02157 Functional Programming

Finite Trees (I)

Michael R. Hansen

DTU Informatics
Department of Informatics and Mathematical Modelling

Overview

Finite Trees

- Algebraic Datatypes.
  - Non-recursive type declarations: Disjoint union (Lecture 4)
  - Recursive type declarations: Finite trees
- Recursions following the structure of trees
- Illustrative examples:
  - Search trees
  - Expression trees
  - File systems
  - . . .
- Mutual recursion, layered pattern, polymorphic type declarations

Overview

Finite Trees

- Algebraic Datatypes.
  - Non-recursive type declarations: Disjoint union (Lecture 4)
  - Recursive type declarations: Finite trees
- Recursions following the structure of trees
- Illustrative examples:
  - Search trees
  - Expression trees
  - File systems
  - . . .
- Mutual recursion, layered pattern, polymorphic type declarations

Overview

Finite Trees

- Algebraic Datatypes.
  - Non-recursive type declarations: Disjoint union (Lecture 4)
  - Recursive type declarations: Finite trees
- Recursions following the structure of trees
- Illustrative examples:
  - Search trees
  - Expression trees
  - File systems
  - . . .
- Mutual recursion, layered pattern, polymorphic type declarations

Overview

Finite Trees

- Algebraic Datatypes.
  - Non-recursive type declarations: Disjoint union (Lecture 4)
  - Recursive type declarations: Finite trees
- Recursions following the structure of trees
- Illustrative examples:
  - Search trees
  - Expression trees
  - File systems
  - . . .
- Mutual recursion, layered pattern, polymorphic type declarations

Finite trees

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

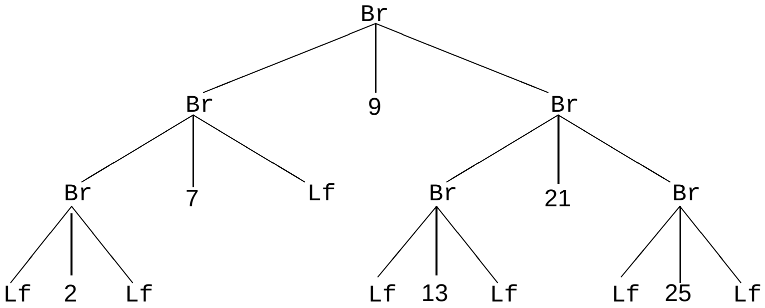
Example: A *binary search tree*

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.

Finite 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.

Example: Binary Trees

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

```
type Tree = Lf
          | Br of Tree*int*Tree;;

Lf;;
val it : Tree = Lf

Br;;
val it : Tree * int * Tree -> Tree = <fun:clo@4>
```

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)))
```

Example: Binary Trees

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

```
type Tree = Lf
          | Br of Tree*int*Tree;;

Lf;;
val it : Tree = Lf

Br;;
val it : Tree * int * Tree -> Tree = <fun:clo@4>
```

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 $\texttt{Br}(t_1, n, t_2)$ 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)))
```

Example: Binary Trees

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

```
type Tree = Lf
          | Br of Tree*int*Tree;;

Lf;;
val it : Tree = Lf

Br;;
val it : Tree * int * Tree -> Tree = <fun:clo@4>
```

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

```
let rec insert i = function
  | Lf                    ->  Br(Lf,i,Lf)
  | Br(t1,j,t2) as tr ->
      match compare i j with
      | 0             -> tr
      | n when n<0  -> Br(insert i t1 , j, t2)
      | _             -> Br(t1,j, insert i t2);;
val insert : int -> Tree -> Tree
```

Example:

```
let t1 = Br(Lf, 3, Br(Lf, 5, Lf));;
let t2 = insert 4 t1;;
val t2 : Tree = Br (Lf,3,Br (Br (Lf,4,Lf),5,Lf))
```

Binary search trees: Insertion

- Recursion on the structure of trees
- Constructors `Lf` and `Br` are used in patterns
- The search tree condition is an invariant for `insert`

```
let rec insert i = function
  | Lf                     ->  Br(Lf,i,Lf)
  | Br(t1,j,t2) as tr ->
      match compare i j with
      | 0             -> tr
      | n when n<0  -> Br(insert i t1 , j, t2)
      | _             -> Br(t1,j, insert i t2);;
val insert : int -> Tree -> Tree
```

Example:

```
let t1 = Br(Lf, 3, Br(Lf, 5, Lf));;
let t2 = insert 4 t1;;
val t2 : Tree = Br (Lf,3,Br (Br (Lf,4,Lf),5,Lf))
```

Binary search trees: Insertion

- Recursion on the structure of trees
- Constructors `Lf` and `Br` are used in patterns
- The search tree condition is an invariant for `insert`

```
let rec insert i = function
  | Lf                      ->  Br(Lf,i,Lf)
  | Br(t1,j,t2) as tr ->
      match compare i j with
      | 0           -> tr
      | n when n<0  -> Br(insert i t1 , j, t2)
      | _           -> Br(t1,j, insert i t2);;
val insert : int -> Tree -> Tree
```

Example:

```
let t1 = Br(Lf, 3, Br(Lf, 5, Lf));;
let t2 = insert 4 t1;;
val t2 : Tree = Br (Lf,3,Br (Br (Lf,4,Lf),5,Lf))
```

Binary search trees: `member` and `inOrder` traversal

```
let rec memberOf  i = function
  | Lf           -> false
  | Br(t1,j,t2) -> match compare i j with
                   | 0   -> true
                   | n when n<0 -> memberOf i t1
                   | _          -> memberOf i t2;;
val memberOf : int -> Tree -> bool
```

In-order traversal

```
let rec inOrder = function
  | Lf           -> []
  | Br(t1,j,t2) -> inOrder t1 @ [j] @ inOrder t2;;

val toList : Tree -> int list
```

gives a sorted list

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

Binary search trees: `member` and `inOrder` traversal

```
let rec memberOf  i = function
  | Lf          -> false
  | Br(t1,j,t2) -> match compare i j with
                   | 0   -> true
                   | n when n<0 -> memberOf i t1
                   | _          -> memberOf i t2;;
val memberOf : int -> Tree -> bool
```

In-order traversal

```
let rec inOrder = function
  | Lf          -> []
  | Br(t1,j,t2) -> inOrder t1 @ [j] @ inOrder t2;;

val toList : Tree -> int list
```

gives a sorted list

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

Binary search trees: member and inOrder traversal

```
let rec memberOf  i = function
  | Lf           -> false
  | Br(t1,j,t2) -> match compare i j with
                    | 0   -> true
                    | n when n<0 -> memberOf i t1
                    | _          -> memberOf i t2;;
val memberOf : int -> Tree -> bool
```

In-order traversal

```
let rec inOrder = function
  | Lf           -> []
  | Br(t1,j,t2) -> inOrder t1 @ [j] @ inOrder t2;;

val toList : Tree -> int list
```

gives a sorted list

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

Deletions in search trees

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

```
let rec delMin = function
  | Br(Lf,i,t2) -> (i,t2)
  | Br(t1,i,t2) -> let (m,t1') = delMin t1
                   (m, Br(t1',i,t2));;
```

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

```
let rec delete j = function
  | Lf          -> Lf
  | Br(t1,i,t2) ->
      match compare i j with
      | n when n<0 -> Br(t1,i,delete j t2)
      | n when n>0 -> Br(delete j t1,i,t2)
      | _          ->
           match t2 with
           | Lf -> t1
           | _  -> let (m,t2') = delMin t2
                   Br(t1,m,t2');;
```

Deletions in search trees

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

```
let rec delMin = function
  | Br(Lf,i,t2) -> (i,t2)
  | Br(t1,i,t2) -> let (m,t1') = delMin t1
                   (m, Br(t1',i,t2));;
```

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

```
let rec delete j = function
  | Lf          -> Lf
  | Br(t1,i,t2) ->
      match compare i j with
      | n when n<0 -> Br(t1,i,delete j t2)
      | n when n>0 -> Br(delete j t1,i,t2)
      | _          ->
          match t2 with
          | Lf -> t1
          | _  -> let (m,t2') = delMin t2
                  Br(t1,m,t2');;
```

Parameterize type declarations

The programs on search trees just requires an ordering on elements – they no not need to be integers.

A polymorphic tree type is declared as follows:

```
type Tree<'a> = Lf | Br of Tree<'a> * 'a * Tree<'a>;;
```

Program texts are unchanged (though polymorphic now), for example

```
let rec insert i = function
      ....
  | Br(t1,j,t2) as tr -> match compare i j with
      .... ;;
val insert: 'a -> Tree<'a> -> Tree<'a> when 'a: comparison

let ti = insert 4  (Br(Lf, 3, Br(Lf, 5, Lf)));;
val ti : Tree<int> = Br (Lf,3,Br (Br (Lf,4,Lf),5,Lf))

let ts = insert "4" (Br(Lf, "3", Br(Lf, "5", Lf)));;
val ts : Tree<string>
       = Br (Lf,"3",Br (Br (Lf,"4",Lf),"5",Lf))
```

Parameterize type declarations

The programs on search trees just requires an ordering on elements
– they no not need to be integers.

A polymorphic tree type is declared as follows:

```
type Tree<'a> = Lf | Br of Tree<'a> * 'a * Tree<'a>;;
```

Program texts are unchanged (though polymorphic now), for example

```
let rec insert i = function
      ....
  | Br(t1,j,t2) as tr -> match compare i j with
      .... ;;
val insert: 'a -> Tree<'a> -> Tree<'a> when 'a: comparison

let ti = insert 4  (Br(Lf, 3, Br(Lf, 5, Lf)));;
val ti : Tree<int> = Br (Lf,3,Br (Br (Lf,4,Lf),5,Lf))

let ts = insert "4" (Br(Lf, "3", Br(Lf, "5", Lf)));;
val ts : Tree<string>
       = Br (Lf,"3",Br (Br (Lf,"4",Lf),"5",Lf))
```

Parameterize type declarations

The programs on search trees just requires an ordering on elements
– they no not need to be integers.

A polymorphic tree type is declared as follows:

```
type Tree<'a> = Lf | Br of Tree<'a> * 'a * Tree<'a>;;
```

Program texts are unchanged (though polymorphic now), for example

```
let rec insert i = function
      ....
  | Br(t1,j,t2) as tr -> match compare i j with
      .... ;;
val insert: 'a -> Tree<'a> -> Tree<'a> when 'a: comparison

let ti = insert 4  (Br(Lf, 3, Br(Lf, 5, Lf)));;
val ti : Tree<int> = Br (Lf,3,Br (Br (Lf,4,Lf),5,Lf))

let ts = insert "4" (Br(Lf, "3", Br(Lf, "5", Lf)));;
val ts : Tree<string>
      = Br (Lf,"3",Br (Br (Lf,"4",Lf),"5",Lf))
```

Parameterize type declarations

The programs on search trees just requires an ordering on elements – they no not need to be integers.

A polymorphic tree type is declared as follows:

```
type Tree<'a> = Lf | Br of Tree<'a> * 'a * Tree<'a>;;
```

Program texts are unchanged (though polymorphic now), for example

```
let rec insert i = function
       ....
  | Br(t1,j,t2) as tr -> match compare i j with
       .... ;;
val insert: 'a -> Tree<'a> -> Tree<'a> when 'a: comparison

let ti = insert 4  (Br(Lf, 3, Br(Lf, 5, Lf)));;
val ti : Tree<int> = Br (Lf,3,Br (Br (Lf,4,Lf),5,Lf))

let ts = insert "4" (Br(Lf, "3", Br(Lf, "5", Lf)));;
val ts : Tree<string>
       = Br (Lf,"3",Br (Br (Lf,"4",Lf),"5",Lf))
```

Parameterize type declarations

The programs on search trees just requires an ordering on elements – they no not need to be integers.

A polymorphic tree type is declared as follows:

```
type Tree<'a> = Lf | Br of Tree<'a> * 'a * Tree<'a>;;
```

Program texts are unchanged (though polymorphic now), for example

```
let rec insert i = function
       ....
  | Br(t1,j,t2) as tr -> match compare i j with
       .... ;;
val insert: 'a -> Tree<'a> -> Tree<'a> when 'a: comparison

let ti = insert 4  (Br(Lf, 3, Br(Lf, 5, Lf)));;
val ti : Tree<int> = Br (Lf,3,Br (Br (Lf,4,Lf),5,Lf))

let ts = insert "4" (Br(Lf, "3", Br(Lf, "5", Lf)));;
val ts : Tree<string>
       = Br (Lf,"3",Br (Br (Lf,"4",Lf),"5",Lf))
```

Higher-order functions for tree traversals

For example

```
let rec inFoldBack f t e =
    match t with
    | Lf           -> e
    | Br(t1,x,t2) -> let er = inFoldBack f t2 e
                     inFoldBack f t1 (f x er);;
val inFoldBack: ('a -> 'b -> 'b) -> Tree<'a> -> 'b -> 'b
```

satisfies

$$\mathtt{inFoldBack}\, f\; t\; e = \mathtt{List.foldBack}\, f\; (\mathtt{inOrder}\; t)\; e$$

It traverses the tree without building the list- For example:

```
let ta = Br(Br(Br(Lf,-3,Lf),0,Br(Lf,2,Lf)),5,Br(Lf,7,Lf));;

inOrder ta;;
val it : int list = [-3; 0; 2; 5; 7]

inFoldBack (-) ta 0;;
val it : int = 1
```

Higher-order functions for tree traversals

For example

```
let rec inFoldBack f t e =
    match t with
    | Lf          -> e
    | Br(t1,x,t2) -> let er = inFoldBack f t2 e
                     inFoldBack f t1 (f x er);;
val inFoldBack: ('a -> 'b -> 'b) -> Tree<'a> -> 'b -> 'b
```

satisfies

$$\texttt{inFoldBack}\, f\; t\; e = \texttt{List.foldBack}\, f\,(\texttt{inOrder}\; t)\; e$$

It traverses the tree without building the list- For example:

```
let ta = Br(Br(Br(Lf,-3,Lf),0,Br(Lf,2,Lf)),5,Br(Lf,7,Lf));;

inOrder ta;;
val it : int list = [-3; 0; 2; 5; 7]

inFoldBack (-) ta 0;;
val it : int = 1
```

Example: Expression Trees

```
type Fexpr =
    | Const of float
    | X
    | Add of Fexpr * Fexpr
    | Sub of Fexpr * Fexpr
    | Mul of Fexpr * Fexpr
    | Div of Fexpr * Fexpr;;
```

Defines 6 constructors:

- `Const: float -> Fexpr`
- `X : Fexpr`
- `Add: Fexpr * Fexpr -> Fexpr`
- `Sub: Fexpr * Fexpr -> Fexpr`
- `Mul: Fexpr * Fexpr -> Fexpr`
- `Div: Fexpr * Fexpr -> Fexpr`

Example: Expression Trees

```
type Fexpr =
  | Const of float
  | X
  | Add of Fexpr * Fexpr
  | Sub of Fexpr * Fexpr
  | Mul of Fexpr * Fexpr
  | Div of Fexpr * Fexpr;;
```

Defines 6 constructors:

- `Const: float -> Fexpr`
- `X : Fexpr`
- `Add: Fexpr * Fexpr -> Fexpr`
- `Sub: Fexpr * Fexpr -> Fexpr`
- `Mul: Fexpr * Fexpr -> Fexpr`
- `Div: Fexpr * Fexpr -> Fexpr`

Symbolic Differentiation `D: Fexpr -> Fexpr`

A classic example in functional programming:

```
let rec D = function
  | Const _       -> Const 0.0
  | X             -> Const 1.0
  | Add(fe1,fe2) -> Add(D fe1,D fe2)
  | Sub(fe1,fe2) -> Sub(D fe1,D fe2)
  | Mul(fe1,fe2) -> Add(Mul(D fe1,fe2),Mul(fe1,D fe2))
  | Div(fe1,fe2) -> Div(
                      Sub(Mul(D fe1,fe2),Mul(fe1,D fe2)),
                      Mul(fe2,fe2));;
```

Notice the direct correspondence with the rules of differentiation.

Can be tried out directly, as tree are ”just” values, for example:

```
D(Add(Mul(Const 3.0, X), Mul(X, X)));;
val it : Fexpr =
  Add
    (Add (Mul (Const 0.0,X),Mul (Const 3.0,Const 1.0)),
     Add (Mul (Const 1.0,X),Mul (X,Const 1.0)))
```

Symbolic Differentiation `D: Fexpr -> Fexpr`

A classic example in functional programming:

```
let rec D = function
  | Const _       -> Const 0.0
  | X             -> Const 1.0
  | Add(fe1,fe2) -> Add(D fe1,D fe2)
  | Sub(fe1,fe2) -> Sub(D fe1,D fe2)
  | Mul(fe1,fe2) -> Add(Mul(D fe1,fe2),Mul(fe1,D fe2))
  | Div(fe1,fe2) -> Div(
                       Sub(Mul(D fe1,fe2),Mul(fe1,D fe2)),
                       Mul(fe2,fe2));;
```

Notice the direct correspondence with the rules of differentiation.

Can be tried out directly, as tree are ”just” values, for example:

```
D(Add(Mul(Const 3.0, X), Mul(X, X)));;
val it : Fexpr =
  Add
    (Add (Mul (Const 0.0,X),Mul (Const 3.0,Const 1.0)),
     Add (Mul (Const 1.0,X),Mul (X,Const 1.0)))
```

Expressions: Computation of values

Given a value (a float) for X, then every expression denote a float.

```
compute : float -> Fexpr -> float
```

```
let rec compute x = function
  | Const r        -> r
  | X              -> x
  | Add(fe1,fe2)  -> compute x fe1 + compute x fe2
  | Sub(fe1,fe2)  -> compute x fe1 - compute x fe2
  | Mul(fe1,fe2) -> compute x fe1 * compute x fe2
  | Div(fe1,fe2)  -> compute x fe1 / compute x fe2;;
```

Example:

```
compute 4.0 (Mul(X, Add(Const 2.0, X)));;
val it : float = 24.0
```

Expressions: Computation of values

Given a value (a float) for X, then every expression denote a float.

```
compute : float -> Fexpr -> float
```

```
let rec compute x = function
  | Const r        -> r
  | X              -> x
  | Add(fe1,fe2)  -> compute x fe1 + compute x fe2
  | Sub(fe1,fe2)  -> compute x fe1 - compute x fe2
  | Mul(fe1,fe2) -> compute x fe1 * compute x fe2
  | Div(fe1,fe2)  -> compute x fe1 / compute x fe2;;
```

Example:

```
compute 4.0 (Mul(X, Add(Const 2.0, X)));;
val it : float = 24.0
```

Expressions: Computation of values

Given a value (a float) for X, then every expression denote a float.

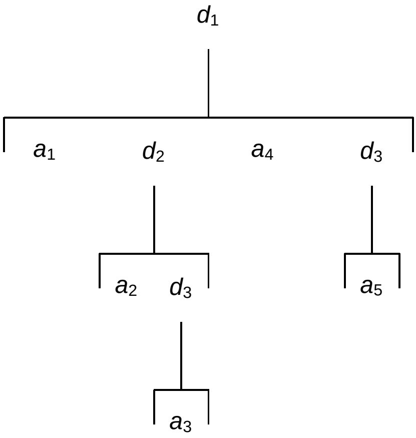
```
compute : float -> Fexpr -> float
```

```
let rec compute x = function
  | Const r        -> r
  | X              -> x
  | Add(fe1,fe2)   -> compute x fe1 + compute x fe2
  | Sub(fe1,fe2)   -> compute x fe1 - compute x fe2
  | Mul(fe1,fe2) -> compute x fe1 * compute x fe2
  | Div(fe1,fe2)   -> compute x fe1 / compute x fe2;;
```

Example:

```
compute 4.0 (Mul(X, Add(Const 2.0, X)));;
val it : float = 24.0
```

Mutual recursion. Example: File system



- A file system is a list of elements
- an element is a file or a directory, which is a named file system

Mutually recursive type declarations

- are combined using and

```
type FileSys = Element list
and Element =
  | File of string
  | Dir of string * FileSys
```

```
let d1 =
  Dir("d1",[File "a1";
            Dir("d2", [File "a2";
                       Dir("d3", [File "a3"])]);
            File "a4";
            Dir("d3", [File "a5"])
           ])
```

The type of d1 is ?

Mutually recursive type declarations

- are combined using and

```
type FileSys = Element list
and Element =
   | File of string
   | Dir of string * FileSys

let d1 =
  Dir("d1",[File "a1";
            Dir("d2", [File "a2";
                       Dir("d3", [File "a3"])]);
            File "a4";
            Dir("d3", [File "a5"])
           ])
```

The type of d1 is ?

Mutually recursive type declarations

- are combined using and

```
type FileSys = Element list
and Element =
  | File of string
  | Dir of string * FileSys

let d1 =
  Dir("d1",[File "a1";
            Dir("d2", [File "a2";
                       Dir("d3", [File "a3"])]);
            File "a4";
            Dir("d3", [File "a5"])
           ])
```

The type of d1 is ?

Mutually recursive function declarations

- are combined using and

Example: extract the names occurring in file systems and elements.

```
let rec namesFileSys = function
  | []    -> []
  | e::es -> (namesElement e) @ (namesFileSys es)
and namesElement = function
  | File s    -> [s]
  | Dir(s,fs) -> s :: (namesFileSys fs) ;;
val namesFileSys : Element list -> string list
val namesElement : Element -> string list
```

```
namesElement d1 ;;
val it : string list = ["d1"; "a1"; "d2"; "a2";
                        "d3"; "a3"; "a4"; "d3"; "a5"]
```

Mutually recursive function declarations

- are combined using and

Example: extract the names occurring in file systems and elements.

```
let rec namesFileSys = function
  | []    -> []
  | e::es -> (namesElement e) @ (namesFileSys es)
and namesElement = function
  | File s    -> [s]
  | Dir(s,fs) -> s :: (namesFileSys fs) ;;
val namesFileSys : Element list -> string list
val namesElement : Element -> string list


namesElement d1 ;;
val it : string list = ["d1"; "a1"; "d2"; "a2";
                        "d3"; "a3"; "a4"; "d3"; "a5"]
```

Summary

Finite Trees

- concepts
- illustrative examples

Notice the strength of having trees as values.

Notice that polymorphic types and mutual recursion are NOT biased to trees.

Summary

Finite Trees

- concepts
- illustrative examples

Notice the strength of having trees as values.

Notice that polymorphic types and mutual recursion are NOT biased to trees.

Summary

Finite Trees

- concepts
- illustrative examples

Notice the strength of having trees as values.

Notice that polymorphic types and mutual recursion are NOT biased to trees.