

- An inductive proof for a fact $P(n)$, for all $n \geq \alpha$ consists in two steps:
 - Proof of the **base case** $P(\alpha)$, and
 - The **inductive step**: assume that $P(n - 1)$ is true and show that $P(n)$ is also true.

Example Proof that all the crows have the same colour: For all sets C of crows, $|C| \geq 1$, it holds that all the crows in set C are identical in colour.

- Base case, $|C| = 1$: immediate.
- For a set of crows C , $|C| = n$, remove a crow for the set; the remaining (a set of size $n - 1$) have the same colour by inductive assumption. Repeat by removing other crow. The desired property follows.

Note. According to the Webster's Revised Unabridged Dictionary, crow is "A bird, **usually** black, of the genus *Corvus* [...]"

EXAMPLE: LISTS AS SETS

- Membership ($x \in A$):
- Union ($A \cup B$), intersection ($A \cap B$), difference ($A \setminus B$):

- Constructor: no recursion. `makeSet x = [x]`

- The same process is used for building recursive functions: One should provide the **base case(s)** and the **recursive definition(s)**:
 - To write a function $f :: Integer \rightarrow Integer$, write the **base case** (definition for $f\ 0$) and the **inductive case** (use $f\ (n - 1)$ to write a definition for $f\ n$).

Example Computing the factorial:

- * Base case: `fact 0 = 1`
- * Induction step: `fact n = n * fact (n-1)`
- To write a function $f :: [a] \rightarrow [a]$, use induction over the **length** of the argument; the base case is $f\ []$ and the inductive case is $f\ (x : xs)$ defined using $f\ xs$.

Example Write a function that concatenates two lists together. We perform induction on the length of the **first** argument:

- * Base case: `concat [] ys = ys`
- * Induction step: `concat (x:xs) ys = x : concat xs ys`
- Induction is also an extremely useful tool to **prove** functions that are already written

EXAMPLE: LISTS AS SETS

- Membership ($x \in A$):


```
member x [] = False
member x (y:ys) | x == y = True
                  | True   = member x ys
```
- Union ($A \cup B$), intersection ($A \cap B$), difference ($A \setminus B$):

- Constructor: no recursion. `makeSet x = [x]`

EXAMPLE: LISTS AS SETS

- Membership ($x \in A$):

```
member x [] = False
member x (y:ys) | x == y = True
                  | True   = member x ys
```

- Union ($A \cup B$), intersection ($A \cap B$), difference ($A \setminus B$):

```
union [] t = t
union (x:xs) t | member x t = union xs t
                | True      = x : union xs t

intersection [] t = []
intersection (x:xs) t | member x t = x : intersection xs t
                      | True      = intersection xs t

difference [] t = []
difference (x:xs) t | member x t = difference xs t
                    | True      = x : difference xs t
```

- Constructor: no recursion. `makeSet x = [x]`

HIGHER ORDER FUNCTIONS

In Haskell, all objects (including functions) are first class citizens. That is,

- all objects can be named,
- all objects can be members of a list/tuple,
- all objects can be passed as arguments to functions,
- all objects can be returned from functions,
- all objects can be the value of some expression.

<pre>twice :: (a -> a) -> (a -> a) twice f = g where g x = f (f x)</pre>	<pre>twice :: (a -> a) -> a -> a twice f x = f (f x)</pre>
<pre>compose f g = h where h x = f (g x)</pre>	<pre>compose f g x = f (g x) -- or -- compose f g = f.g</pre>

TO CURRY OR NOT TO CURRY

curried form:	uncurried form:
<pre>compose :: (a->b) -> (c->a) -> c->b compose f g = f.g</pre>	<pre>compose :: (a->b, c->a) -> c->b compose (f,g) = f.g</pre>

- In Haskell, any function takes **one** argument and returns **one** values.

– What if we need more than one argument?

Uncurried We either present the arguments packed in a tuple, or

Curried We use partial application: we build a function that takes one argument and that return a function which in turn takes one argument and returns another function which in turn...

curried:	uncurried:
<pre>add :: (Num a) => a -> a -> a add x y = x + y -- equivalent to the explicit version -- add x = g -- where g y = x + y incr :: (Num a) => a -> a incr = add 1</pre>	<pre>add :: (Num a) => (a, a) -> a add (x,y) = x + y incr :: (Num a) => a -> a incr x = add (1,x)</pre>

TO CURRY OR NOT TO CURRY (CONT'D)

What if we have a curried function and we want an uncurried one (or the other way around)?

- The following two functions are **predefined**:

```
curry f = g
  where g a b = f (a,b)
uncurry g = f
  where f (a,b) = g a b
```

or:

Note that the two functions are curried themselves...

TO CURRY OR NOT TO CURRY (CONT'D)

What if we have a curried function and we want an uncurried one (or the other way around)?

- The following two functions are **predefined**:

```
curry f = g
  where g a b = f (a,b)
uncurry g = f
  where f (a,b) = g a b
```

or:

```
curry f a b = f (a,b)
uncurry g (a,b) = g a b
```

Note that the two functions are curried themselves...

NON-LOCAL VARIABLES

- Given a nonnegative number $x :: \text{Float}$, write a function `mySqrt` that computes an approximation of $\sqrt{x}$ with precision $\epsilon = 0.0001$.
 - Newton says that, if y_n is an approximation of $\sqrt{x}$, then a better approximation is $y_{n+1} = (y_n + x/y_n)/2$.

NON-LOCAL VARIABLES

- Given a nonnegative number $x :: \text{Float}$, write a function `mySqrt` that computes an approximation of $\sqrt{x}$ with precision $\epsilon = 0.0001$.
 - Newton says that, if y_n is an approximation of $\sqrt{x}$, then a better approximation is $y_{n+1} = (y_n + x/y_n)/2$.

```
mySqrt  :: Float -> Float
mySqrt x = sqrt' x
  where sqrt' y = if good y then y else sqrt' (improve y)
        good y = abs (y*y - x) < eps
        improve y = (y + x/y)/2
        eps = 0.0001
```
 - x is very similar to a global variable in procedural programming.

NON-LOCAL VARIABLES

- Given a nonnegative number $x :: \text{Float}$, write a function `mySqrt` that computes an approximation of $\sqrt{x}$ with precision $\epsilon = 0.0001$.
 - Newton says that, if y_n is an approximation of $\sqrt{x}$, then a better approximation is $y_{n+1} = (y_n + x/y_n)/2$.

```
mySqrt  :: Float -> Float
mySqrt x = sqrt' x
  where sqrt' y = if good y then y else sqrt' (improve y)
        good y = abs (y*y - x) < eps
        improve y = (y + x/y)/2
        eps = 0.0001
```
 - x is very similar to a global variable in procedural programming.
 - Even closer to procedural programming:

```
mySqrt  :: Float -> Float
mySqrt x = until done improve x
  where done y = abs (y*y - x) < eps
        improve y = (y + x/y)/2
        eps = 0.0001
```

NON-LOCAL VARIABLES

- Given a nonnegative number $x :: \text{Float}$, write a function `mySqrt` that computes an approximation of $\sqrt{x}$ with precision $\epsilon = 0.0001$.
 - Newton says that, if y_n is an approximation of $\sqrt{x}$, then a better approximation is $y_{n+1} = (y_n + x/y_n)/2$.

```
mySqrt :: Float -> Float
mySqrt x = sqrt' x
  where sqrt' y = if good y then y else sqrt' (improve y)
        good y = abs (y*y - x) < eps
        improve y = (y + x/y)/2
        eps = 0.0001
```

- x is very similar to a global variable in procedural programming.
- Even closer to procedural programming:

<pre>mySqrt :: Float -> Float mySqrt x = until done improve x where done y = abs (y*y - x) < eps improve y = (y + x/y)/2 eps = 0.0001</pre>	<pre>until :: (a -> Bool) -> (a -> a) -> a -> a until p f x p x = x True = until p f (f x)</pre>
---	---

ACCUMULATING RESULTS

1.

```
mystery x = aux x []
  where aux [] ret = ret
        aux (x:xs) ret = aux xs (x:ret)
```

ACCUMULATING RESULTS

1.

```
reverse x = pour x []
  where pour [] ret = ret
        pour (x:xs) ret = pour xs (x:ret)
```

2.

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

- What is the difference between these two implementations?

MAPS

- `map` applies a function to each element in a list.

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

- For example:

```
upto m n = if m > n then [] else m: upto (m+1) n
square x = x * x
```

```
Prelude> map ((<) 3) [1,2,3,4]
[True,True,False,False]
Prelude> sum (map square (upto 1 10))
385
Prelude>
```

MAPS (CONT'D)

- Intermission: *zip* and *unzip*.

```
Prelude> zip [0,1,2,3,4] "hello"
[(0,'h'),(1,'e'),(2,'l'),(3,'l'),(4,'o')]
Prelude> zip [0,1,2,3,4,5,6] "hello"
[(0,'h'),(1,'e'),(2,'l'),(3,'l'),(4,'o')]
Prelude> unzip [(0,'h'),(1,'e'),(2,'l'),(4,'o')]
([0,1,2,4],"helo")
Prelude>
```

- A more complex (and useful) example of *map*:

```
mystery :: (Ord a) => [a] -> Bool
mystery xs = and (map (uncurry (<=)) (zip xs (tail xs)))
```

FILTERS

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = if p x then x : filter xs else filter xs
```

- Example:

```
mystery :: [(String,Int)] -> [String]
mystery xs = map fst (filter ( ((<=) 80) . snd ) xs)
```

MAPS (CONT'D)

- Intermission: *zip* and *unzip*.

```
Prelude> zip [0,1,2,3,4] "hello"
[(0,'h'),(1,'e'),(2,'l'),(3,'l'),(4,'o')]
Prelude> zip [0,1,2,3,4,5,6] "hello"
[(0,'h'),(1,'e'),(2,'l'),(3,'l'),(4,'o')]
Prelude> unzip [(0,'h'),(1,'e'),(2,'l'),(4,'o')]
([0,1,2,4],"helo")
Prelude>
```

- A more complex (and useful) example of *map*:

```
nondec :: (Ord a) => [a] -> Bool
nondec xs = and (map (uncurry (<=)) (zip xs (tail xs)))
```

- This finds whether the argument list is in nondecreasing order.

FILTERS

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = if p x then x : filter xs else filter xs
```

- Example:

```
getAs :: [(String,Int)] -> [String]
getAs xs = map fst (filter ( ((<=) 80) . snd ) xs)
```

```
Prelude> getAs [("a",70),("b",80),("c",91),("d",79)]
["b","c"]
```

- The final grades for some course are kept as a list of pairs (student name, grade). Find all the students that got an A.

FOLDS

$$[l_1, l_2, \dots, l_n] \xrightarrow{\text{foldr}} l_1 \bullet (l_2 \bullet (l_3 \bullet (\dots \bullet (l_n \bullet \text{id}) \dots)))$$

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op id [] = id
foldr op id (x:xs) = x 'op' (foldr op id xs)
```

$$[l_1, l_2, \dots, l_n] \xrightarrow{\text{foldl}} (\dots (((\text{id} \bullet l_1) \bullet l_2) \bullet l_3) \bullet \dots \bullet l_n)$$

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl op id [] = id
foldl op id (x:xs) = foldl op (x 'op' id) xs
```

- Almost all the interesting functions on lists are or can be implemented using *foldr* or *foldl*:

```
and = foldr (&&) True          concat = foldr (++) []
sum = foldr (+) 0              length = foldr oneplus 0
map f = foldr ((:).f) []       where oneplus x n = 1 + n
```

LIST COMPREHENSION

- Examples:

```
triples :: Int -> [(Int,Int,Int)]
triples n = [(x,y,z) | x <- [1..n], y <- [1..n], z <- [1..n]]
-- or [(x,y,z) | x <- [1..n], y <- [x..n], z <- [z..n]]
pyth :: (Int,Int,Int) -> Bool
pyth (x,y,z) = x*x + y*y == z*z
triads :: Int -> [(Int,Int,Int)]
triads n = [(x,y,z) | (x,y,z) <- triples n, pyth (x,y,z)]
```

- General form:

$$[exp | gen_1, gen_2, \dots, gen_n, guard_1, guard_2, \dots, guard_p]$$

- Quicksort:

```
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y <= x] ++ [x] ++
               qsort [y | y <- xs, y > x]
```