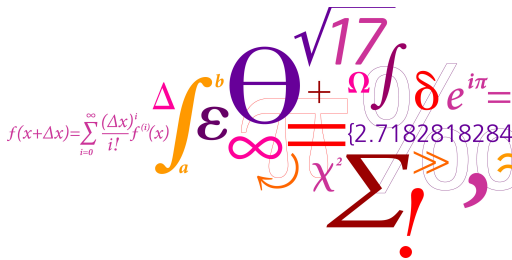


02157 Functional Programming

Finite Trees (I)

Michael R. Hansen



DTU Compute

Department of Applied Mathematics and Computer Science

- Finite trees
- Functional, Structural and Property tests
by example

using xUnit

Finite trees

Finite trees

- **recursive** declarations of algebraic types
- **meaning** of type declarations: rules generating values

Finite trees

- **recursive** declarations of algebraic types
- **meaning** of type declarations: rules generating values
- typical recursions following the structure of trees

Finite trees

- **recursive** declarations of algebraic types
- **meaning** of type declarations: rules generating values
- typical recursions following the structure of trees
- trees with a **fixed branching structure**

Finite trees

- **recursive** declarations of algebraic types
- **meaning** of type declarations: rules generating values
- typical recursions following the structure of trees
- trees with a **fixed branching structure**
- trees with a **variable number of sub-trees**

Finite trees

- **recursive** declarations of algebraic types
- **meaning** of type declarations: rules generating values
- typical recursions following the structure of trees
- trees with a **fixed branching structure**
- trees with a **variable number of sub-trees**
- illustrative examples

Finite trees

- **recursive** declarations of algebraic types
- **meaning** of type declarations: rules generating values
- typical recursions following the structure of trees
- trees with a **fixed branching structure**
- trees with a **variable number of sub-trees**
- illustrative examples

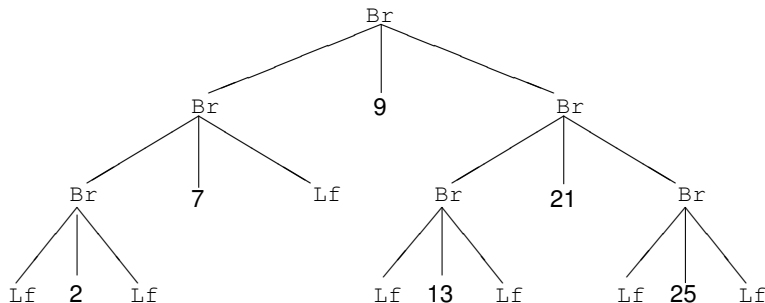
Mutually recursive type and function declarations

A *finite tree* is a value containing sub-components of the same type

Finite trees

A *finite tree* is a value containing sub-components of the *same type*

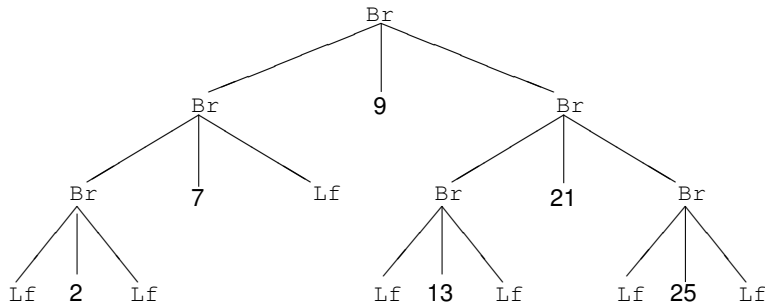
Example: A *binary tree*



Finite trees

A *finite tree* is a value containing sub-components of the *same type*

Example: A *binary tree*



A tree is a connected, acyclic, undirected graph, where

- the top node (carrying value 9) is called the **root**
- a **branch node** has two **children**
- a node without children is called a **leaf**

constructor **Br**
constructor **Lf**

Example: Binary Trees

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

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

Example: Binary Trees

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

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

The declaration provides **rules** for generating trees:

- 1 **Lf** is a tree
- 2 if t_1, t_2 are trees and n is an integer, then $\text{Br}(t_1, n, t_2)$ is a tree.
- 3 the type **Tree** contains no other values than those generated by repeated use of Rules 1. and 2.

Example: Binary Trees

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

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

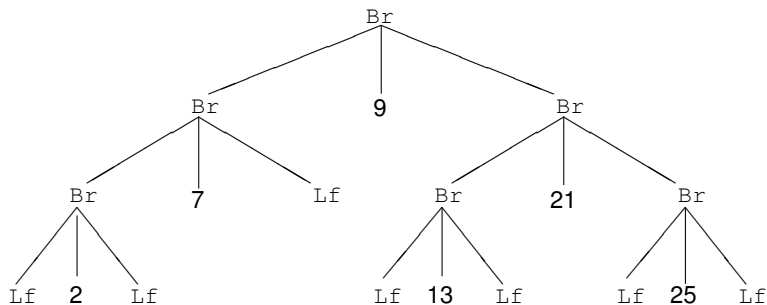
The declaration provides **rules** for generating trees:

- 1 **Lf** is a tree
- 2 if t_1, t_2 are trees and n is an integer, then $\text{Br}(t_1, n, t_2)$ is a tree.
- 3 the type **Tree** contains no other values than those generated by repeated use of Rules 1. and 2.

The tags **Lf** and **Br** are called **constructors**:

```
Lf   : Tree  
Br   : Tree * int * Tree → Tree
```

Example: Binary Trees



Corresponding F#-value:

```

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

Traversals of binary trees

- Pre-order traversal: First visit the root node, then traverse the left sub-tree in pre-order and finally traverse the right sub-tree in pre-order.
- In-order traversal: First traverse the left sub-tree in in-order, then visit the root node and finally traverse the right sub-tree in in-order.
- Post-order traversal: First traverse the left sub-tree in post-order, then traverse the right sub-tree in post-order and finally visit the root node.

Traversals of binary trees

- Pre-order traversal: First visit the root node, then traverse the left sub-tree in pre-order and finally traverse the right sub-tree in pre-order.
- In-order traversal: First traverse the left sub-tree in in-order, then visit the root node and finally traverse the right sub-tree in in-order.
- Post-order traversal: First traverse the left sub-tree in post-order, then traverse the right sub-tree in post-order and finally visit the root node.

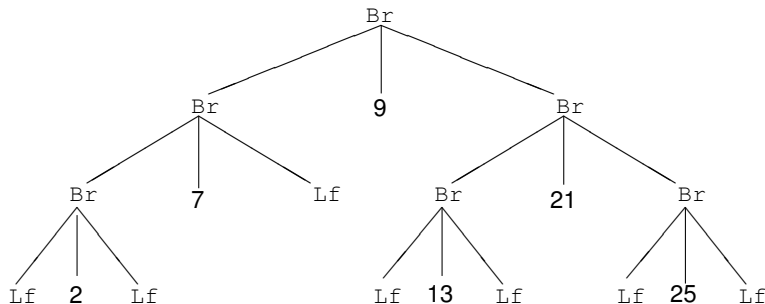
In-order traversal

```
let rec inOrder =  
  function  
    | Lf          -> []  
    | Br(t1,j,t2) -> inOrder t1 @ [j] @ inOrder t2;;  
val toList : Tree -> int 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 tree

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

Example: A *binary search tree*



- Recursion following the structure of trees

Binary search trees: Insertion

- Recursion following the structure of trees
- Constructors `Lf` and `Br` are used in `patterns` to decompose a tree into its parts

Binary search trees: Insertion

- Recursion following the structure of trees
- Constructors `Lf` and `Br` are used in **patterns** to decompose a tree into its parts
- Constructors `Lf` and `Br` are used in **expressions** to construct a tree from its parts

Binary search trees: Insertion

- Recursion following the structure of trees
- Constructors `Lf` and `Br` are used in **patterns** to decompose a tree into its parts
- Constructors `Lf` and `Br` are used in **expressions** to construct a tree from its parts
- The search tree condition is an **invariant** for `insert`

Binary search trees: Insertion

- Recursion following the structure of trees
- Constructors `Lf` and `Br` are used in **patterns** to decompose a tree into its parts
- Constructors `Lf` and `Br` are used in **expressions** to construct a tree from its parts
- 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 -> // Layered pattern
      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
```

Binary search trees: Insertion

- Recursion following the structure of trees
- Constructors `Lf` and `Br` are used in **patterns** to decompose a tree into its parts
- Constructors `Lf` and `Br` are used in **expressions** to construct a tree from its parts
- 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 -> // Layered pattern
      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))
```

```
let rec contains i =  
  function  
  | Lf          -> false  
  | Br(_,j,_)   when i=j -> true  
  | Br(t1,j,_)  when i<j -> contains i t1  
  | Br(_,j,t2)  -> contains i t2;;  
val contains : int -> Tree -> bool  
  
let t = Br(Br(Br(Lf,2,Lf),7,Lf),  
           9,  
           Br(Br(Lf,13,Lf),21,Br(Lf,25,Lf)));;  
  
contains 21 t;;  
val it : bool = true  
  
contains 4 t;;  
val it : bool = false
```

Parameterize type declarations

The programs on search trees require only an ordering on elements
– they do not need to be integers.

Parameterize type declarations

The programs on search trees require only an ordering on elements
– they do not need to be integers.

A polymorphic tree type is declared as follows:

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

Parameterize type declarations

The programs on search trees require only an ordering on elements
– they do 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 text is 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
```

Parameterize type declarations

The programs on search trees require only an ordering on elements
– they do 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 text is 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))
```

Parameterize type declarations

The programs on search trees require only an ordering on elements
– they do 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 text is 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))
```

- Declaration of a recursive algebraic data type, that is, a type for a finite tree
- Meaning of the type declaration:
 - rules for generating values
- Archetypical functions on trees:
 - gathering information from a tree
 - inspecting a tree
 - construction of a new tree

Example: `inOrder`

Example: `contains`

Example: `insert`

Consider $f(x)$:

$$3 \cdot (x - 1) - 2 \cdot x$$

Manipulation of arithmetical expressions

Consider $f(x)$:

$$3 \cdot (x - 1) - 2 \cdot x$$

We may be interested in

- computation of values, e.g. $f(2)$
- differentiation, e.g. $f'(x) = (3 \cdot 1 + 0 \cdot (x - 1)) - (2 \cdot 1 + 0 \cdot x)$
- simplification of the expressions, e.g. $f'(x) = 1$
-

Manipulation of arithmetical expressions

Consider $f(x)$:

$$3 \cdot (x - 1) - 2 \cdot x$$

We may be interested in

- computation of values, e.g. $f(2)$
- differentiation, e.g. $f'(x) = (3 \cdot 1 + 0 \cdot (x - 1)) - (2 \cdot 1 + 0 \cdot x)$
- simplification of the expressions, e.g. $f'(x) = 1$
-

We would like a suitable representation of such arithmetical expressions that supports the above manipulations

Manipulation of arithmetical expressions

Consider $f(x)$:

$$3 \cdot (x - 1) - 2 \cdot x$$

We may be interested in

- computation of values, e.g. $f(2)$
- differentiation, e.g. $f'(x) = (3 \cdot 1 + 0 \cdot (x - 1)) - (2 \cdot 1 + 0 \cdot x)$
- simplification of the expressions, e.g. $f'(x) = 1$
-

We would like a suitable representation of such arithmetical expressions that supports the above manipulations

How would you visualize the expressions as a tree?

- root?
- leaves?
- branches?

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;;
```

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

- Can you write 3 values of type Fexpr?
- Drawings of trees?

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

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

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;;
```

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
```

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

Declare a function

```
substX: Fexpr -> Fexpr -> Fexpr
```

so that `substX  $e'$   $e$` is the expression obtained from e by substituting every occurrence of `X` with e'

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

Declare a function

```
substX: Fexpr -> Fexpr -> Fexpr
```

so that `substX  $e'$   $e$` is the expression obtained from e by substituting every occurrence of x with e'

For example:

```
let ex = Add(Sub(X, Const 2.0), Mul(Const 4.0, X));;

substX (Div(X,X)) ex;;
val it : Fexpr =
  Add(Sub(Div(X,X), Const 2.0), Mul(Const 4.0, Div(X,X)))
```

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.

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)))
```

Trees with a variable number of sub-trees

An archetypical declaration:

```
type ListTree<'a> = Node of 'a * (ListTree<'a> list)
```

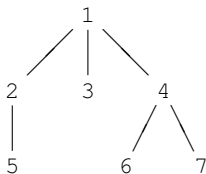
- `Node(x, [])` represents a leaf tree containing the value x
- `Node(x, [ $t_0$ ; ... ;  $t_{n-1}$ ])` represents a tree with value x in the root and with n sub-trees represented by the values $t_0, \dots, t_{n-1}$

Trees with a variable number of sub-trees

An archetypical declaration:

```
type ListTree<'a> = Node of 'a * (ListTree<'a> list)
```

- `Node(x, [])` represents a leaf tree containing the value x
- `Node(x, [ $t_0$ ; ... ;  $t_{n-1}$ ])` represents a tree with value x in the root and with n sub-trees represented by the values $t_0, \dots, t_{n-1}$

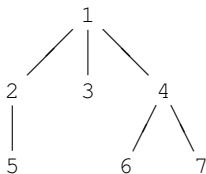


Trees with a variable number of sub-trees

An archetypical declaration:

```
type ListTree<'a> = Node of 'a * (ListTree<'a> list)
```

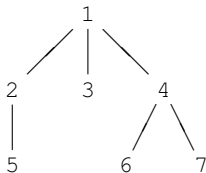
- `Node(x, [])` represents a leaf tree containing the value x
- `Node(x, [ $t_0$ ; ... ;  $t_{n-1}$ ])` represents a tree with value x in the root and with n sub-trees represented by the values $t_0, \dots, t_{n-1}$



It is represented by the value `t1` where

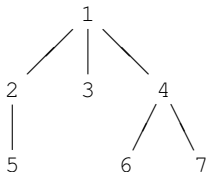
```
let t7 = Node(7, []);;      let t6 = Node(6, []);;
let t5 = Node(5, []);;      let t3 = Node(3, []);;
let t2 = Node(2, [t5]);;    let t4 = Node(4, [t6; t7]);;
let t1 = Node(1, [t2; t3; t4]);;
```

Depth-first traversal of a ListTree



Corresponds to the following order of the elements: 1, 2, 5, 3, 4, 6, 7

Depth-first traversal of a ListTree



Corresponds to the following order of the elements: 1, 2, 5, 3, 4, 6, 7

Invent **a more general function** traversing a list of List trees:

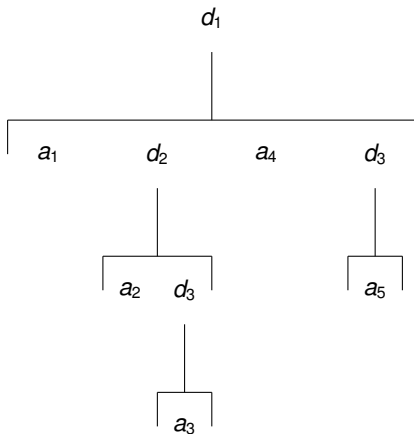
```

let rec depthFirstList =
  function
  | [] -> []
  | Node(n,ts)::trest -> n::depthFirstList(ts @ trest)
depthFirstList : ListTree<'a> list -> 'a list

let depthFirst t = depthFirstList [t]
depthFirst1 : t::ListTree<'a> -> 'a list

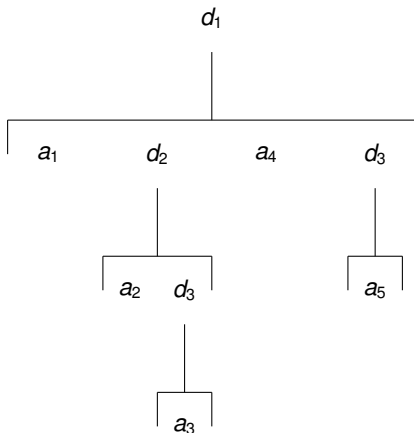
depthFirst t1;;
val it : int list = [1; 2; 5; 3; 4; 6; 7]
  
```

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**

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**

We focus on structure now – not on file content

Mutually recursive type declarations

- are combined using **and**

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

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"])
  ])
```

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
```

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"]
```

Mini-project 2: A tiny coherent story

Semantics of expressions

```
type Exp = Add of Exp*Exp | X | ...  
sem: Exp -> int -> int
```

Mini-project 2: A tiny coherent story

Semantics of expressions

```
type Exp = Add of Exp*Exp | X | ...  
sem: Exp -> int -> int
```

A stack machine

```
type Instruction = ...  
exec: Instruction list -> int
```

Mini-project 2: A tiny coherent story

Semantics of expressions

```
type Exp = Add of Exp*Exp | X | ...  
sem: Exp -> int -> int
```

A stack machine

```
type Instruction = ...  
exec: Instruction list -> int
```

Compilation

```
compile: Exp -> int -> Instruction list
```

Mini-project 2: A tiny coherent story

Semantics of expressions

```
type Exp = Add of Exp*Exp | X | ...  
sem: Exp -> int -> int
```

A stack machine

```
type Instruction = ...  
exec: Instruction list -> int
```

Compilation

```
compile: Exp -> int -> Instruction list
```

Compiler is correct if $\text{sem } e \ x = \text{exec}(\text{compile } e \ x)$.

Mini-project 2: A tiny coherent story

Semantics of expressions

```
type Exp = Add of Exp*Exp | X | ...  
sem: Exp -> int -> int
```

A stack machine

```
type Instruction = ...  
exec: Instruction list -> int
```

Compilation

```
compile: Exp -> int -> Instruction list
```

Compiler is correct if $\text{sem } e \ x = \text{exec}(\text{compile } e \ x)$.

Optimization: Reduction of expressions

```
red: Exp -> Exp
```

preserving semantics: $\text{sem } e \ x = \text{sem}(\text{red } e) \ x$.

Mini-project 2: A tiny coherent story

Semantics of expressions

```
type Exp = Add of Exp*Exp | X | ...  
sem: Exp -> int -> int
```

A stack machine

```
type Instruction = ...  
exec: Instruction list -> int
```

Compilation

```
compile: Exp -> int -> Instruction list
```

Compiler is correct if $\text{sem } e \ x = \text{exec}(\text{compile } e \ x)$.

Optimization: Reduction of expressions

```
red: Exp -> Exp
```

preserving semantics: $\text{sem } e \ x = \text{sem}(\text{red } e) \ x$.

A appetizer illustrating fundamental CS concepts

- exercising concepts on finite trees

Finite Trees

- recursive declarations of algebraic types
- meaning of type declarations: rules generating values

Finite Trees

- recursive declarations of algebraic types
- meaning of type declarations: rules generating values
- typical recursions following the structure of trees

Finite Trees

- recursive declarations of algebraic types
- meaning of type declarations: rules generating values
- typical recursions following the structure of trees
- trees with a fixed branching structure

Finite Trees

- recursive declarations of algebraic types
 - meaning of type declarations: rules generating values
 - typical recursions following the structure of trees
 - trees with a fixed branching structure
 - trees with a variable number of sub-trees including two techniques
 - use of mutually recursive function declarations
 - use of a more general helper function
- to handle the arbitrary branching

Tree list $\rightarrow \dots$

Finite Trees

- recursive declarations of algebraic types
- meaning of type declarations: rules generating values
- typical recursions following the structure of trees
- trees with a fixed branching structure
- trees with a variable number of sub-trees including two techniques
 - use of mutually recursive function declarations
 - use of a more general helper function

Tree list $\rightarrow \dots$

to handle the arbitrary branching

- illustrative examples

Finite Trees

- recursive declarations of algebraic types
- meaning of type declarations: rules generating values
- typical recursions following the structure of trees
- trees with a fixed branching structure
- trees with a variable number of sub-trees including two techniques
 - use of mutually recursive function declarations
 - use of a more general helper function

Tree list $\rightarrow \dots$

to handle the arbitrary branching

- illustrative examples

Mutually recursive type and function declarations

On Software Test

On Software Test

- functional and structural tests
- an eye to property based tests

On Software Test

- functional and structural tests
- an eye to property based tests
- A four-project solution using [xUnit](#)

Systematic approaches aiming at finding bugs in programs

Systematic approaches aiming at finding bugs in programs

Different kinds of errors in programs:

- **Syntax errors** like `3 * (2 - x` missing `)`
- **Type errors**, like `3 * ("2" - x)`
- **Semantic error**. A syntactically well-formed and type correct program computes a wrong answer
not discovered by the compiler

Systematic approaches aiming at finding bugs in programs

Different kinds of errors in programs:

- **Syntax errors** like $3 * (2 - x$ missing $)$
- **Type errors**, like $3 * ("2" - x)$
- **Semantic error**. A syntactically well-formed and type correct program computes a wrong answer
not discovered by the compiler

E.W. Dijkstra, EDW249, 1970:

*Program testing can be used to show the presence of bugs,
but never to show their absence!*

Systematic approaches aiming at finding bugs in programs

Different kinds of errors in programs:

- **Syntax errors** like `3 * (2 - x` missing `)`
- **Type errors**, like `3 * ("2" - x)`
- **Semantic error**. A syntactically well-formed and type correct program computes a wrong answer
not discovered by the compiler

E.W. Dijkstra, EDW249, 1970:

*Program testing can be used to show the presence of bugs,
but never to show their absence!*

In "our" context:

- test for finding semantic errors in functions Unit testing

Two important (among an abundance of) techniques

- **Structural test** (or white-box test or internal test)

The tester inspects **the program** and constructs a **test suite** that shows that all branches can be executed

- **Functional test** (or black-box test or external test)

The tester focuses on **the requirement** (*not the program*) and constructs a **test suite** aiming at justifying that the program solves the task it is supposed to solve.

Two important (among an abundance of) techniques

- **Structural test** (or white-box test or internal test)

The tester inspects **the program** and constructs a **test suite** that shows that all branches can be executed

- **Functional test** (or black-box test or external test)

The tester focuses on **the requirement** (*not the program*) and constructs a **test suite** aiming at justifying that the program solves the task it is supposed to solve.

The two techniques **complement** each other, for example

- a structural test cannot detect that a sub-task is not implemented
- a functional test cannot detect the presence of dead code

Two important (among an abundance of) techniques

- **Structural test** (or white-box test or internal test)

The tester inspects **the program** and constructs a **test suite** that shows that all branches can be executed

- **Functional test** (or black-box test or external test)

The tester focuses on **the requirement** (*not the program*) and constructs a **test suite** aiming at justifying that the program solves the task it is supposed to solve.

The two techniques **complement** each other, for example

- a structural test cannot detect that a sub-task is not implemented
- a functional test cannot detect the presence of dead code

A **test suite** must at least include

- input
- expected output
- It is used **to assess the quality of the tests**

Functional test: an example

```
type Lid      = string
type Flight   = string
type Airport  = string
```

```
type Route          = (Flight * Airport) list
type LuggageCatalogue = (Lid * Route) list
```

```
withFlight: Flight -> LuggageCatalogue -> Lid list
```

Requirement: `withFlight f lc` gives the list of identifiers of luggage that are on flight `f` according to `lc`

Functional test: an example

```
type Lid      = string
type Flight   = string
type Airport  = string
```

```
type Route          = (Flight * Airport) list
type LuggageCatalogue = (Lid * Route) list
```

```
withFlight: Flight -> LuggageCatalogue -> Lid list
```

Requirement: `withFlight f lc` gives the list of identifiers of luggage that are on flight `f` according to `lc`

Destructive thinking when designing the test suite:

- Which mistakes may occur in envisioned implementations?

Functional test: an example

```
type Lid      = string
type Flight   = string
type Airport  = string
```

```
type Route          = (Flight * Airport) list
type LuggageCatalogue = (Lid * Route) list
```

```
withFlight: Flight -> LuggageCatalogue -> Lid list
```

Requirement: `withFlight f lc` gives the list of identifiers of luggage that are on flight `f` according to `lc`

Destructive thinking when designing the test suite:

- Which mistakes may occur in envisioned implementations?

Input should be covered by a finite number of test cases

Functional test: an example test suite (1)

Requirement: `withFlight f lc` gives the list of identifiers of luggage that are on flight `f` according to `lc`

Functional test: an example test suite (1)

Requirement: `withFlight f lc` gives the list of identifiers of luggage that are on flight `f` according to `lc`

TestId	Input Property
A	Empty catalogue
B1	one element cat., flight first in route
B2	one element cat., flight later in route
B3	one element cat., flight not in route
C1	one+ element cat, flight not in any route
C2	one+ element cat, flight in one route
C3	one+ element cat, flight in several routes

Functional test: an example test suite (1)

Requirement: `withFlight f lc` gives the list of identifiers of luggage that are on flight `f` according to `lc`

TestId	Input Property
A	Empty catalogue
B1	one element cat., flight first in route
B2	one element cat., flight later in route
B3	one element cat., flight not in route
C1	one+ element cat, flight not in any route
C2	one+ element cat, flight in one route
C3	one+ element cat, flight in several routes

Are there superfluous test cases?

Functional test: an example test suite (1)

Requirement: `withFlight f lc` gives the list of identifiers of luggage that are on flight `f` according to `lc`

TestId	Input Property
A	Empty catalogue
B1	one element cat., flight first in route
B2	one element cat., flight later in route
B3	one element cat., flight not in route
C1	one+ element cat, flight not in any route
C2	one+ element cat, flight in one route
C3	one+ element cat, flight in several routes

Are there superfluous test cases?

Destructive thinking: Are these cases sufficient?

Functional test: an example test suite (1)

Requirement: `withFlight f lc` gives the list of identifiers of luggage that are on flight `f` according to `lc`

TestId	Input Property
A	Empty catalogue
B1	one element cat., flight first in route
B2	one element cat., flight later in route
B3	one element cat., flight not in route
C1	one+ element cat, flight not in any route
C2	one+ element cat, flight in one route
C3	one+ element cat, flight in several routes

Are there superfluous test cases?

Destructive thinking: Are these cases sufficient?

What about, for example

- C2a: one+ element cat, flight appears first in a route ?
- C2b: one+ element cat, flight appears inside a route ?
- C2c: one+ element cat, flight appears at the end of a route ?

Functional test: an example test suite (2)

TestId	Input (f/cat)	expected output contains
A	"f" []	nothing
...	...	...
C2c	"f" catC2c	"lid2"
C3	...	

where

```
let catC2c = [("lid1", [("f1", "a1")]);
              ("lid2", [("f2", "a2"); ("f", "a")]) ]
```

Functional test: an example test suite (2)

TestId	Input (f/cat)	expected output contains
A	"f" []	nothing
...	...	...
C2c	"f" catC2c	"lid2"
C3	...	

where

```
let catC2c = [ ("lid1", [ ("f1", "a1") ] );
               ("lid2", [ ("f2", "a2"); ("f", "a") ] ) ]
```

Functional test:

- Based on educated guesswork: What can possibly go wrong?
You cannot be sure that all errors will be spotted.

Functional test: an example test suite (2)

TestId	Input (f/cat)	expected output contains
A	"f" []	nothing
...	...	...
C2c	"f" catC2c	"lid2"
C3	...	

where

```
let catC2c = [ ("lid1", [ ("f1", "a1") ] );
               ("lid2", [ ("f2", "a2"); ("f", "a") ] ) ]
```

Functional test:

- Based on educated guesswork: What can possibly go wrong?
You cannot be sure that all errors will be spotted.
- It is useful to design the functional test during program development.

Sharpens the understanding of the problem and its solution.

Functional test: an example test suite (2)

TestId	Input (f/cat)	expected output contains
A	"f" []	nothing
...	...	...
C2c	"f" catC2c	"lid2"
C3	...	

where

```
let catC2c = [("lid1", [("f1", "a1")]);
              ("lid2", [("f2", "a2"); ("f", "a")]) ]
```

Functional test:

- Based on educated guesswork: What can possibly go wrong?
You cannot be sure that all errors will be spotted.
- It is useful to design the functional test during program development.

Sharpens the understanding of the problem and its solution.

Is an empty route in a catalogue meaningful?

Structural test: an example test suite (1)

```
type LuggageCatalogue = (Lid * Route) list

// routeOf: Lid -> LuggageCatalogue -> Route option
let rec routeOf lid =
  function
  | []                                -> None                // c1
  | (lid',r)::_ when lid=lid'        -> Some r                // c2
  | _::rest                          -> routeOf lid rest    // c3
```

A structural test should exercise

- every branch of the program and 0, 1 and more recursive calls.

Structural test: an example test suite (1)

```

type LuggageCatalogue = (Lid * Route) list

// routeOf: Lid -> LuggageCatalogue -> Route option
let rec routeOf lid =
  function
  | []                                -> None           // c1
  | (lid',r)::_ when lid=lid'         -> Some r         // c2
  | _::rest                          -> routeOf lid rest // c3

```

A structural test should exercise

- every branch of the program and 0, 1 and more recursive calls.

Choice	TestId	Input Property	Recursive calls
c1	A	empty cat.	0
c2	B	non-empty cat	0
c3	C	one element cat, lid not found	1
c3	D	more than one element cat, lid found	> 1

Structural test: an example test suite (1)

```

type LuggageCatalogue = (Lid * Route) list

// routeOf: Lid -> LuggageCatalogue -> Route option
let rec routeOf lid =
  function
  | []                                -> None           // c1
  | (lid', r)::_ when lid=lid'       -> Some r          // c2
  | _::rest                          -> routeOf lid rest // c3

```

A structural test should exercise

- every branch of the program and 0, 1 and more recursive calls.

Choice	TestId	Input Property	Recursive calls
c1	A	empty cat.	0
c2	B	non-empty cat	0
c3	C	one element cat, lid not found	1
c3	D	more than one element cat, lid found	> 1

Test C makes Test A superfluous

Structural test: an example test suite (2)

```

let rec routeOf lid =
  function
  | []                                -> None                // c1
  | (lid', r) :: _ when lid=lid'     -> Some r                // c2
  | _ :: rest                        -> routeOf lid rest    // c3

```

TestId	Input (lid/cat)	expected output
B	"l0" [("l0", r_0)]	Some r_0
C	"l4" [("l0", r_0)]	None
D	"l2" [("l0", r_0); ("l1", r_1); ("l2", r_2)]	Some r_2

where

- $r_i = [(\text{"f0"}, \text{"a0"}); \dots; (\text{"fi"}, \text{"ai"})]$, for $i = 0, 1, 2$.

Structural test: an example test suite (2)

```

let rec routeOf lid =
  function
  | []                                -> None           // c1
  | (lid', r) :: _ when lid=lid'     -> Some r          // c2
  | _ :: rest                        -> routeOf lid rest // c3

```

TestId	Input (lid/cat)	expected output
B	"l0" [("l0", r_0)]	Some r_0
C	"l4" [("l0", r_0)]	None
D	"l2" [("l0", r_0); ("l1", r_1); ("l2", r_2)]	Some r_2

where

- $r_i = [("f0", "a0"); \dots; ("fi", "ai")]$, for $i = 0, 1, 2$.

Each test case is kept "small" focusing only on the concerned Input Property

Structural test: an example test suite (2)

```

let rec routeOf lid =
  function
  | []                                -> None           // c1
  | (lid', r) :: _ when lid=lid'     -> Some r          // c2
  | _ :: rest                        -> routeOf lid rest // c3

```

TestId	Input (lid/cat)	expected output
B	"l0" [("l0", r_0)]	Some r_0
C	"l4" [("l0", r_0)]	None
D	"l2" [("l0", r_0); ("l1", r_1); ("l2", r_2)]	Some r_2

where

- $r_i = [("f0", "a0"); \dots; ("fi", "ai")]$, for $i = 0, 1, 2$.

Each test case is kept "small" focusing only on the concerned Input Property

Designing a Structural test is typically easier than designing a Functional test

Property-based test: A fundamental property (III)

```
type Route                = (Flight * Airport) list
type LuggageCatalogue = (Lid * Route) list

withFlight: Flight -> LuggageCatalogue -> Lid list
routeOf: Lid -> LuggageCatalogue -> Route option
```

Functional and Structural tests:

- test just a small number of (hopefully well-justified) single cases for each function

Property-based test: A fundamental property (III)

```
type Route           = (Flight * Airport) list
type LuggageCatalogue = (Lid * Route) list

withFlight: Flight -> LuggageCatalogue -> Lid list
routeOf: Lid -> LuggageCatalogue -> Route option
```

Functional and Structural tests:

- test just a small number of (hopefully well-justified) single cases for each function

Property-based test (PBT): tests having arguments

- Fundamental properties every input must satisfy
- Properties may express relationships between functions
- Validation using random samples

Property-based test: A fundamental property (III)

```

type Route                = (Flight * Airport) list
type LuggageCatalogue    = (Lid * Route) list

withFlight: Flight -> LuggageCatalogue -> Lid list
routeOf: Lid -> LuggageCatalogue -> Route option

```

Functional and Structural tests:

- test just a small number of (hopefully well-justified) single cases for each function

Property-based test (PBT): tests having arguments

- Fundamental properties every input must satisfy
- Properties may express relationships between functions
- Validation using random samples

Property(lc: LuggageCatalogue): for every lc
 for every lid occurring in lc::
 for every f occurring in routeOf lid lc:
lid is in withFlight f lc

A four-project solution is uploaded to the Material folder on Learn

- A class library project containing `inRoute`, `routeOf` and `withFlight`
- A test project containing structural tests for `routeOf`
- A test project containing functional tests for `withFlight`
- A test project containing a PBT for `routeOf` and `withFlight`

`xUnit` is used as testing tool

A four-project solution is uploaded to the Material folder on Learn

- A class library project containing `inRoute`, `routeOf` and `withFlight`
- A test project containing structural tests for `routeOf`
- A test project containing functional tests for `withFlight`
- A test project containing a PBT for `routeOf` and `withFlight`

`xUnit` is used as testing tool

It happens that `withFlight` is implemented by a programmer having weird ideas.

Run the tests in the folder `XTests` using the command:

```
...\src\XTests > dotnet test
```

where `X` is 'Functional', 'Structural', 'Property'.

Automating the tests

A four-project solution is uploaded to the Material folder on Learn

- A class library project containing `inRoute`, `routeOf` and `withFlight`
- A test project containing structural tests for `routeOf`
- A test project containing functional tests for `withFlight`
- A test project containing a PBT for `routeOf` and `withFlight`

`xUnit` is used as testing tool

It happens that `withFlight` is implemented by a programmer having weird ideas.

Run the tests in the folder `XTests` using the command:

```
...\src\XTests > dotnet test
```

where `X` is 'Functional', 'Structural', 'Property'.

Spot errors, correct the declaration of `withFlight` and rerun tests

- Functional and Structural test are complementary techniques

- Functional and Structural test are complementary techniques
- Test can show the presence of bugs not the absence

- Functional and Structural test are complementary techniques
- Test can show the presence of bugs not the absence
- Test suite descriptions: the basis for assessing test quality
 - well-designed descriptions increase confidence in programs

- Functional and Structural test are complementary techniques
- Test can show the presence of bugs not the absence
- Test suite descriptions: the basis for assessing test quality
 - well-designed descriptions increase confidence in programs
- Design a functional test suite during program development
 - sharpens the understanding of the problem

- Functional and Structural test are complementary techniques
- Test can show the presence of bugs not the absence
- Test suite descriptions: the basis for assessing test quality
 - well-designed descriptions increase confidence in programs
- Design a functional test suite during program development
 - sharpens the understanding of the problem
- PBT is a further supplement

- Functional and Structural test are complementary techniques
- Test can show the presence of bugs not the absence
- Test suite descriptions: the basis for assessing test quality
 - well-designed descriptions increase confidence in programs
- Design a functional test suite during program development
 - sharpens the understanding of the problem
- PBT is a further supplement
- Automate test.
 - Rerun test whenever the program is revised

- Functional and Structural test are complementary techniques
- Test can show the presence of bugs not the absence
- Test suite descriptions: the basis for assessing test quality
 - well-designed descriptions increase confidence in programs
- Design a functional test suite during program development
 - sharpens the understanding of the problem
- PBT is a further supplement
- Automate test.
 - Rerun test whenever the program is revised

The note:

- *Systematic software test*, by Peter Sestoft, 1998

is uploaded to the Material folder on Learn.