Higher order functions

```
listsum' :: [Integer] -> Integer
listsum' []          = 0
listsum' (x:l)       = x + (listsum' l)

listsum :: [Integer] -> Integer
listsum = foldl (+) 0

foldl :: (a -> b -> a) -> a -> [b] -> a
```

(defined in standard prelude)

```
foldl f z []         = z
foldl f z (x:xs)     = foldl f (f z x) xs
```

Higher order functions

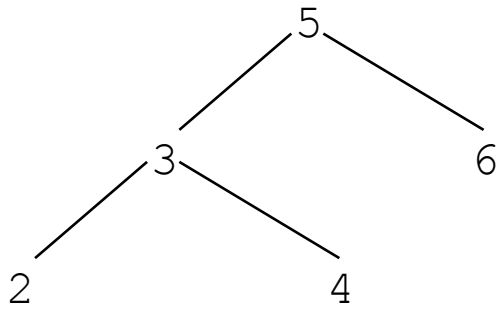
- Higher order functions are simply functions that:
  - Accept functions as parameters
  - Return functions
  - (or both)
- We've already done it in lisp
  - Especially with the lambda functions
- You can even get the same basic effect in C with function ptrs

Data types

Data types are not defined by their representation, but in terms of constructor operations (cf. ML).

```
data Tree a = Leaf a | Node a (Tree a)(Tree a)
```

```
Node 5 (Node 3 (Leaf 2) (Leaf 4)) (Leaf 6) :: Tree Integer
```



Data types and higher order functions

```
data IntTree   = Empty | Node Int IntTree IntTree

sumTree :: IntTree -> Int
sumTree Empty          = 0
sumTree (Node n t1 t2) = n + sumTree t1 + sumTree t2

depth :: IntTree -> Int
depth Empty            = 0
depth (Node n t1 t2)   = 1 + max (depth t1) (depth t2)

occurs :: IntTree -> Int -> Int
occurs Empty x         = 0
occurs (Node n t1 t2) x
   | n==x              = 1 + occurs t1 x + occurs t2 x
   | otherwise         =     occurs t1 x + occurs t2 x

inTree :: Int -> IntTree -> Bool
inTree x Empty           = False
inTree x (Node n t1 t2) = n==x || x `inTree` t1 || x `inTree` t2
```

Data types and higher order functions

Traversals as higher-order function:

```
foldIntTree :: (Int -> a -> a -> a) -> a -> IntTree -> a
foldIntTree f x Empty        = x
foldIntTree f x (Node n t1 t2) = f n (foldIntTree f x t1)
   (foldIntTree f x t2)

sumTree          = foldIntTree (\n m p -> n + m + p) 0
depth            = foldIntTree (\_ m p -> 1 + max m p) 0
occurs t x       = foldIntTree (\n m p -> (if n==x then 1 else 0) + m + p) 0 t
inTree x         = foldIntTree (\n b1 b2 -> n==x || b1 || b2) False
```

List comprehension

A *list comprehension* has the form `[ e | q1, …, qn ]`, where the q_i qualifiers are either

- *generators* of the form `p <- l`, where `p` is a pattern of type `t` and `l` is an expression of type `[t]`
- *guards,* which are arbitrary expressions of type `Bool`
- *local bindings* that provide new definitions for use in the generated expression `e` or subsequent guards and generators.

```
[ x |  xs <- [[(1,2),(3,4)],[(3,5),(3,2)]],
   (3,x) <- xs,
   even x ]
```

yields the list `[4,2]`.

Lazy evaluation

- In Java, ML and Lisp, parameters are called by value.
- lazy evaluation = parameters remain unevaluated until they are really needed (similar to call by name).

```
undef :: Integer
undef = undef
```

`fst (2, undef)` yields `2` and
`snd (2, undef)` does not terminate.

Example (infinite list)

```
primes :: [Integer]
primes = sieve [2.. ]

sieve (p:x) = p : sieve [ n | n <- x, n `mod` p > 0 ]

prime :: Int -> Integer
prime n = primes !! (n-1)

prime 1000

7919 :: Integer
```

Be careful:

- `2 `elem` primes` terminates (result true) but
- `4 `elem` primes` does not terminate.

Classes

- are used for overloading of function symbols.
- are predicates on types.

Example (Standard prelude):

```
class Eq a where
  (==), (/=)  :: a -> a-> Bool
  -- Minimal complete definition:
  --   (==) or (/=)
  x /= y             = not (x == y)
  x == y             = not (x /= y)
```

Classes (cont'd)

A type class or simply a `class` defines a collection of types over which specific functions are defined.

Members of a `class` are called instances. Built-in instances of `Eq` include the base types `Int`, `Float`, `Bool`, `Char`, tuples and lists built from types which are themselves instances of `Eq`, e.g., `(Int,Bool)` and `[[Char]]`.

```
elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys) = (x == y) || elem x ys
```

Instances of classes

- An instance of a class is a type together with declarations of the functions required by the class definition.

Example (Standard prelude):

```
instance Eq Char where
  c == c'     =      fromEnum c == fromEnum c'
```

Context of a function:

```
contained :: Eq a => a -> Tree a -> Bool
   ^^^^
```

Context: type `a` has to be an instance of class `Eq`.

Instances of classes

Examples:

```
instance Eq Bool where
   True  == True  = True
   False == False = True
   _     == _     = False

instance Eq a => Eq [a] where
    []     == []     =  True
    (x:xs) == (y:ys) =  x==y && xs==ys
    _      == _      =  False

instance (Eq a, Eq b) => Eq (a,b) where
   (x,y) == (z,w) = x==z && y==w
```

Derived instances

- Automatically generated instance
- Body of the instance declaration is derived syntactically from the definition of the type
- Possible for the classes `Eq`, `Ord`, `Enum`, `Bounded`, `Show` or `Read`

Example:

```
data Tree a = Leaf a | Node a (Tree a)(Tree a) deriving Eq
```

will generate

```
instance Eq a => Eq (Tree a) where
   (Leaf x)        == (Leaf y)            = x == y
   (Node x s1 s2) == (Node y t1 t2)       =     x == y
                                            && s1 == t1
                                            && s2 == t2
```

Derived classes

To be ordered, a type must carry operations >, >= and so on, as well as the

equality operations.

```
class (Eq a) => Ord a where
    compare                 :: a -> a -> Ordering
    (<), (<=), (>=), (>)    :: a -> a -> Bool
    max, min                :: a -> a -> a
```

Prolog

- PROgramming in LOGic
- goal-oriented: define *what* the problem is, not a detailed solution of *how* to solve it
  - list of subgoals: satisfying the subgoals results in achieving the goal
  - alternative subgoals and backtracking
- declarative meaning: objects and their relationships
- procedural meaning: how and in what order relationships are evaluated
- efficiency concerns

Prolog - Overview

- facts and rules
  - facts are always true
  - rules state that a statement is true if certain goals and subgoals are satisfied
- questions
- clauses
  - facts: head, no body
  - questions: body, no head
  - rules: head and body

Prolog – Royal Family

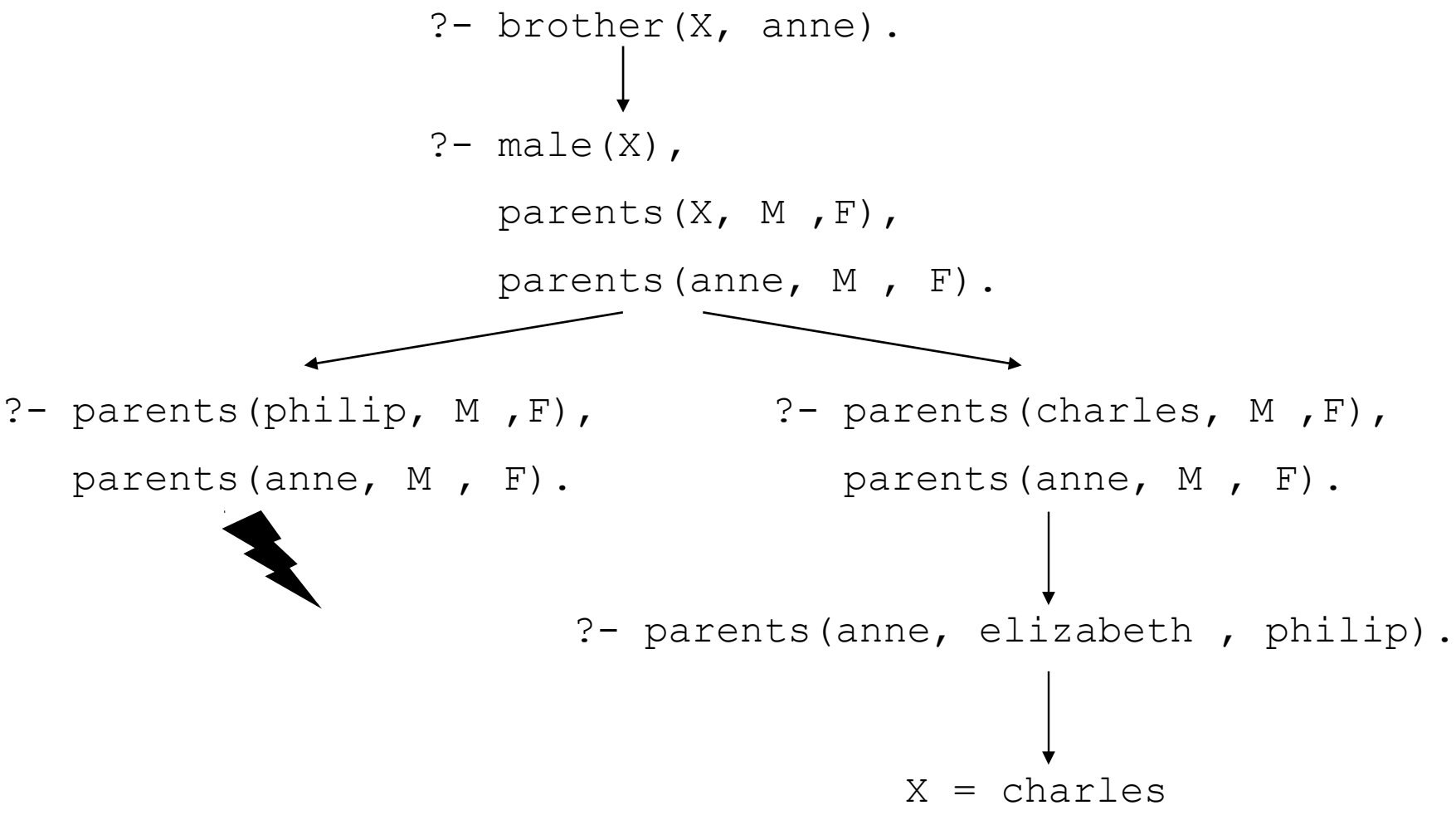
```
male(philip).
female(elizabeth).
male(charles).
female(anne).
male(andrew).
male(edward).
female(diana).
male(william).
male(harry).
parents(charles, elizabeth, philip).
parents(anne, elizabeth, philip).
parents(andrew, elizabeth, philip).
parents(edward, elizabeth, philip).
parents(william, diana, charles).
parents(harry, diana, charles).
```

Prolog – Royal Family (cont.)

```
brother(X, Y) :- male(X),
                 parents(X, M , F),
                 parents(Y, M , F).

?- brother(edward, anne).
   (X ← edward, Y ← anne)
⇒ male(edward), parents(edward, M, F), parents(anne, M, F).
⇒ parents(edward, M, F), parents(anne, M, F).
   (M ← elizabeth, F ← philip)
⇒ parents(edward, elizabeth, philip),
  parents(anne, elizabeth, philip).
```

Prolog – Royal Family (cont.)



Prolog – Example

```
mother(jane, george).
father(john, george).
brother(bill, john).

parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
uncle(Z, Y) :- parent(P, Y), brother(Z, P).
```

Prolog – Example (cont.)