<2);
> val it = true : bool
```

provided = is defined on components

Tuple patterns

Extract components of tuples

```
- val ((x,_),(_,y,_)) = ((1,true),("a","b",false));
> val x = 1 : int
  val y = "b" : string
```

Pattern matching yields bindings

Restriction

```
- val (x,x) = (1,1);
! Toplevel input:
! val (x,x) = (1,1);
!         ^
! identifier is bound twice in a pattern
```

Infix functions

Directives: `infix` d f and `infixr` d f. d is the precedence of f

Example: exclusive-or

```
infix 0 xor        (* or just infix xor
                      -- lowest precedence  *)

fun false xor true  = true
  | true  xor false = true
  | _     xor _     = false
```

type ?

```
- 1 < 2+3 xor 2.0 / 3.0 > 1.0;
> val it = true : bool
```

Infix status can be removed by `nonfix xor`

```
- xor(1 < 2+3, 2.0 / 3.0 > 1.0);
> val it = true : bool
```

Let expressions — `let` *dec* `in` *e* `end`

Bindings obtained from *dec* are valid only in e

Example: Solve $ax^2 + bx + c = 0$

```
type equation = real * real * real
type solution = real * real

exception Solve;  (* declares an exception *)

fun solve(a,b,c) =
   let val d = b*b-4.0*a*c
   in if d < 0.0 orelse a = 0.0 then raise Solve
      else ((~b+Math.sqrt d)/(2.0*a)
           ,(~b-Math.sqrt d)/(2.0*a))
   end;
```

The type of `solve` is `equation -> solution`

d is declared once and used 3 times readability, efficiency

Local declarations — `local` dec_2 `in` dec_2 `end`

Bindings obtained from dec_1 are valid only in dec_2

```
local
   fun disc(a,b,c) = b*b - 4.0*a*c
in
   exception Solve;

   fun hasTwoSolutions(a,b,c) = disc(a,b,c)>0.0
                                andalso a<>0.0;

   fun solve(a,b,c) =
      let val d = disc(a,b,c)
      in if d < 0.0 orelse a = 0.0 then raise Solve
         else ((~b+Math.sqrt d)/(2.0*a)
              ,(~b-Math.sqrt d)/(2.0*a))
      end
end;
```

Lists: 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:

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

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)

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

Lists

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

$[v_1, \ldots, v_n]$ ($[\,]$ 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:

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

```
- [(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:

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

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

The type constructor: `list`

If τ is a type, so is 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

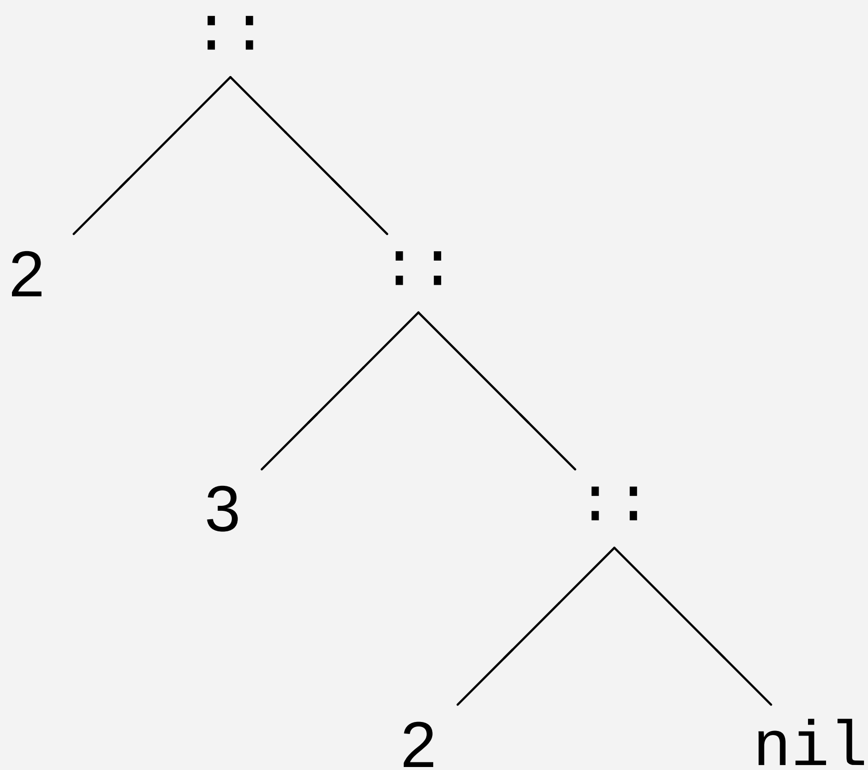
A non-empty list $[x_1, x_2, \ldots, x_n]$, $n \geq 1$, consists of

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

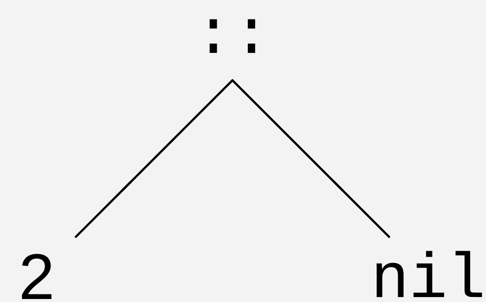
Trees for lists

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)

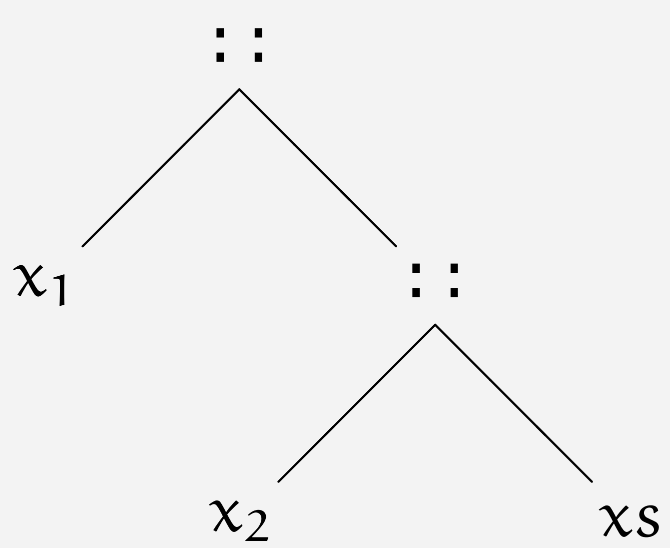
`::` associate to the right, i.e. $x_1{::}x_2{::}xs$ means $x_1{::}(x_2{::}xs)$

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]
⇝  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:

$$[x_1, x_2, \ldots, x_m] \mathbin{@} [y_1, y_2, \ldots, y_n]$$
$$= [x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n]$$

Properties

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

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

$$
\begin{array}{lll}
 & \texttt{[1,2] @ [3,4]} & \\
\rightsquigarrow & \texttt{1::([2] @ [3,4])} & (\texttt{x} \mapsto 1,\ \texttt{xs} \mapsto [2],\ \texttt{ys} \mapsto [3,4]) \\
\rightsquigarrow & \texttt{1::(2::([] @ [3,4]))} & (\texttt{x} \mapsto 2,\ \texttt{xs} \mapsto [\,],\ \texttt{ys} \mapsto [3,4]) \\
\rightsquigarrow & \texttt{1::(2::[3,4])} & (\texttt{ys} \mapsto [3,4]) \\
\rightsquigarrow & \texttt{1::[2,3,4]} & \\
\rightsquigarrow & \texttt{[1,2,3,4]} &
\end{array}
$$

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 $\mathtt{rev}\ [x_1, x_2, \ldots, x_n] = [x_n, \ldots, x_2, x_1]$

```
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

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

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* is it contains no type variables, otherwise it is *polymorphic*

SML resticts 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

- `remove`(x, ys) : removes all occurrences of x in the list *ys*
- `prefix`(xs, ys) : the list *xs* is a prefix of the list ys (ex. 5.10)
- `sum`(p, xs) : the sum of all elements in xs satisfying the predicate p: `int -> bool` (ex. 5.15)
- From list of pairs to pair of lists:

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

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`

Overview

- Disjoint Sets
  - The `datatype` – simple version
  - case expressions

Disjoint Sets: An Example

A *shape* is either a circle, a square, or a triangle

- the union of three disjoint sets

A *datatype* declaration for shapes:

```
datatype shape = Circle of real
               | Square of real
               | Triangle of real*real*real;
```

Answer from the SML system:

```
> datatype  shape
>   con Circle = fn : real -> shape
>   con Square = fn : real -> shape
>   con Triangle = fn : real * real * real -> shape
```

Constructors of a datatype

The *tags* `Circle`, `Square` and `Triangle` are *constructors* of values of type `shape`

```
- Circle 2.0;
> val it = Circle 2.0 : shape

- Triangle(1.0, 2.0, 3.0);
> val it = Triangle(1.0, 2.0, 3.0) : shape

- Square 4.0;
> val it = Square 4.0 : shape
```

Equality on `shape`s is defined provided ...

```
- Triangle(1.0, 2.0, 3.0) = Square 2.0;
> val it = false :  bool
```

Constructors in Patterns

```
fun area(Circle r)            = Math.pi * r * r
  | area(Square a)            = a * a
  | area(Triangle(a,b,c)) =
             let val d = (a + b + c)/2.0
             in Math.sqrt(d*(d-a)*(d-b)*(d-c))
             end;
> val area = fn : shape -> real
```

- a constructor only matches itself

$$
\begin{array}{ll}
 & \texttt{area (Circle 1.2)} \\
\rightsquigarrow & (\texttt{Math.pi * r * r},\ [\texttt{r} \mapsto \texttt{1.2}]) \\
\rightsquigarrow & \ldots
\end{array}
$$

The `case`-expression

Form:

$$\texttt{case}\ \mathit{exp}\ \texttt{of}$$
$$\mathit{pat}_1\ \texttt{=>}\ e_1$$
$$|\ \mathit{pat}_2\ \texttt{=>}\ e_2$$
$$\ldots$$
$$|\ \mathit{pat}_k\ \texttt{=>}\ e_k$$

Example:

```
fun area s =
   case s of
         (Circle r)              => Math.pi * r * r
       | (Square a)              => a*a
       | (Triangle(a,b,c)) =>
              let val d = (a + b + c)/2.0
              in Math.sqrt(d*(d-a)*(d-b)*(d-c))
              end;
```

Enumeration types – the `order` type

```
datatype order = LESS | EQUAL | GREATER;
```

Predefined ‘compare’ functions, e.g.

$$\texttt{Int.compare}(x, y) = \begin{cases} \texttt{LESS} & \text{if } x < y \\ \texttt{EQUAL} & \text{if } x = y \\ \texttt{GREATER} & \text{if } x > y \end{cases}$$

Example:

```
fun countLEG []          = (0,0,0)
 |  countLEG(x::rest) =
       let val (y1,y2,y3) = countLEG rest in
          case Int.compare(x,0) of
              LESS    => (y1+1,y2  ,y3  )
            | EQUAL   => (y1  ,y2+1,y3  )
            | GREATER => (y1  ,y2  ,y3+1)
       end;
```

The `option` type

```
datatype 'a option = NONE | SOME of 'a;
```

Example

```
fun smallest []        = NONE
  | smallest(x::xs) =
      case smallest xs of
          NONE => SOME x
        | SOME y => if x< y then SOME x else SOME y;

> val smallest = fn : int list -> int option

- smallest [2, ~3, 6];
> val it = SOME ~3 : int option
```

smallest — continued

The predefined function `valOf`:

```
exception Option;

fun valOf(SOME x) = x
  | valOf NONE    = raise Option;

> val 'a valOf = fn : 'a option -> 'a

-  3 + valOf(smallest [1,2,9]);
> val it = 4 : int
```

Overview

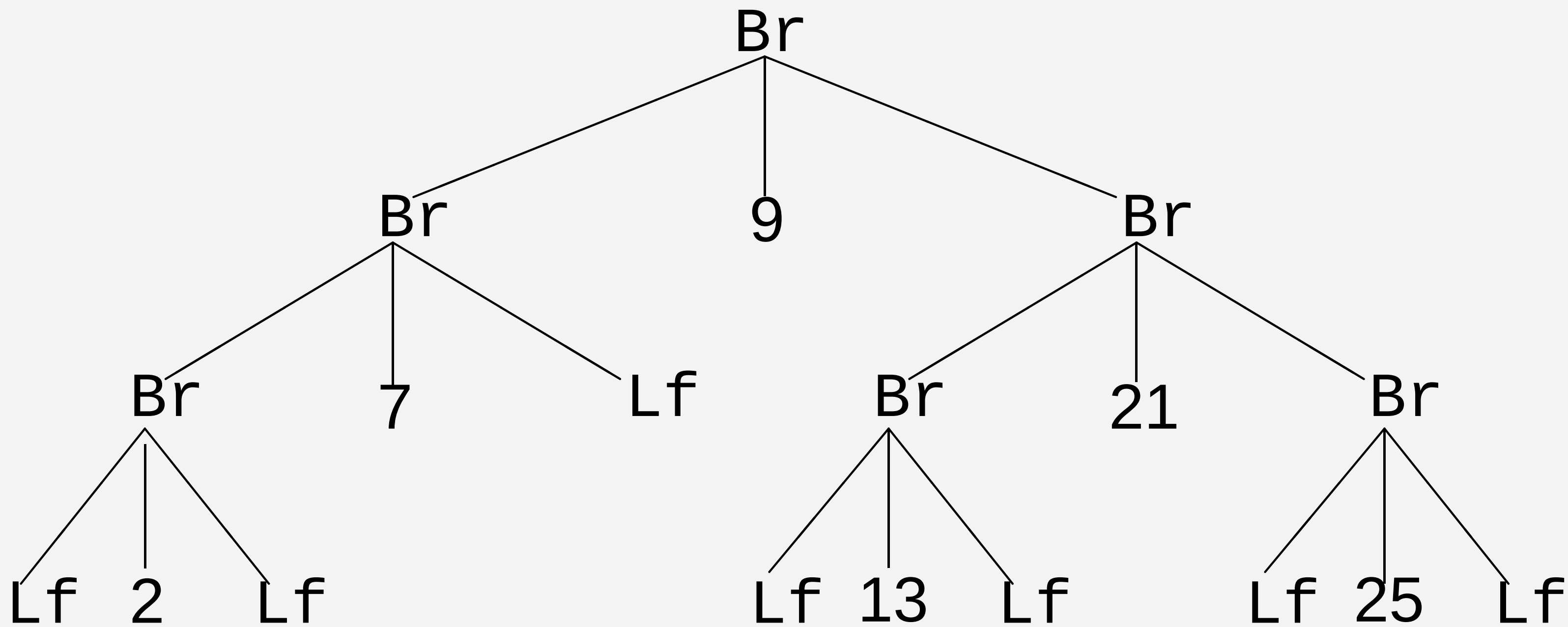
Finite Trees

- Algebraic Datatypes.
- Recursions following the structure of trees.

Trees

A *finite tree* is a value which may contain a subcomponent of the same type.

Example: A *binary search tree*



Condition: for every node containing the value x: every value in the left subtree is smaller then x, and every value in the right subtree is greater than x.

Binary Trees

A *recursive datatype* is used to represent values with are trees.

```
datatype tree = Lf | Br of tree*int*tree;
> datatype tree
> con Lf = Lf : tree
> con Br = fn : tree * int * tree -> tree
```

Binary Trees

A *recursive datatype* is used to represent values with are trees.

```
datatype tree = Lf | Br of tree*int*tree;
> datatype tree
> con Lf = Lf : tree
> con Br = fn : tree * int * tree -> tree
```

The two parts in the declaration are rules for generating trees:

- `Lf` is a tree
- if t_1, t_2 are trees, n is an integer, then `Br`(t_1, n, t_2) is a tree.

Binary Trees

A *recursive datatype* is used to represent values with are trees.

```
datatype tree = Lf | Br of tree*int*tree;
> datatype tree
> con Lf = Lf : tree
> con Br = fn : tree * int * tree -> tree
```

The two parts in the declaration are rules for generating trees:

- `Lf` is a tree
- if t_1, t_2 are trees, n is an integer, then `Br`(t_1, n, t_2) is a tree.

The tree from the previous slide is denoted by:

```
Br(Br(Br(Lf,2,Lf),7,Lf),
   9,
   Br(Br(Lf,13,Lf),21,Br(Lf,25,Lf)))
```

Binary search trees: Insertion

Recursion on the structure of trees:

- Constructors `Lf` and `Br` are used in patterns

```
fun insert(i, Lf)                = Br(Lf,i,Lf)
  | insert(i, tr as Br(t1,j,t2)) =
        case Int.compare(i,j) of
              EQUAL   => tr
            | LESS    => Br(insert(i,t1),j,t2)
            | GREATER => Br(t1,j,insert(i,t2))
```

- The search tree condition is an invariant for `insert`

Example:

```
- val t1 = Br(Lf, 3, Br(Lf, 5, Lf));

- val t2 = insert(4, t1);
> val t2 = Br(Lf, 3, Br(Br(Lf, 4, Lf), 5, Lf)) : tree
```

Binary search trees: `member` and `toList`

```
fun member(i, Lf)              = false
  | member(i, Br(t1,j,t2)) =
        case Int.compare(i,j) of
              EQUAL   => true
            | LESS    => member(i,t1)
            | GREATER => member(i,t2)
> val member = fn : int * tree -> bool
```

In-order traversal

```
fun toList Lf              = []
  | toList(Br(t1,j,t2)) = toList t1 @ [j] @ toList t2;
> val toList = fn : tree -> int list
```

gives a sorted list

```
- toList(Br(Br(Lf,1,Lf), 3, Br(Br(Lf,4,Lf), 5, Lf)));
> val it = [1, 3, 4, 5] : int list
```

Deletions in search trees

Delete minimal element in a search tree: `tree -> int * tree`

```
fun delMin(Br(Lf,i,t2)) = (i,t2)
  | delMin(Br(t1,i,t2)) = let val (m,t1') = delMin t1
                          in (m, Br(t1',i,t2)) end
```

Delete element in a search tree: `tree * int -> tree`

```
fun delete(Lf,_)             = Lf
  | delete(Br(t1,i,t2),j) =
      case Int.compare(i,j) of
           LESS    => Br(t1,i,delete(t2,j))
         | GREATER => Br(delete(t1,j),i,t2)
         | EQUAL   =>
            (case (t1,t2) of
                  (Lf,_) => t2
                | (_,Lf) => t1
                |  _     => let val (m,t2') = delMin t2
                            in Br(t1,m,t2') end)
```

Expression Trees

```
infix 6 ++ --;
infix 7 ** //;

datatype fexpr =
    Const of real
  | X
  | ++ of fexpr * fexpr | -- of fexpr * fexpr
  | ** of fexpr * fexpr | // of fexpr * fexpr

> datatype  fexpr
    con ** : fexpr * fexpr -> fexpr
    con ++ : fexpr * fexpr -> fexpr
    con -- : fexpr * fexpr -> fexpr
    con // : fexpr * fexpr -> fexpr
    con X : fexpr
    con Const : real -> fexpr
```

Expressions: Computation of values

```
comp : fexpr * real -> real
```

```
fun comp(Const r,_)    = r
  | comp(X,y)          = y
  | comp(fe1 ++ fe2,y) = comp(fe1,y) + comp(fe2,y)
  | comp(fe1 -- fe2,y) = comp(fe1,y) - comp(fe2,y)
  | comp(fe1 ** fe2,y) = comp(fe1,y) * comp(fe2,y)
  | comp(fe1 // fe2,y) = comp(fe1,y) / comp(fe2,y)
```

Example:

```
comp(X ** (Const 2.0 ++ X), 4.0);
> val it = 24.0 : real
```

Overview

Contents

- Higher-order functions
- Anonymous functions
- Higher-order list functions (in the library)
  - map
  - exists, all, filter
  - foldl, foldr

- Many recursive declarations follows the same schema.
  — Succinct declarations using higher-order functions.
- Parameterization of program modules

Higher-order functions

A function $f : \tau_1 \rightarrow \tau_2$ is a *higher-order function*, if a function type $\tau \rightarrow \tau'$ occurs in either τ_1 or τ_2 or both.

Functions are first class citizens

Higher-order functions

A function $f : \tau_1 \rightarrow \tau_2$ is a *higher-order function*, if a function type $\tau \rightarrow \tau'$ occurs in either τ_1 or τ_2 or both.

```
fun f x = let fun g y = x+y in g end;
> val f = fn : int -> int -> int

- f 2;
> val it = fn : int -> int

- it 3;
> val it = 5 : int
```

Functions are first class citizens

Anonymous functions

Expressions denoting functions can be written using `fn` expressions:

$$\texttt{fn}\ \ pat_1 \texttt{=>} e_1 \ \ |\ \ pat_2 \texttt{=>} e_2 \ \ |\ \ \cdots\ \ |\ \ pat_n \texttt{=>} e_n$$

Yields the value obtained by evaluation of the expression:

$$\begin{array}{l} \texttt{let}\ \ \texttt{fun}\ \ f\,x = \texttt{case}\ x\ \texttt{of} \\ \qquad\qquad pat_1 \texttt{=>} e_1 \ \ |\ \ pat_2 \texttt{=>} e_2 \ \ |\ \ \cdots\ \ |\ \ pat_n \texttt{=>} e_n \\ \texttt{in}\ \ f\ \texttt{end} \end{array}$$

Examples:

```
fn n => 2 * n;

fn 0 => false | _ => true;

fn r => Math.pi * r * r;
```

Declarations having the same structure

```
fun posList []       = []
  | posList (x::xs) = (x > 0)::posList xs;
val posList = fn : int list -> bool list

posList [4, ~5, 6];
> val it = [true,false,true] : bool list
```

Applies the function `fn x => x > 0` to each element in a list

```
fun addElems []           = []
  | addElems ((x,y)::zs) = (x + y)::addElems zs;
> val addElems = fn : (int * int) list -> int list

addElems [(1,2),(3,4)];
> val it = [3, 7] : int list
```

Applies the sum function `op+` to each pair of integers in a list

The function: `map`

Applies a function to each element in a list

$$\texttt{map}\ f\ [v_1, v_2, \ldots, v_n] = [f(v_1), f(v_2), \ldots, f(v_n)]$$

Declaration

Library function

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

Succinct declarations can be achieved using map, e.g.

```
val posList = map (fn x => x > 0);
> val posList = fn : int list -> bool list

val addElems = map op+
- val addElems = fn : (int * int) list -> int list
```

Declaration of higher-order functions

Commonly used form

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

General form

$$
\begin{array}{ll}
\texttt{fun} & f\ pat_{11}\ pat_{12}\ \ldots\ pat_{1n} = e_1 \\
\quad | & f\ pat_{21}\ pat_{22}\ \ldots\ pat_{2n} = e_2 \\
\quad | & \ldots \\
\quad | & f\ pat_{k1}\ pat_{k2}\ \ldots\ pat_{kn} = e_k
\end{array}
$$

Exercise

Declare a function

$$\texttt{g}\ [x_1, \ldots, x_n] = [x_1^2 + 1, \ldots, x_n^2 + 1]$$

Remember

$$\texttt{map}\ f\ [v_1, v_2, \ldots, v_n] \ = \ [f(v_1), f(v_2), \ldots, f(v_n)]$$

Higher-order list functions: `exists`

$$\texttt{exists}\; p\; xs = \begin{cases} \text{true} & \text{if } p(x) = \texttt{true} \text{ for some } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Declaration

Library function

```
fun exists p []       = false
  | exists p (x::xs) = p x orelse exists p xs;
> val exists = fn: ('a -> bool) -> 'a list -> bool
```

Example

```
exists (fn x => x>=2) [1,3,1,4];
> val it = true : bool
```

Exercise

Declare `member` function using `exists`.

```
infix member;

fun x member ys = exists   ????? ;
> val member = fn : ''a * ''a list -> bool
```

Remember

$$\texttt{exists}\ \mathrm{p}\ xs = \begin{cases} \text{true} & \text{if } \mathrm{p}(\mathrm{x}) = \texttt{true} \text{ for some } \mathrm{x} \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Higher-order list functions: `all`

$$\mathtt{all}\ p\ xs = \begin{cases} \text{true} & \text{if } p(x) = \text{true, for all elements } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Declaration

Library function

```
fun all p []       = true
  | all p (x::xs) = p x andalso all p xs;
> val all = fn: ('a -> bool) -> 'a list -> bool
```

Example

```
all (fn x => x>=2) [1,3,1,4];
> val it = false : bool
```

Exercise

Declare a function

$$\texttt{subset}(\mathit{xs}, \mathit{ys})$$

which is true when every element in the lists xs is in ys, and false otherwise.

Remember

$$\texttt{all}\ \mathrm{p}\ \mathit{xs} = \begin{cases} \text{true} & \text{if } \mathrm{p}(x) = \text{true, for all elements } x \text{ in } \mathit{xs} \\ \text{false} & \text{otherwise} \end{cases}$$

Higher-order list functions: `filter`

`filter` p *xs* is the list of those elements x of *xs* where $p(x) =$ `true`.

Declaration — Library function

```
fun filter p []       = []
  | filter p (x::xs) = if p x then x :: filter p xs
                       else filter p xs;
> val filter = fn: ('a -> bool) -> 'a list -> 'a list
```

Example

```
filter Char.isAlpha [#"1", #"p", #"F", #"-"];
> val it = [#"p", #"F"] : char list
```

where `Char.isAlpha` c is true iff $c \in \{\#"A", \ldots, \#"Z"\} \cup \{\#"a", \ldots, \#"z"\}$

Exercise

Declare a function

$$\texttt{inter}(xs, ys)$$

which contains the common elements of the lists xs and ys — i.e. their intersection.

Remember `filter` p xs is the list of those elements x of xs where $p(x) = \texttt{true}$.

Higher-order list functions: foldr (1)

foldr 'accumulates' a function f from a 'start value' b over the elements of a list $[x_1, x_2, \ldots, x_n]$ (from right to left):

$$\mathtt{foldr}\ f\ b\ [x_1, x_2, \ldots, x_{n-1}, x_n] = f(x_1, \underbrace{f(x_2, \ldots, f(x_{n-1}, f(x_n, b))\cdots)}_{\mathtt{foldr}\ f\ b\ [x_2, \ldots, x_{n-1}, x_n]})$$

Declaration

Library function

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

Example: the lenght function

```
fun length xs = foldr (fn (_,y) => y+1) 0 xs;
> val length = fn : 'a list -> int
length [4,5,6];
> val it = 3 : int
```

Higher-order list functions: `foldr` (2)

Accumulation of an infix operator $\oplus$. Evaluation is as follows

$$\texttt{foldr op}\oplus\ b\,[x_1,x_2,\ldots,x_n] \rightsquigarrow x_1 \oplus (x_2 \oplus \cdots \oplus (x_n \oplus b)\cdots)$$

Examples: Addition and Append

```
fun sumr xs = foldr op+ 0 xs;
> val sumr = fn : int list -> int

sumr [1,2,3,4];
> val it = 10 : int

fun append(xs,ys) = foldr op:: ys xs;
> val append = fn : 'a list * 'a list -> 'a list

append([1,2,3],[4,5]);
> val it = [1,2,3,4,5] : int list
```

Exercise: union of sets

Let an insertion function be declared by

```
fun insert(x, ys) = if x member ys then ys else x::ys
```

Declare a union function on sets.

Remember:

$$\texttt{foldr}\ \texttt{op}\oplus\ b\,[x_1,x_2,\ldots,x_n] \rightsquigarrow x_1 \oplus (x_2 \oplus \cdots \oplus (x_n \oplus b)\cdots)$$

Higher-order list functions: `foldl` (1)

`foldl` 'accumulates' a function f from a 'start value' b over the elements of a list $[x_1, x_2, \ldots, x_n]$ (from left to right):

$$\mathtt{foldl}\ f\ b\ [x_1,x_2,\ldots,x_{n-1},x_n] = \underbrace{f(x_n, f(x_{n-1},\ldots, f(x_2, \overbrace{f(x_1, b)}^{b'})\cdots))}_{\mathtt{foldl}\ f\ b'\ [x_2,\ldots,x_{n-1},x_n]}$$

Declaration

Library function

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

Higher-order list functions: `foldl` (2)

Accumulation of an infix operator $\oplus$. Evaluation is as follows

$$\texttt{foldl}\ \texttt{op}\oplus\ b\,[x_1,x_2,\ldots,x_n] \leadsto (x_n \oplus \cdots \oplus (x_2 \oplus (x_1 \oplus b))\cdots)$$

Examples

```
fun rev xs  = foldl op:: [] xs;
> val rev = fn : 'a list -> 'a list

rev [1,2,3];
> val it = [3, 2, 1] : int list
```