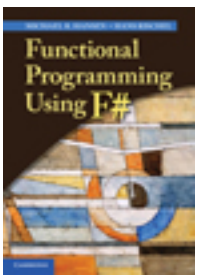


Functional Programming Using F#

Michael R. Hansen, Hans Rischel

Book DOI: http://dx.doi.org/10.1017/CBO9781139093996

Online ISBN: 9781139093996

Hardback ISBN: 9781107019027

Paperback ISBN: 9781107684065

Chapter

6 - Finite trees pp. 121-148

Chapter DOI: http://dx.doi.org/10.1017/CBO9781139093996.007

Cambridge University Press

6

Finite trees

This chapter is about trees, which are structures that may contain subcomponents of the same type. A list is an example of a tree. The list `1::[2;3;4]`, for example, contains a subcomponent `[2;3;4]` that is also a list. In this chapter we will introduce the concept of a tree through a variety of examples.

In F# we use a *recursive* type declaration to represent a set of values which are trees. The constructors of the type correspond to the rules for building trees, and patterns containing constructors are used when declaring functions on trees.

We motivate the use of finite trees and recursive types by a number of examples: Chinese boxes, symbolic differentiation, expression trees, search trees, file systems, trees with different kinds of nodes and electrical circuits.

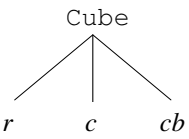
6.1 Chinese boxes

A *Chinese box* is a coloured cube that contains a coloured cube that ... that contains a coloured cube that contains nothing. More precisely, a Chinese box is either *Nothing* or a *Cube* characterised by its *side length*, *colour* and the *contained* Chinese box. This characterization can be considered as stating rules for generating Chinese boxes, and it is used in the following definition of Chinese boxes as *trees*:

The set Cbox of *Chinese boxes* can be represented as the set of trees generated by the rules:

Rule 1: The tree `Nothing` is in Cbox.

Rule 2: If r is a float number, if c is a colour, and if cb is in Cbox, then the tree:

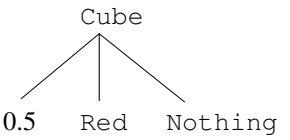


is also in Cbox.

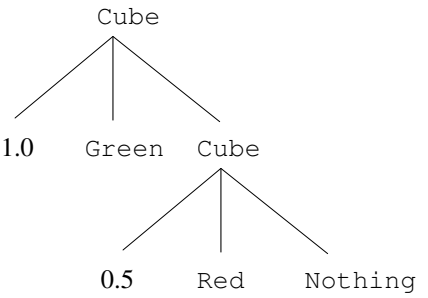
Rule 3: The set Cbox contains no other values than the trees generated by repeated use of Rule 1 and Rule 2.

The following example shows how this definition can be used to generate elements of Cbox.

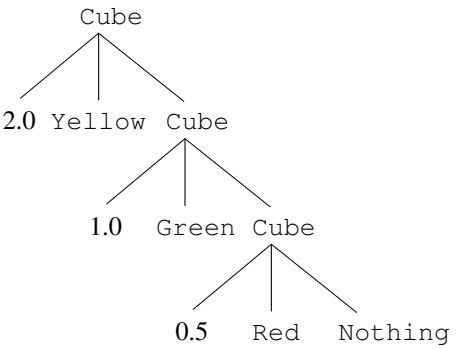
Step a: The void tree `Nothing` is a member of Cbox by Rule 1.
Step b: The following tree is a member of Cbox by Step a and Rule 2:



Step c: The following tree is a member of Cbox by Step b and Rule 2:



Step d: The following tree is a member of Cbox by Step c and Rule 2:



Type declaration

Using the following type `Colour`:

```
type Colour = Red | Blue | Green | Yellow | Purple;;
```

we declare a type `Cbox` representing the set of Chinese boxes as follows:

```
type Cbox = | Nothing                          // 1.
            | Cube of float * Colour * Cbox;; // 2.
```

The declaration is recursive, because the declared type `Cbox` occurs in the argument type of the constructor `Cube`. The constructors `Nothing` and `Cube` correspond to the above rules 1 and 2 for generating trees, so we can redo the above steps a through d with values of type `Cbox`:

Step a′: The constructor `Nothing` is a value of type `Cbox`.

Step b′: The value `Cube(0.5,Red,Nothing)` of type `Cbox` represents the tree generated in Step b.

Step c′: The value `Cube(1.0,Green,Cube(0.5,Red,Nothing))` of type `Cbox` represents the tree generated in Step c.

Step d′: The value:

```
Cube(2.0,Yellow,Cube(1.0,Green,Cube(0.5,Red,Nothing)))
```

of type `Cbox` represents the tree generated in Step d.

These examples show the relationship between trees and values of type `Cbox`, and we note the following statements where the last one follows from Rule 3 for generating trees:

- Different values of type `Cbox` represent different trees.
- Any tree is represented by a value of type `Cbox`.

Hence, a value of type `Cbox` is just a way of writing a tree instead of drawing it. F# does not draw trees when printing values of type `Cbox` – the interactive F# system prints the textual form of the value:

```
let cb1 = Cube(0.5, Red, Nothing);;
val cb1 : Cbox = Cube (0.5,Red,Nothing)

let cb2 = Cube(1.0, Green, cb1);;
val cb2 : Cbox = Cube (1.0,Green,Cube (0.5,Red,Nothing))

let cb3 = Cube(2.0, Yellow, cb2);;
val cb3 : Cbox = Cube (2.0,Yellow,Cube (1.0,Green,
                    Cube (0.5,Red,Nothing)))
```

Patterns

In Section 3.8 we have seen declarations containing patterns for tagged values. Constructors for trees can occur in patterns just like constructors for tagged values. An example of a *tree pattern* is `Cube(r,c,cb)`, containing identifiers `r`, `c` and `cb` for the components. This pattern denotes the tree in Figure 6.1.

This pattern will, for example, match the tree shown in Figure 6.2 corresponding to the value `Cube(1.0,Green,Cube(0.5,Red,Nothing))` with bindings

```
r  ↦  1.0
c  ↦  Green
cb ↦  Cube(0.5,Red,Nothing)
```

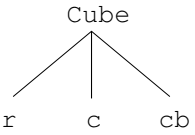


Figure 6.1 Tree for pattern `Cube(r,c,cb)`

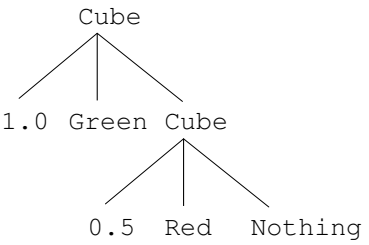


Figure 6.2 Tree for value `cb2`

where `cb` is bound to a value of type `cbox` corresponding to the tree shown in Step b on Page 122.

The *inductive* definition of the trees implies that any tree will either match the empty tree corresponding to the pattern:

```
Nothing
```

according to Rule 1 in the definition of trees, or the tree pattern for a cube in Figure 6.1 corresponding to:

```
Cube(r,c,cb)
```

according to Rule 2 in the definition of trees.

Function declarations

We give a declaration of the function:

```
count: Cbox -> int
```

such that the value of the expression: `count(`*cb*`)` is the number of cubes of the Chinese box *cb*:

```
let rec count = function
    | Nothing       -> 0
    | Cube(r,c,cb)  -> 1 + count cb;;
val count : Cbox -> int
```

The declaration divides into two cases, one with pattern `Nothing` and the other with pattern `Cube(r,c,cb)`. Thus, the declaration follows the inductive definition of Chinese boxes.

This function can be applied to the above values `cb2` and `cb3`:

```
count cb2 + count cb3;;
val it : int = 5
```

Invariant for Chinese boxes

A Chinese box must satisfy the invariant that the length of its sides is a positive floating-point number, which is larger than the side length of any cube it contains. The above four Chinese boxes in steps a to d satisfy this invariant, but using the generation process for trees one can construct the tree in Figure 6.3 that violates the invariant (i.e., it does not correspond to any Chinese box).

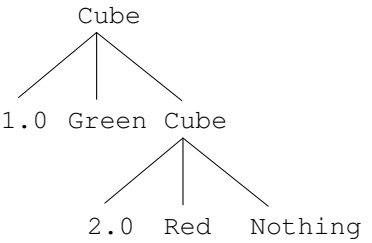


Figure 6.3 A tree violating the invariant

When declaring a function on Chinese boxes by the use of the type `Cbox` we must ensure that the function respects the invariant, that is, the function will only compute values of type `Cbox` satisfying the invariant when applied to values satisfying the invariant.

Insertion function

We can declare an insertion function on Chinese boxes:

```
insert: float * Colour * Cbox -> Cbox
```

The value of the expression `insert(`r`,`c`,`cb`)` is the Chinese box obtained from cb by inserting an extra cube with side length r and colour c at the proper place among the cubes in the box. The function `insert` is a partial function, that raises an exception in case the insertion would violate the invariant for Chinese boxes:

```
let rec insert(r,c,cb) =
  if r <= 0.0 then failwith "ChineseBox"
  else match cb with
       | Nothing          -> Cube(r,c,Nothing)
       | Cube(r1,c1,cb1) ->
            match compare r r1 with
            | t when t > 0 -> Cube(r,c,cb)
            | 0            -> failwith "ChineseBox"
            | _            -> Cube(r1,c1,insert(r,c,cb1));;
```

```
insert(2.0,Yellow,insert(1.0,Green,Nothing));;
val it : Cbox = Cube (2.0,Yellow,Cube (1.0,Green,Nothing))

insert(1.0,Green,insert(2.0,Yellow,Nothing));;
val it : Cbox = Cube (2.0,Yellow,Cube (1.0,Green,Nothing))

insert(1.0,Green,Cube(2.0,Yellow,Cube(1.0,Green,Nothing)));;
System.Exception: ChineseBox
Stopped due to error
```

Note, that any legal Chinese box can be generated from the box `Nothing` by repeated use of `insert`.

Other F# representations of Chinese Boxes

One may argue that the type `cbox` is unnecessarily complicated as Chinese boxes may simply be modelled using lists:

```
type Cbox = (float * Colour) list
```

This is, however, essentially the same as the above `Cbox` type of trees, as the list type is a special case of the general concept of recursive types (cf. Section 6.3).

One may also argue that it is strange to have a constructor `Nothing` denoting a non-existing Chinese box, and one might rather discard the empty box and divide the Chinese boxes into those consisting of a single cube and those consisting of multiple cubes, as expressed in the following declaration:

```
type Cbox1 = | Single of float * Colour
             | Multiple of float * Colour * Cbox1;;
```

Using this type, we get the following declarations of the functions `count` and `insert`:

```
let rec count1 = function
  | Single _          -> 1
  | Multiple(_,_,cb) -> 1 + count1 cb;;

let rec insert1 (r1,c1,cb2) =
  if r1 <= 0.0 then failwith "insert1: Chinese box"
  else match cb2 with
       | Single (r2,c2)       ->
         match compare r1 r2 with
         | t when t < 0 -> Multiple(r2,c2,Single(r1,c1))
         | 0            -> failwith "ChineseBox"
         |  _           -> Multiple(r1,c1,cb2)
       | Multiple (r2,c2,cb3) ->
         match compare r1 r2 with
         | t when t < 0 -> Multiple(r2,c2,insert1(r1,c1,cb3))
         | 0            -> failwith "ChineseBox"
         | _            -> Multiple(r1,c1,cb2);;
```

We have now suggested several representations for Chinese boxes. The preferable choice of representation will in general depend on which functions we have to define. The clumsy declaration of the `insert1` function contains repeated sub-expressions and this indicates that the first model for Chinese boxes with a `Nothing` value is to be preferred.

6.2 Symbolic differentiation

We want to construct a program for computing the derivative of a real function of one variable. The program should, for example, compute the derivative $\mathrm{f}'(x) = 2x \cdot \cos(x^2)$ of the function $\mathrm{f}(x) = \sin(x^2)$. The concept of *function* in F# cannot be used for this purpose, as such a function declaration just gives the means of computing values of the function. For example:

```
let f x = sin(x * x);;
val f : float -> float

f 2.0;;
val it : float = -0.7568024953
```

The differentiation is a manipulation of the *expression* denoting a function, so we need a *representation* of the *structure* of such expressions. This can be done using expression trees.

We restrict our attention to expressions constructed from real-valued constants and the variable x, using the arithmetic functions: addition, subtraction, multiplication and division, and the real functions sin, cos, log and exp. We use the symbols `Add`, `Sub`, `Mul` and `Div` to represent the arithmetic operators, and the symbols `Sin`, `Cos`, `Log` and `Exp` to represent the special functions. The expressions $\sin(x \cdot x)$ and $(\sin\ x) \cdot x$ will then be represented by the expression trees shown in Figure 6.4.

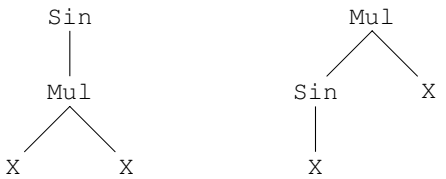


Figure 6.4 Trees for $\sin(x \cdot x)$ and $(\sin\ x) \cdot x$

The different order of the operators in these expressions is reflected in the trees: the tree for $\sin(x \cdot x)$ contains a sub-tree for the sub-expression $x \cdot x$, which again contains two sub-trees for the sub-expressions x and x, while the tree for $(\sin\ x) \cdot x$ contains sub-trees for the sub-expressions $\sin\ x$ and x.

The set of finite expression trees Fexpr is generated inductively by the following rules:

Rule 1: For every float number r, the tree for the constant r shown in Figure 6.5 is a member of `Fexpr`.

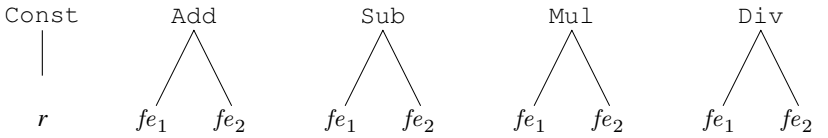


Figure 6.5 Tree for the constant r and trees for dyadic operators

Rule 2: The tree X is in Fexpr.

Rule 3: If fe_1 and fe_2 are in Fexpr, then the trees for the dyadic operators for addition, subtraction, multiplication and division shown in Figure 6.5 are members of Fexpr.

Rule 4: If fe is in Fexpr, then the trees for the special functions shown in Figure 6.6 are in Fexpr.

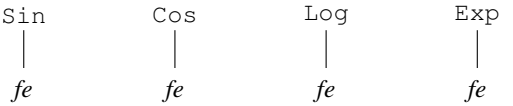


Figure 6.6 Trees for special functions

Rule 5: The set Fexpr contains no other values than the trees generated by rules 1. to 4.

Type declaration

Expression trees can be represented in F# by values of a recursively defined type. We get the following declaration of the type `Fexpr`:

```
type Fexpr = | Const of float
             | X
             | Add of Fexpr * Fexpr
             | Sub of Fexpr * Fexpr
             | Mul of Fexpr * Fexpr
             | Div of Fexpr * Fexpr
             | Sin of Fexpr
             | Cos of Fexpr
             | Log of Fexpr
             | Exp of Fexpr;;
```

For instance, the expression trees for $\sin(x \cdot x)$ and $(\sin\ x) \cdot x$ are represented by the values `Sin(Mul(X,X))` and `Mul(Sin X,X)` of type `Fexpr`.

Patterns

The following patterns correspond to values of type `Fexpr`:

```
Const r          X
Add(fe1,fe2)     Sub(fe1,fe2)     Mul(fe1,fe2)     Div(fe1,fe2)
Sin fe           Cos fe           Log fe           Exp fe
```

These patterns can be used in function declarations with a division into clauses according to the structure of expression trees.

$g'(x) = 0$ when $g(x) = c$, for $c \in \mathbb{R}$	Constant
$f'(x) = 1$ when $f(x) = x$	Identity
$(f(x) + g(x))' = f'(x) + g'(x)$	Addition
$(f(x) - g(x))' = f'(x) - g'(x)$	Subtraction
$(f(x) \cdot g(x))' = f'(x) \cdot g(x) + f(x) \cdot g'(x)$	Multiplication
$(f(x)/g(x))' = (f'(x) \cdot g(x) - f(x) \cdot g'(x))/(g(x))^2$	Division
$f(g(x))' = f'(g((x))) \cdot g'(x)$	Composition
$(\sin x)' = \cos x$ and $(\cos x)' = -\sin x$	Trigonometry
$(\log x)' = 1/x$	Logarithmic
$(\exp x)' = \exp x$	Exponential

Table 6.1 *Differentiation rules*

Function declaration

We are now in a position to declare a function

```
D: Fexpr -> Fexpr
```

such that `D`(fe) is a representation of the derivative with respect to x of the function represented by fe. The declaration for `D` has a clause for each constructor generating a value of type `Fexpr`, and each clause is a direct translation of the corresponding mathematical differentiation rule (see Tables 6.1 and 6.2):

```
let rec D = function
  | Const _     -> Const 0.0
  | X           -> Const 1.0
  | Add(fe,ge)  -> Add(D fe, D ge)
  | Sub(fe,ge)  -> Sub(D fe, D ge)
  | Mul(fe,ge)  -> Add(Mul(D fe, ge), Mul(fe, D ge))
  | Div(fe,ge)  -> Div(Sub(Mul(D fe,ge), Mul(fe,D ge)),
                       Mul(ge,ge))
  | Sin fe      -> Mul(Cos fe, D fe)
  | Cos fe      -> Mul(Const -1.0, Mul(Sin fe, D fe))
  | Log fe      -> Div(D fe, fe)
  | Exp fe      -> Mul(Exp fe, D fe);;
val D : Fexpr -> Fexpr
```

Table 6.2 *Differentiation function*

The following examples illustrate the use of the function:

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

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

Note, that these examples show results which can be reduced. For example, the above value of `D(Mul(Const 3.0, Exp X))` could be reduced to `Mul(Const 3.0, Exp X)` if a product with a zero factor was reduced to zero, and if adding zero or multiplying by one was absorbed. It is an interesting, non-trivial, task to declare a function that reduces expressions to a particular, simple form.

Conversion to textual representation

The following function: `toString: Fexpr -> string`, will produce a textual representation of a function expression:

```
let rec toString = function
    | Const x       -> string x
    | X             -> "x"
    | Add(fe1,fe2)  -> "(" + (toString fe1) + ")"
                       + " + " + "(" + (toString fe2) + ")"
    | Sub(fe1,fe2)  -> "(" + (toString fe1) + ")"
                       + " - " + "(" + (toString fe2) + ")"
    | Mul(fe1,fe2)  -> "(" + (toString fe1) + ")"
                       + " * " + "(" + (toString fe2) + ")"
    | Div(fe1,fe2)  -> "(" + (toString fe1) + ")"
                       + " / " + "(" + (toString fe2) + ")"
    | Sin fe        -> "sin(" + (toString fe) + ")"
    | Cos fe        -> "cos(" + (toString fe) + ")"
    | Log fe        -> "log(" + (toString fe) + ")"
    | Exp fe        -> "exp(" + (toString fe) + ")";;
val toString : Fexpr -> string

toString(Mul(Cos(Mul(X, X)),
             Add(Mul(Const 1.0, X), Mul(X, Const 1.0))));;
val it : string =
  "(cos((x) * (x))) * (((1) * (x)) + ((x) * (1)))"

toString(Add(Mul(X, Mul(X, X)) , Mul(X, X)));;
val it : string = "((x) * ((x) * (x))) + ((x) * (x))"
```

The function `toString` puts brackets around every operand of an operator and every argument of a function. It is possible to declare a better `toString` function that avoids unnecessary brackets. See Exercise 6.3.

6.3 Binary trees. Parameterized types

The constructors in a type declaration may have polymorphic types containing type variables. These type variables are *parameters* of the type, and written in angle brackets <...> just following the type constructor in the declaration.

An example of a type declaration with parameters is the type `BinTree<'a,'b>` of *binary trees* with leaves containing elements of type `'a` and nodes containing elements of type `'b`:

```
type BinTree<'a,'b> =
     | Leaf of 'a
     | Node of BinTree<'a,'b> * 'b * BinTree<'a,'b>;;
```

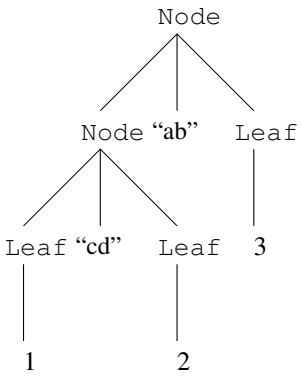


Figure 6.7 A tree t_1 of type `BinTree<int,string>`

The tree t_1 in Figure 6.7 corresponds to the value

```
let t1 = Node(Node(Leaf 1,"cd",Leaf 2),"ab",Leaf 3);;
```

of type `BinTree<int,string>`. The top node with element `"ab"` is called the *root* of the tree t_1 (trees are drawn upside-down in computer science with the root at the top and the leaves at the bottom). The trees t_2 and t_3 in Figure 6.8 corresponding to the values

```
let t2 = Node(Leaf 1,"cd",Leaf 2);;
let t3 = Leaf 3;;
```

are called the *left sub-tree* and *the right sub-tree* of t_1.

The type `BinTree` is *polymorphic* and allows polymorphic values like:

```
Node(Node(Leaf [],[],Leaf []),[],Leaf []);;
```

of type `BinTree<'a list,'b list>`.

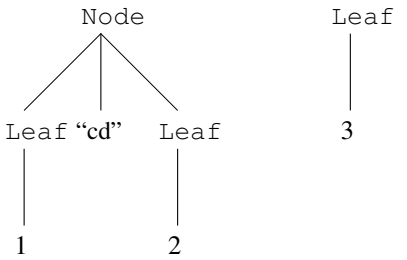


Figure 6.8 Left and right sub-tree t_2 and t_3 of the tree t_1

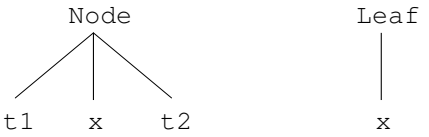


Figure 6.9 Trees for patterns `Node(t1,x,t2)` and `Leaf x`

The constructors `Node` and `Leaf` can be used in *patterns* like

```
Leaf x
Node(t1,x,t2)
```

corresponding to the pattern trees in Figure 6.9. Using this we may, for example, declare a function `depth` computing the depth of a binary tree:

```
let rec depth = function
    | Leaf _         -> 0
    | Node(t1,_,t2) -> 1 + max (depth t1) (depth t2);;
val depth : BinTree<'a,'b> -> int

depth t1;;
val it : int = 2
```

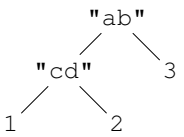


Figure 6.10 Simplified drawing of the tree `t1` in Figure 6.7

In the following we will often use simplified drawings of trees where the constructors have been left out and replaced by the value attached to the node. Such a simplified drawing of the tree `t1` in Figure 6.7 is shown in Figure 6.10.

6.4 Traversal of binary trees. Search trees

A *traversal* of a binary tree is a function *visiting the nodes* of the tree in a certain order. We may hence assume that the leafs do not carry any information corresponding to a simplified `BinTree` type:

```
type BinTree<'a> = | Leaf
                   | Node of BinTree<'a> * 'a * BinTree<'a>;;
```

A traversal of a tree of this type is then described by a function of type:

```
BinTree<'a> -> 'a list
```

We consider three kinds of traversals:

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.

The declarations look as follows:

```
let rec preOrder = function
    | Leaf          -> []
    | Node(tl,x,tr) -> x :: (preOrder tl) @ (preOrder tr);;
val preOrder : BinTree<'a> -> 'a list

let rec inOrder = function
    | Leaf          -> []
    | Node(tl,x,tr) -> (inOrder tl) @ [x] @ (inOrder tr);;
val inOrder : BinTree<'a> -> 'a list

let rec postOrder = function
    | Leaf          -> []
    | Node(tl,x,tr) -> (postOrder tl) @ (postOrder tr) @ [x];;
val postOrder : BinTree<'a> -> 'a list
```

We define values `t3` and `t4` of type `BinTree<int>` by:

```
let t3 = Node(Node(Leaf, -3, Leaf), 0, Node(Leaf, 2, Leaf));;
let t4 = Node(t3, 5, Node(Leaf, 7, Leaf));;
```

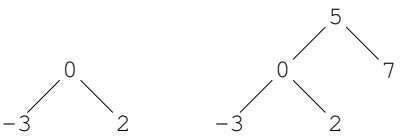


Figure 6.11 Trees corresponding to values `t3` and `t4`

Simplified drawings of the corresponding trees without constructors are shown in Figure 6.11. We will use such simplified drawings in the following.

Applying the traversal functions to `t4` we get:

```
preOrder t4;;
val it : int list = [5; 0; -3; 2; 7]

inOrder t4;;
val it : int list = [-3; 0; 2; 5; 7]

postOrder t4;;
val it : int list = [-3; 2; 0; 7; 5]
```

The reader should compare these lists with the figure and the oral descriptions of the traversal functions.

Traversal of binary trees can more generally be described by `fold` and `foldBack` functions defined such that the following holds:

```
preFold f e t     = List.fold f e (preOrder t)
preFoldBack f t e = List.foldBack f (preOrder t) e
```

and similar for in-order and post-order traversals. These functions should be declared to accumulate the values in the nodes while traversing the tree – without actually building the list. We show one of the declarations:

```
let rec postFoldBack f t e =
    match t with
    | Leaf          -> e
    | Node(tl,x,tr) ->
          let ex = f x e
          let er = postFoldBack f tr ex
          postFoldBack f tl er;;
val postFoldBack : ('a -> 'b -> 'b) -> BinTree<'a> -> 'b -> 'b

postFoldBack (fun x xs -> x::xs) t4 [];;
val it : int list = [-3; 2; 0; 7; 5]
```

The other declarations are left as exercises.

The type system of F# allows a great variety of tree types and there is no standard `Tree` library with a standard `tree` type. Functions like `inOrder` or `inFold` are hence defined individually according to need in each program using a tree type.

There are also imperative tree traversal functions where an imperative function is called whenever an element in the tree is visited, see Section 8.9.

Search trees

We restrict the type variable `'a` in our `BinTree` type to types with an ordering:

```
type BinTree<'a when 'a : comparison> =
     | Leaf
     | Node of BinTree<'a> * 'a * BinTree<'a>;;
```

A value of type `BinTree<'a>` is then called a *search tree* if it satisfies the following condition:

> Every node `Node`$(t_{\text{left}}, a, t_{\text{right}})$ satisfies:
> $a' < a$ for every value a' occurring in t_{left} and
> $a'' > a$ for every value a'' occurring in t_{right}.

This condition is called the *search tree invariant*. The trees `t3` and `t4` defined above and shown in Figure 6.11 satisfy this invariant and are hence search trees.

A search tree can be used to represent a *finite set* $\{a_0, a_1, \ldots, a_{n-1}\}$. This representation is particularly efficient when the tree is balanced (see discussion on Page 136).

A function `add` for adding a value to a search tree can be defined as follows:

```
let rec add x t =
    match t with
    | Leaf                    -> Node(Leaf,x,Leaf)
    | Node(tl,a,tr) when x<a -> Node(add x tl,a,tr)
    | Node(tl,a,tr) when x>a -> Node(tl,a,add x tr)
    | _                       -> t;;
val add: 'a -> BinTree<'a> -> BinTree<'a> when 'a: comparison
```

It builds a single-node tree when adding a value `x` to an empty tree. When adding to a non-empty tree with root `a` the value is added to the left sub-tree if `x` $<$ `a` and to the right sub-tree if `x` $>$ `a`. The tree is left unchanged if `x` $=$ `a` because the value `x` is then already member of the represented set.

Adding the value `4` to the search tree `t4`

```
let t5 = add 4 t4;;
val t5 : BinTree<int> =
 Node
  (Node(Node(Leaf,-3,Leaf),0,Node(Leaf,2,Node(Leaf,4,Leaf))),
   5,Node(Leaf,7,Leaf))
```

gives the tree in Figure 6.12.

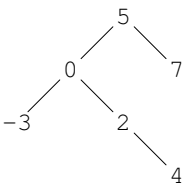


Figure 6.12 Search trees corresponding to the value `t5`

It follows by an inductive argument that an *in-order traversal* of a *search tree* will visit the elements in *ascending order* because the elements in the left sub-tree are smaller than the root element while the elements in the right sub-tree are larger – and this applies inductively to any sub-tree. We get for instance:

```
inOrder t5;;
val it : int list = [-3; 0; 2; 4; 5; 7]
```

An in-order traversal of a search tree will hence give a list where the elements in the nodes occur in ascending order.

A function `contains` for testing set membership can be declared by:

```
let rec contains x = function
    | Leaf                  -> false
    | Node(tl,a,_) when x<a -> contains x tl
    | Node(_,a,tr) when x>a -> contains x tr
    | _                     -> true;;
val contains : 'a -> BinTree<'a> -> bool when 'a : comparison

contains 4 t5;;
val it : bool = true
```

It uses the search tree property in only testing the left sub-tree if `x` $<$ the root node value and only the right sub-tree if `x` $>$ the root node value. The number of comparisons made when evaluating a function value: `contains` x t is hence *less or equal* to the *depth* of the tree t. It follows that the tree `t5` in Figure 6.12 is not an optimal representation of the set, because the set can be represented by the tree of depth 2 in Figure 6.13. The tree `t5` was created by the above `add` function, and it would hence require a more sophisticated `add` function to get the "balanced" tree in Figure 6.13 instead.

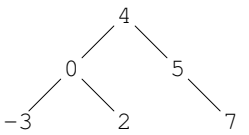


Figure 6.13 Search tree of depth `2` representing same set as `t5`

The number of nodes in a balanced tree with depth k is approximately 2^k and the depth of a balanced tree with n nodes is hence approximately $\log_2 n$. The `Set` and `Map` collections in the F# library use balanced search trees to get efficient implementations. A function like `Set.contains` will hence require circa $\log_2 n$ comparisons when used on a set with n elements. Searching a value (e.g., using `List.exists`) in a list of length n may require up to n comparisons. That makes a big difference for large n (e.g., $\log_2 n \approx 20$ when $n = 1000000$).

6.5 Expression trees

Tree representation of expressions is a common technique in compiler technology. This section gives a bit of the flavour of this technique. The subject is related to the function expression trees presented in Section 6.2.

We consider integer expressions of the form:

integer constant
identifier
− *expression*
expression + *expression*
expression − *expression*
expression ⋆ *expression*
`let` *identifier* = *expression* `in` *expression*
`(` *expression* `)`

They are represented by expression trees of the following type:

```
type ExprTree = | Const of int
                | Ident of string
                | Minus of ExprTree
                | Sum   of ExprTree * ExprTree
                | Diff  of ExprTree * ExprTree
                | Prod  of ExprTree * ExprTree
                | Let of string * ExprTree * ExprTree;;
```

such that, for example, the expression:

```
a * (-3 + (let x = 5 in x + a))
```

is represented by the value:

```
let et =
    Prod(Ident "a",
         Sum(Minus (Const 3),
             Let("x", Const 5, Sum(Ident "x", Ident "a"))));;
```

An expression is evaluated in an *environment* containing *bindings* of identifiers to values. An environment is represented by a value *env* of type `map<string,int>` containing entries with identifier and corresponding value. A `let` tree

$$\texttt{Let}(str, t_1, t_2)$$

is evaluated as follows in an environment *env*:

1. Evaluate t_1 to value v_1
2. Evaluate t_2 in the environment *env* extended with the binding of *str* to *v*.

An evaluation function

```
eval: ExprTree -> map<string,int> -> int
```

can now be defined recursively by dividing into cases according to the structure of the tree:

```
let rec eval t env =
    match t with
    | Const n      -> n
    | Ident s      -> Map.find s env
    | Minus t      -> - (eval t env)
    | Sum(t1,t2)   -> eval t1 env + eval t2 env
    | Diff(t1,t2)  -> eval t1 env - eval t2 env
    | Prod(t1,t2)  -> eval t1 env * eval t2 env
    | Let(s,t1,t2) -> let v1   = eval t1 env
                      let env1 = Map.add s v1 env
                      eval t2 env1;;
val eval : ExprTree -> Map<string,int> -> int
```

We may, for example, evaluate the above representation `et` of an expression in the environment `env` where the identifier `"a"` is bound to the value `-7`:

```
let env = Map.add "a" -7 Map.empty;;

eval et env;;
val it : int = 35
```

6.6 Trees with a variable number of sub-trees. Mutual recursion

Trees with a variable number of sub-trees are obtained by using a type where each node contains a (possibly empty) list of sub-trees. An example is the type:

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

Values of `ListTree` type represent trees where:

`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 $t_0, \ldots, t_{n-1}$

Such a tree is shown in Figure 6.14. 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]);;
```

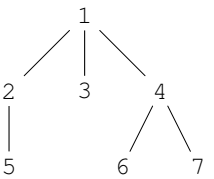


Figure 6.14 Tree represented by the value `t1`

Traversal of list trees

We consider two kinds of traversal of list-trees: depth-first and breadth-first traversal. These traversals correspond to the following order of the elements of the tree in Figure 6.14:

Depth-first:	1, 2, 5, 3, 4, 6, 7
Breadth-first:	1, 2, 3, 4, 5, 6, 7

In both cases we define a function to generate the list of nodes as well as `fold` and `foldBack` functions. The declarations involve functions on *lists* because the sub-trees of a node are organized as a list.

In the depth-first order we first visit the root node and then the nodes in each element of the list of immediate sub-trees: The function `depthFirst` generating a list of elements is declared using the library function `List.collect` (cf. Table 5.1) to apply `depthFirst` to each sub-tree in the list and afterwards collect the obtained lists into one list:

```
let rec depthFirst (Node(x,ts)) =
    x :: (List.collect depthFirst ts);;
val depthFirst : ListTree<'a> -> 'a list
```

The function `depthFirstFold f` is declared using `List.fold` to apply the function to each tree in the list of sub-trees and to transfer the accumulated value to the call on the next sub-tree:

```
let rec depthFirstFold f e (Node(x,ts)) =
    List.fold (depthFirstFold f) (f e x) ts;;
val depthFirstFold: ('a->'b->'a) -> 'a -> ListTree<'b> -> 'a

depthFirstFold (fun a x -> x::a) [] t1;;
val it : int list = [7; 6; 4; 3; 5; 2; 1]
```

The reader should appreciate this short and elegant combination of library functions.

The declaration of `depthFirstFoldBack` is left as an exercise to the reader (cf. Exercise 6.12).

In the breadth-first order we should first visit the root and then the roots of the immediate sub-trees and so on. This view of the problem does, unfortunately, not lead to any useful recursion because the remaining part becomes organized in an inconvenient list of lists of sub-trees.

A nice recursive pattern is instead obtained by constantly keeping track of the list `rest` of sub-trees where the nodes still remain to be visited. Using this idea on the tree in Figure 6.14 we get:

Visit	`rest`
1	`[t2; t3; t4]`
2	`[t3; t4; t5]`
3	`[t4; t5]`
4	`[t5; t6; t7]`
...	...

Each step in this scheme will do the following:

1. Remove the head element `t` of the list `rest`.
2. Get a new `rest` list by appending the list of immediate sub-trees of `t`.
3. Visit the root of `t`.

The traversal finishes when the list `rest` becomes empty.

This pattern is used in the following declarations where the argument of the auxiliary function `breadthFirstList` is a list corresponding to the `rest` list of sub-trees:

```
let rec breadthFirstList = function
    | []                    -> []
    | (Node(x,ts)) :: rest ->
          x :: breadthFirstList(rest@ts);;
val breadthFirstList : ListTree<'a> list -> 'a list

let breadthFirst t = breadthFirstList [t];;
val breadthFirst : ListTree<'a> -> 'a list

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

The declaration of `breadthFirstFoldBack` follows the same pattern:

```
let rec breadthFirstFoldBackList f ts e =
    match ts with
    | []    -> e
    | (Node(x,ts))::rest ->
          f x (breadthFirstFoldBackList f (rest@ts) e);;
val breadthFirstFoldBackList :
  ('a -> 'b -> 'b) -> ListTree<'a> list -> 'b -> 'b

let breadthFirstFoldBack f t e =
                 breadthFirstFoldBackList f [t] e;;
val breadthFirstFoldBack :
  ('a -> 'b -> 'b) -> ListTree<'a> -> 'b -> 'b

breadthFirstFoldBack (fun x a -> x::a) t1 [];;
val it : int list = [1; 2; 3; 4; 5; 6; 7]
```

The declaration of `breadthFirstFold` is left as an exercise to the reader (cf. Exercise 6.12).

There are also imperative versions of these tree traversals where an imperative function is called whenever a node is visited, see Section 8.9 for depth-first traversal and Section 8.13 for breadth-first traversal. The breadth-first traversal uses an imperative queue.

Example of list trees: File system

A file system is a list of named files and named directories where each directory contains another file system. Figure 6.15 shows a directory named d_1 with its associated file system.

The directory d_1 contains two files a_1 and a_4 and two directories d_2 and d_3. The directory d_2 contains a file a_2 and a directory d_3, and so on. Note that the same name may occur in different directories. This structure is an example of a tree with variable number of sub-trees.

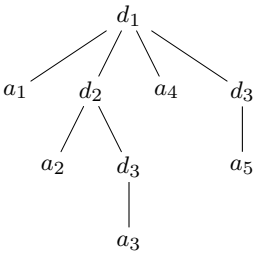


Figure 6.15 Directory with file system

Discarding the contents of files we represent a file system and its contents by two declarations:

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

The first declaration refers to a type `Element` which is declared in the second declaration. This "forward" reference to the type `Element` is allowed by the F# system because `Element` is declared in the second declaration using the keyword `and`. These two declarations constitute an example of *mutually recursive* type declarations, as the type `Element` occurs in the declaration of `FileSys` and the type `FileSys` occurs in the declaration of `Element`.

The directory shown in Figure 6.15 is represented by the value:

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

The declarations below yield functions `namesFileSys` and `namesElement` extracting a list of names of all files (including files in subdirectories) for file systems and elements, respectively:

```
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
```

The above function declarations are *mutually recursive* as the identifier `namesElement` occurs in the declaration of `namesFileSys` while the identifier `namesFileSys` occurs in the declaration of `namesElement`. Mutually recursive functions are declared using the keyword `and` to combine the individual function declarations.

The names of file and directories in the directory `d1` may now be extracted:

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

6.7 Electrical circuits

We consider electrical circuits built from *components* by *serial* or *parallel* composition. We represent a circuit by a value of the following type:

```
type Circuit<'a> = | Comp of 'a
                   | Ser  of Circuit<'a> * Circuit<'a>
                   | Par  of Circuit<'a> * Circuit<'a>;;
```

In Figure 6.16 we show a circuit containing three components with attached values 0.25, 1.0, and 1.5 together with the tree representing the circuit:

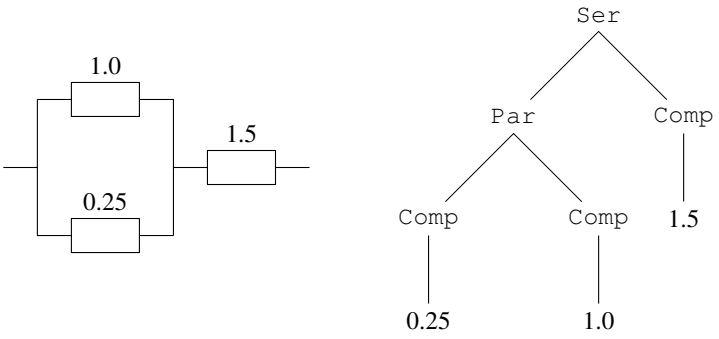


Figure 6.16 Circuit and corresponding tree

In F# the value is written `Ser(Par(Comp 0.25, Comp 1.0), Comp 1.5)`:

```
let cmp = Ser(Par(Comp 0.25, Comp 1.0), Comp 1.5);;
val cmp : Circuit<float>
        = Ser(Par(Comp 0.25,Comp 1.0),Comp 1.5)
```

Using this representation of circuits we can define a function `count` for computing the number of components in a circuit:

```
let rec count = function
    | Comp _     -> 1
    | Ser(c1,c2) -> count c1 + count c2
    | Par(c1,c2) -> count c1 + count c2;;
val count : Circuit<'a> -> int
```

For example:

```
count cmp;;
val it : int = 3
```

We consider now circuits consisting of resistances where the attached values are the resistances of the individual components. Suppose c_1 and c_2 are two circuits with resistances r_1 and r_2, respectively. The resistance of a serial combination of c_1 and c_2 is $r_1 + r_2$, and the resistance of a parallel combination of c_1 and c_2 is given by the formula:

$$\frac{1}{1/r_1 + 1/r_2}$$

Thus, a function `resistance` computing the resistance of a circuit can be declared by:

```
let rec resistance = function
    | Comp r     -> r
    | Ser(c1,c2) -> resistance c1 + resistance c2
    | Par(c1,c2) ->
          1.0 / (1.0/resistance c1 + 1.0/resistance c2);;
val resistance : Circuit<float> -> float
```

For example:

```
resistance cmp;;
val it : float = 1.7
```

Tree recursion

The functions `count` and `resistance` on circuits can be expressed using a generic higher-order function `circRec` for traversing circuits. This function must be parameterized with three functions c, s and p, where

c :	`'a -> 'b`	The value for a single component.
s :	`'b -> 'b -> 'b`	The combined value for two circuits connected in series.
p :	`'b -> 'b -> 'b`	The combined value for two circuits connected in parallel.

Note that s and p have the type `'b -> 'b -> 'b` because they operate on the values for two circuits. Thus, a general higher-order recursion function for circuits will have the type:

```
('a -> 'b) * ('b -> 'b -> 'b) * ('b -> 'b -> 'b)
   -> Circuit<'a> -> 'b
```

and the function is declared by:

```
let rec circRec (c,s,p) = function
    | Comp x     -> c x
    | Ser(c1,c2) ->
          s (circRec (c,s,p) c1) (circRec (c,s,p) c2)
    | Par(c1,c2) ->
          p (circRec (c,s,p) c1) (circRec (c,s,p) c2);;
```

The function `circRec` can, for example, be used to compute the number of components in a circuit by use of the following functions c, s, and p:

c	is	`fun _ -> 1`	Each component counts for 1.
s	is	`(+)`	The count for a serial composition is the sum of the counts.
p	is	`(+)`	The count for a parallel composition is the sum of the counts.

```
let count circ = circRec((fun _ -> 1), (+), (+)) circ : int;;
val count : Circuit<'a> -> int
```

The type `int` is required to resolve the overloaded plus operators.

```
count(Ser(Par(Comp 0.25, Comp 1.0), Comp 1.5));;
val it : int = 3
```

Suppose again that the value attached to every component in a circuit is the resistance of the component. Then the function `circRec` can be used to compute the resistance of a circuit by use of the following functions c, s, and p:

c is `fun r -> r`
The attached value is the resistance.

s is `(+)`
The resistance of a serial composition is the sum of the resistances.

p is `fun r1 r2 -> 1.0/(1.0/r1+1.0/r2)`
The resistance of a parallel composition is computed by this formula.

Using these functions for c, s and p we get:

```
let resistance =
    circRec(
        (fun r -> r),
        (+),
        (fun r1 r2 -> 1.0/(1.0/r1 + 1.0/r2)));;
val resistance : (Circuit<float> -> float)

resistance(Ser(Par(Comp 0.25, Comp 1.0), Comp 1.5));;
val it : float = 1.7
```

Summary

We have introduced the notion of finite trees and motivated this concept through a variety of examples. In F# a recursive type declaration is used to represent a set of values which are trees. The constructors of the type correspond to the rules for building trees, and patterns containing constructors are used when declaring functions on trees. We have also introduced the notions of parameterized types, and mutually recursive type and function declarations.

Exercises

6.1 Declare a function red of type `Fexpr -> Fexpr` to reduce expressions generated from the differentiation program in Section 6.2. For example, sub-expressions of form `Const 1.0` $* e$ can be reduced to e. (A solution is satisfactory if the expression becomes "nicer". It is difficult to design a reduce function so that all trivial sub-expressions are eliminated.)

6.2 Postfix form is a particular representation of arithmetic expressions where each operator is preceded by its operand(s), for example:

$(x + 7.0)$ has postfix form $x\ 7.0\ +$
$(x + 7.0) * (x - 5.0)$ has postfix form $x\ 7.0\ +\ x\ 5.0\ -\ *$

Declare an F# function with type `Fexpr -> string` computing the textual, postfix form of expression trees from Section 6.2.

6.3 Make a refined version of the `toString` function on Page 130 using the following conventions: A subtrahend, factor or dividend must be in brackets if it is an addition or subtraction. A divisor must be in brackets if it is an addition, subtraction, multiplication or division. The argument of a function must be in brackets unless it is a constant or the variable `x`. (Hint: use a set of mutually recursive declarations.)

6.4 Consider binary trees of type `BinTree<'a,'b>` as defined in Section 6.3. Declare functions

1. `leafVals: BinTree<'a,'b> -> Set<'a>` such that `leafVals` t is the set of values occurring the leaves of t,
2. `nodeVals: BinTree<'a,'b> -> Set<'b>` such that `nodeVals` t is the set of values occurring the nodes of t, and
3. `vals: BinTree<'a,'b> -> Set<'a>*Set<'b>` such that `vals` $t = (ls, ns)$, where ls is the set of values occurring the leaves of t and ns is the set of values occurring the nodes of t

6.5 An ancestor tree contains the name of a person and of some of the ancestors of this person. We define the type `AncTree` by:

```
type AncTree = | Unspec
               | Info of AncTree * string * AncTree;;
```

The left sub-tree is the ancestor tree of the farther while the right sub-tree is the ancestor tree of the mother. Write a value of type `ancTree` with at least 5 nodes and make a drawing of the corresponding tree.

Declare functions `maleAnc` and `femaleAnc` to compute the list of names of male and female ancestors of a person in an ancestor tree.

6.6 Consider search trees of type `BinTree<'a>` as defined in Section 6.4. Declare an F# function that can delete an element in such a tree. Hint: Make use of an auxiliary function that deletes the smallest element in a non-empty search tree (and returns that value).

6.7 1. Define a type to represent formulas in propositional logic. A proposition is either an atom given by its name which is a string, or a composite proposition built from atoms using the operators for negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$).
2. A proposition is in *negation normal form* if the negation operator only appears as applied directly to atoms. Write an F# function transforming a proposition into an equivalent proposition in negation normal form, using the de Morgan laws:

$$\begin{aligned} \neg(p \wedge q) &\Leftrightarrow (\neg p) \vee (\neg q) \\ \neg(p \vee q) &\Leftrightarrow (\neg p) \wedge (\neg q) \end{aligned}$$

and the law: $\neg(\neg p) \Leftrightarrow p$.

3. A *literal* is an atom or its negation. A proposition is in *conjunctive normal form* if it is a conjunction of propositions, where each conjunct (i.e., proposition in the conjunction) is a disjunction of literals. Write an F# function that transforms a proposition into an equivalent proposition in conjunctive normal form, using the above result and the laws:

$$\begin{array}{rcl} p \vee (q \wedge r) & \Leftrightarrow & (p \vee q) \wedge (p \vee r) \\ (p \wedge q) \vee r & \Leftrightarrow & (p \vee r) \wedge (q \vee r) \end{array}$$

4. A proposition is a *tautology* if it has truth value *true* for any assignment of truth values to the atoms. A disjunction of literals is a tautology exactly when it contains the atom as well as the negated atom for some name occurring in the disjunction. A conjunction is a tautology precisely when each conjunct is a tautology. Write a tautology checker in F#, that is, an F# function which determines whether a proposition is a tautology or not.

6.8 We consider a simple calculator with instructions for addition, subtraction, multiplication and division of floats, and the functions: sin, cos, log and exp.

The *instruction set* of the calculator is modelled by the following F# `type`:

```
type Instruction = | ADD | SUB | MULT | DIV | SIN
                   | COS | LOG | EXP  | PUSH of float
```

The calculator is a *stack machine*, where a *stack* is a list of floats.

The *execution* of an instruction maps a stack to a new stack:

The execution of `ADD` with stack $\boxed{a\ b\ c\ \cdots}$ yields a new stack: $\boxed{(b+a)\ c\ \cdots}$, where the top two elements a and b on the stack have been replaced by the single element $(b + a)$. Similarly with regard to the instructions, `SUB`, `MULT` and `DIV`, which all work on the *top* two elements of the stack.

The execution of one of the instructions `SIN`, `COS`, `LOG` and `EXP` applies the corresponding function to the top element of the stack. For example, the execution of `LOG` with stack $\boxed{a\ b\ c\ \cdots}$ yields the new stack: $\boxed{\log(a)\ b\ c\ \cdots}$.

The execution of `PUSH` r with the stack $\boxed{a\ b\ c\ \cdots}$ *pushes* r on top of the stack, that is, the new stack is: $\boxed{r\ a\ b\ c\ \cdots}$.

1. Declare a type `Stack` for representing the stack, and declare an F# function to interpret the execution of a single instruction:

   ```
   intpInstr: Stack -> Instruction -> Stack
   ```

2. A *program* for the calculator is a list of instructions $[i_1, i_2, \ldots, i_n]$. A program is *executed* by executing the instructions $i_1, i_2, \ldots, i_n$ one after the other, in that order, starting with an empty stack. The result of the execution is the top value of the stack when all instructions have been executed.

   Declare an F# function to interpret the execution of a program:

   ```
   intpProg: Instruction list -> float
   ```

3. Declare an F# function

```
trans: Fexpr * float -> Instruction list
```

where `Fexpr` is the type for expression trees declared in Section 6.2. The value of the expression `trans`(fe, x) is a program prg such that `intpProg`(prg) gives the float value of fe when `X` has the value x. Hint: The instruction list can be obtained from the postfix form of the expression. (See Exercise 6.2.)

6.9 A company consists of departments with sub-departments, which again can have sub-departments, and so on. (The company can also be considered as a department.)

1. Assume that each department has a name and a (possibly empty) list of sub-departments. Declare an F# type `Department`.
2. Extend this type so that each department has its own gross income.
3. Declare a function to extract a list of pairs (*department name*, *gross income*), for all departments.
4. Declare a function to extract the total income for a given department by adding up its gross income, including the income of its sub-departments.
5. Declare a function to extract a list of pairs (*department name*, *total income*) for all departments.
6. Declare a function `format` of type `Department -> string`, which can be used to get a textual form of a department such that names of sub-departments will occur suitably indented (e.g., with four spaces) on separate lines. (Use `printf` to print out the result. Do not use `printf` in the declaration of `format`.)

6.10 Consider expression trees of type `ExprTree` declared in Section 6.5. Extend the type with an if-then-else expression of the form: `if` b `then` e_1 `else` e_2, where b is a boolean expression and e_1 and e_2 are expressions. An example could be:

```
if a*3>b+c && a>0 then c+d else e
```

Furthermore, extend the declaration of the `eval` function accordingly. Hint: make use of a mutually recursive type and function declarations.

6.11 Write the steps of the evaluation of the expression:

```
depthFirstFold (fun a x -> x::a) [] t1
```

See Page 139.

6.12 Declare the functions `depthFirstFold` and `breadthFirstFoldBack` on list trees, cf. Section 6.6.