Record Types

```
{- name="Jones", salary=20300, age=26};

val it =
{age = 26, name = "Jones", salary = 20300}
: {age : int, name : string, salary : int}


- {1="Jones", 2=20300,3=26};

> val it = ("Jones", 20300, 26)
           : string * int * int
```

Record Pattern Matching

```
- val emp1 =
{name="Jones", salary=20300, age=26};

> val emp1 =
{age = 26, name = "Jones", salary = 20300}
: {age : int, name : string, salary : int}

- val {name=n1,salary=s1,age=a1}= emp1;

> val n1 = "Jones" : string
  val s1 = 20300 : int
  val a1 = 26 : int

- val {name=n1,salary=s1,...} = emp1;

> val n1 = "Jones" : string
  val s1 = 20300 : int

- val {name,age,...} = emp1;

> val name = "Jones" : string
  val age = 26 : int
```

Record Types

```
type employee = {name: string,
                 salary: int,
                 age: int};
> type employee
```

```
fun tax (e: employee) =
        real(#salary e)*0.22
```

Or,

```
fun tax ({salary,...}: employee) =
       real(salary)*0.22;
```

Enumerated Types

Consider the King and his court:

```
datatype degree = Duke
                | Marquis
                | Earl
                | Viscount
                | Baron;

datatype person =
             King
           | Peer of degree*string*int
           | Knight of string
           | Peasant of string;
```

All constructors are distinct.

Functions on Datatypes

```
[King,
 Peer(Duke, "Gloucester", 5),
 Knight "Gawain",
 Peasant "Jack Cade"];

val it  = ... : person list
```

```
fun superior (King, Peer _) = true
  | superior (King, Knight _) = true
  | superior (King, Peasant _) = true
  | superior (Peer _,Knight _) = true
  | superior (Peer _, Peasant _) = true
  | superior (Knight _, Peasant _) = true
  | superior _ = false;
```

Exceptions

Exceptions are raised when there is no matching pattern, when an overflow occurs, when a subscript is out of range, or some other run-time error occurs.

Exceptions can also be explicitly raised.

```
exception Failure;
exception BadVal of Int;
```

```
raise Failure
raise (BadVal 5)
```

E `handle` P_1 `=>` E_1 `|` ... `|` P_n `=>` E_n

Recursive Datatypes

The built-in type operator of lists might be defined as follows:

```
infix :: ;

datatype 'a list = nil
                 | :: of 'a * 'a list;
```

Binary Trees:

```
datatype 'a tree =
            Lf
          | Br of 'a * 'a tree * 'a tree;


Br(1, Br(2, Br(4, Lf, Lf),
            Br(5, Lf, Lf)),
      Br(3, Lf, Lf))
```

Functions on Trees

Counting the number of branch nodes

```
fun count Lf              = 0
  | count (Br(v,t1,t2)) =
            1+count(t1)+count(t2);
```

```
val count = fn : 'a tree -> int
```

Depth of a tree

```
fun depth Lf              = 0
  | depth (Br(v,t1,t2)) =
           1+Int.max(depth t1, depth t2);
```

```
val depth = fn : 'a tree -> int
```

Listing a Tree

Three different ways to list the data elements of a tree

Pre-Order

```
fun preorder Lf            = []
  | preorder (Br(v,t1,t2))=
     [v] @ preorder t1 @ preorder t2;
```

In-Order

```
fun inorder Lf             = []
  | inorder (Br(v,t1,t2))=
     inorder t1 @ [v] @ inorder t2;
```

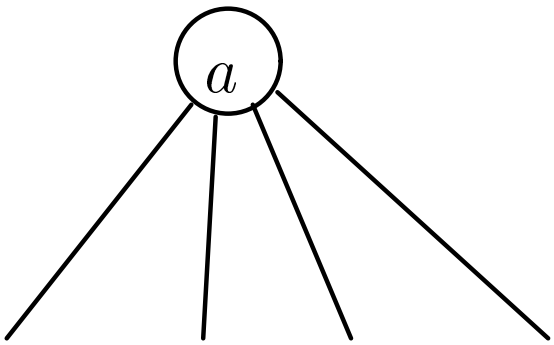
Post-Order

```
fun postorder Lf           = []
  | postorder (Br(v,t1,t2))=
     postorder t1 @ postorder t2 @ [v];
```

Multi-Branching Trees

To define a datatype of a tree where each node can have any number of children

```
datatype 'a mtree =
        Branch of 'a * ('a mtree) list;
```



To recursively define functions, we can use `map`.

```
fun double (Branch(k,ts)) =
           Branch(2*k, map double ts);
```

Arrays

Arrays in an imperative language designate a contiguous block of memory locations.

An array can be updated in place.

```
A[k] := x
```

There are no such arrays in a purely functional language.

A functional array can be thought of as a function from a finite set of integers.

$$A : \{1, \ldots, k\} \rightarrow \alpha$$

We require an update operation:

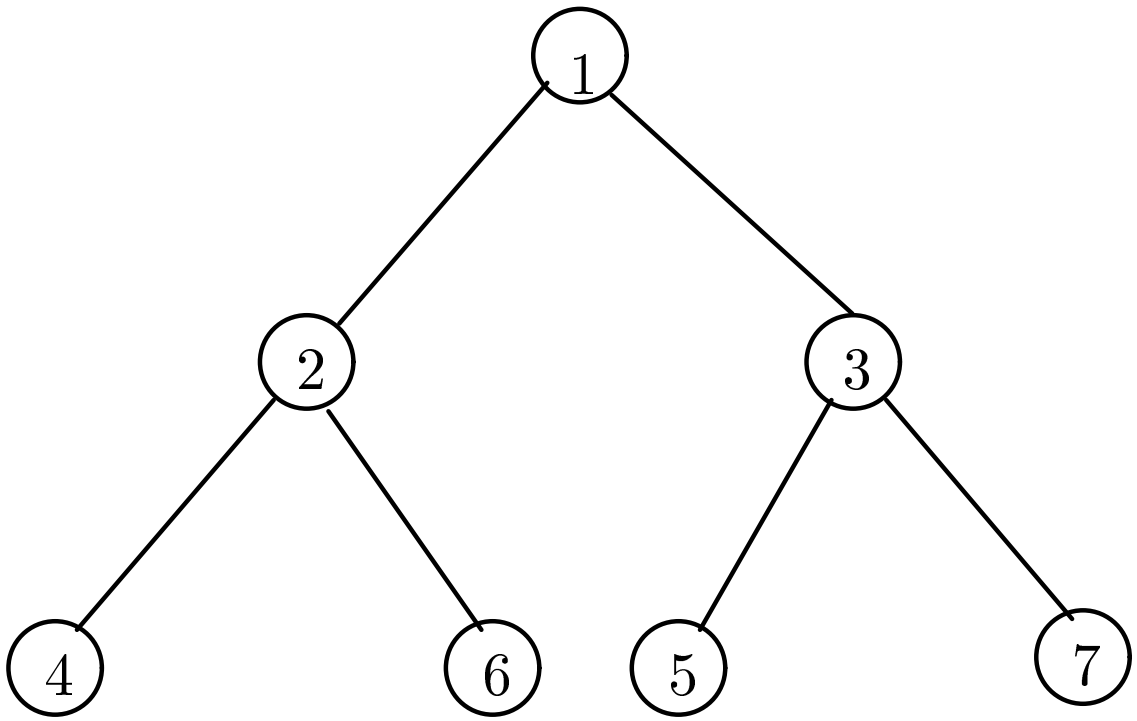
```
B = update(A, k, x)
```

This creates a new copy of the array, which is the same as A except for the value at k.

Arrays as Binary Trees

A tree whose depth is equal to the maximum number of bits in a subscript

For a subscript k, take its binary representation, and at each level, follow the left branch if the bit is 0 and the right branch if it is 1.



Lookup

```
exception Array;    (*out of range*)

fun asub (Lf, _)            = raise Array
  | asub (Br(v,t1,t2), k) =
       if      k = 1
                then v
       else if k mod 2 = 0
                then asub(t1, k div 2)
                else asub(t2, k div 2);
```

```
val asub = fn : 'a tree * int -> 'a
```

For an array A and integer k `asub(A,k)` gives the value of $A[k]$.

Update

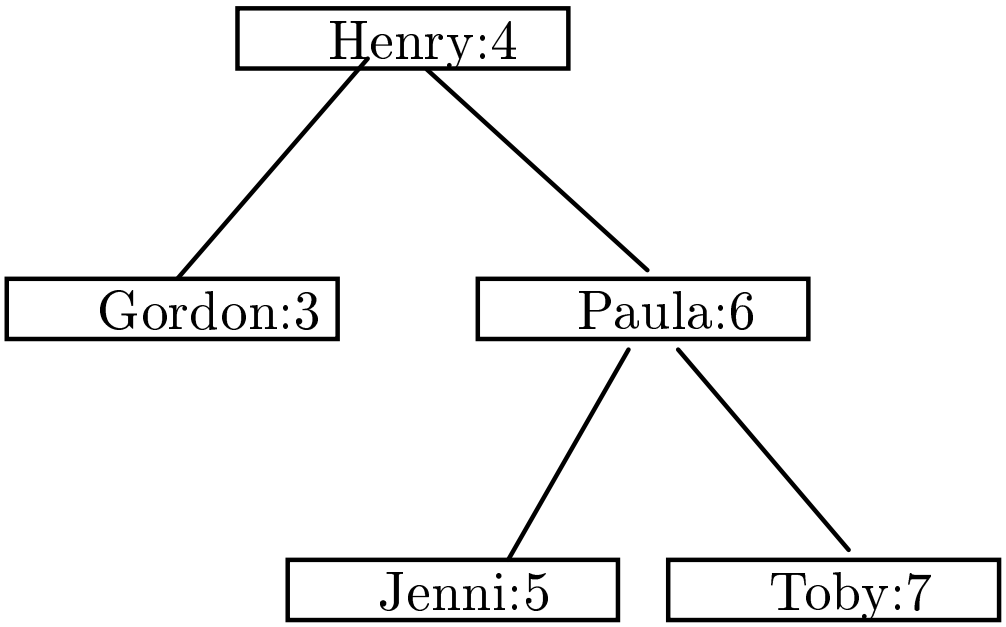
```
fun aupdate (Lf, k, w)           =
     if k = 1 then Br(w, Lf, Lf)
              else raise Array
  | aupdate (Br(v,t1,t2), k, w) =
     if       k = 1
      then
       Br(w, t1, t2)
     else if k mod 2 = 0
      then
       Br(v,aupdate(t1,k div 2,w),t2)
     else
      Br(v,t1,aupdate(t2,k div 2,w));


val aupdate =
 fn : 'a tree * int * 'a -> 'a tree
```

Binary Tree Directories

A *Directory* associates *values* with *keys*.

If a directory is implemented as a list, each update and each lookup takes $O(n)$ operations *on the average*.

When the directory is implemented as a binary search tree, the operations take $O(\log n)$ on the average.



Search Tree Lookup

```
exception Bsearch of string;

fun blookup (Lf, b) = raise Bsearch(b)

  | blookup (Br((a,x), t1, t2), b)  =
     if       b<a then blookup(t1, b)
     else if a<b then blookup(t2, b)
     else x;


exn Bsearch = fn : string -> exn

val blookup =
fn : (string * 'a) tree * string -> 'a
```

Update

```
fun bupdate (Lf, k:string, v) =
      Br((k,v), Lf, Lf)
  | bupdate (Br((a,x),t1,t2), k, v) =
      if       k < a
        then Br((a,x),bupdate(t1,k,v),t2)
      else if a < k
        then Br((a,x),t1,bupdate(t2,k,v))
      else   Br((k,v),t1,t2);     (*a=k*)
```

```
val bupdate =
fn : (string * 'a) tree * string * 'a
     -> (string * 'a) tree
```

Higher-Order Functions

A higher order function (also called a *functional*) is a function that either takes a function as an argument, or yields a function as a result.

Higher order functions are a key feature that distinguishes functional from imperative programming.

- partial evaluation
- general purpose functionals (such as `map`)
- sequences or infinite lists
- search strategies

Function Values

The expression

fn x => E

is a function-valued expression.

Pattern-matching can be used:

fn P_1 => E_1 | ... | P_n => E_n

The declaration

```
val double = fn n=>n*n;
```

is the same as:

```
fun double n = n*n;
```

if E then E_1 else E_2

is defined as:

```
(fn true => E1 | false => E2) E
```

Curried Functions

Every function in ML takes *only one* argument.

For functions that require two arguments, we can use pairs:

```
- op+ (5,4);
> val it = 9 : int
- op+;
> val it = fn : int * int -> int
```

Or we can define:

```
- fun plus m n = m+n;
> val plus = fn : int -> int -> int
```

This is the *curried* version of `plus`

Partial evaluation:

```
- map (plus 4) [2, 3, 4];
> val it = [6, 7, 8] : int list
```