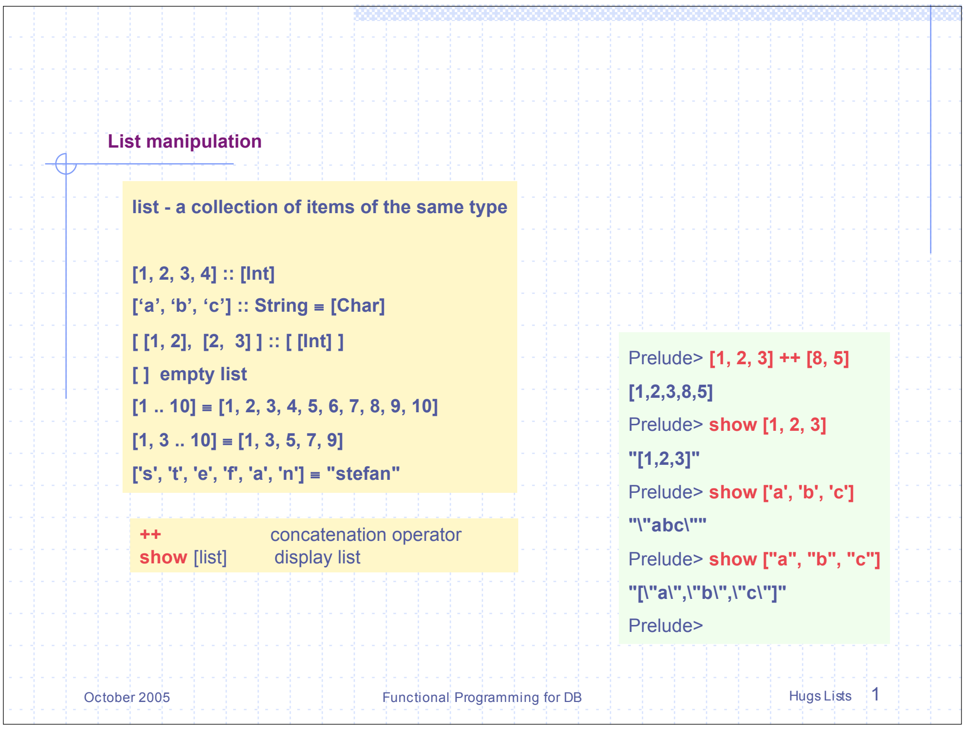


List manipulation

list - a collection of items of the same type

[1, 2, 3, 4] :: [Int]

[‘a’, ‘b’, ‘c’] :: String ≡ [Char]

[[1, 2], [2, 3]] :: [[Int]]

[ ] empty list

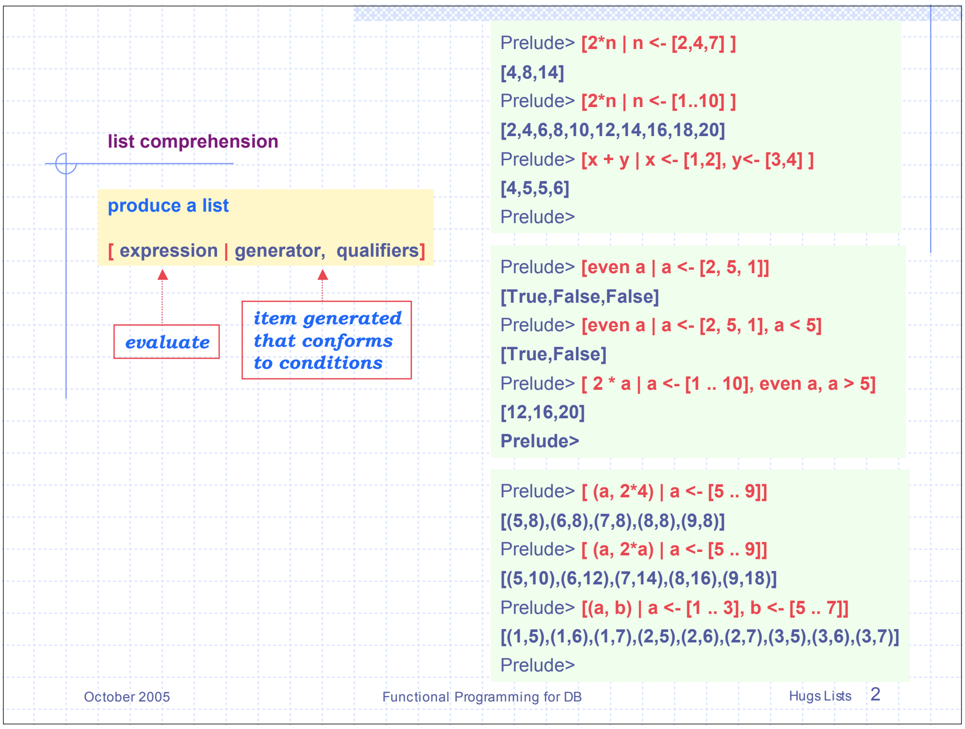
[1 .. 10] ≡ [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

[1, 3 .. 10] ≡ [1, 3, 5, 7, 9]

['s', 't', 'e', 'f', 'a', 'n'] ≡ "stefan"

++	concatenation operator
show [list]	display list

```
Prelude> [1, 2, 3] ++ [8, 5]
[1,2,3,8,5]
Prelude> show [1, 2, 3]
"[1,2,3]"
Prelude> show ['a', 'b', 'c']
"\"abc\""
Prelude> show ["a", "b", "c"]
"[\"a\",\"b\",\"c\"]"
Prelude>
```



list comprehension

produce a list

[expression | generator, qualifiers]

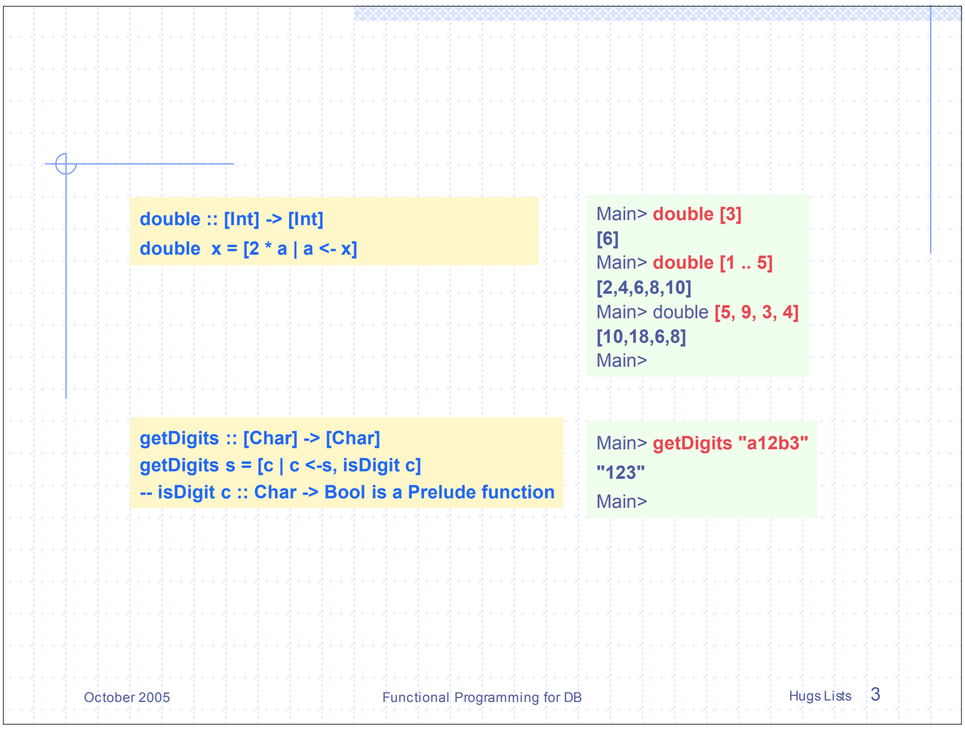
evaluate

item generated that conforms to conditions

```
Prelude> [2*n | n <- [2,4,7] ]
[4,8,14]
Prelude> [2*n | n <- [1..10] ]
[2,4,6,8,10,12,14,16,18,20]
Prelude> [x + y | x <- [1,2], y<- [3,4] ]
[4,5,5,6]
Prelude>
```

```
Prelude> [even a | a <- [2, 5, 1]]
[True,False,False]
Prelude> [even a | a <- [2, 5, 1], a < 5]
[True,False]
Prelude> [ 2 * a | a <- [1 .. 10], even a, a > 5]
[12,16,20]
Prelude>
```

```
Prelude> [ (a, 2*4) | a <- [5 .. 9]]
[(5,8),(6,8),(7,8),(8,8),(9,8)]
Prelude> [ (a, 2*a) | a <- [5 .. 9]]
[(5,10),(6,12),(7,14),(8,16),(9,18)]
Prelude> [(a, b) | a <- [1 .. 3], b <- [5 .. 7]]
[(1,5),(1,6),(1,7),(2,5),(2,6),(2,7),(3,5),(3,6),(3,7)]
Prelude>
```

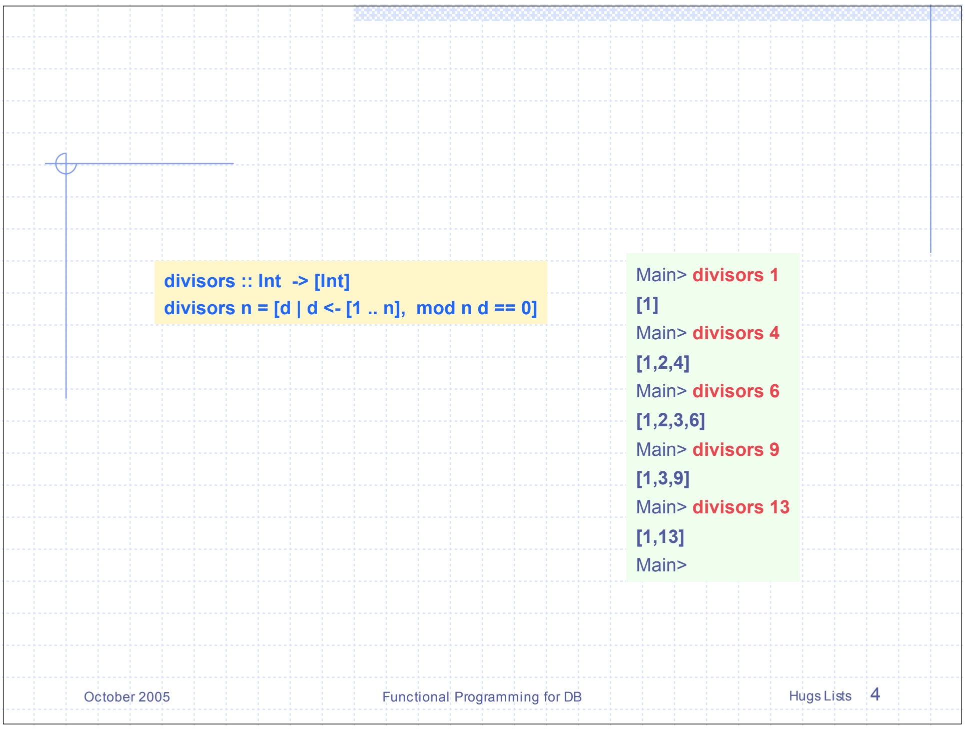


```
double :: [Int] -> [Int]
double  x = [2 * a | a <- x]
```

```
Main> double [3]
[6]
Main> double [1 .. 5]
[2,4,6,8,10]
Main> double [5, 9, 3, 4]
[10,18,6,8]
Main>
```

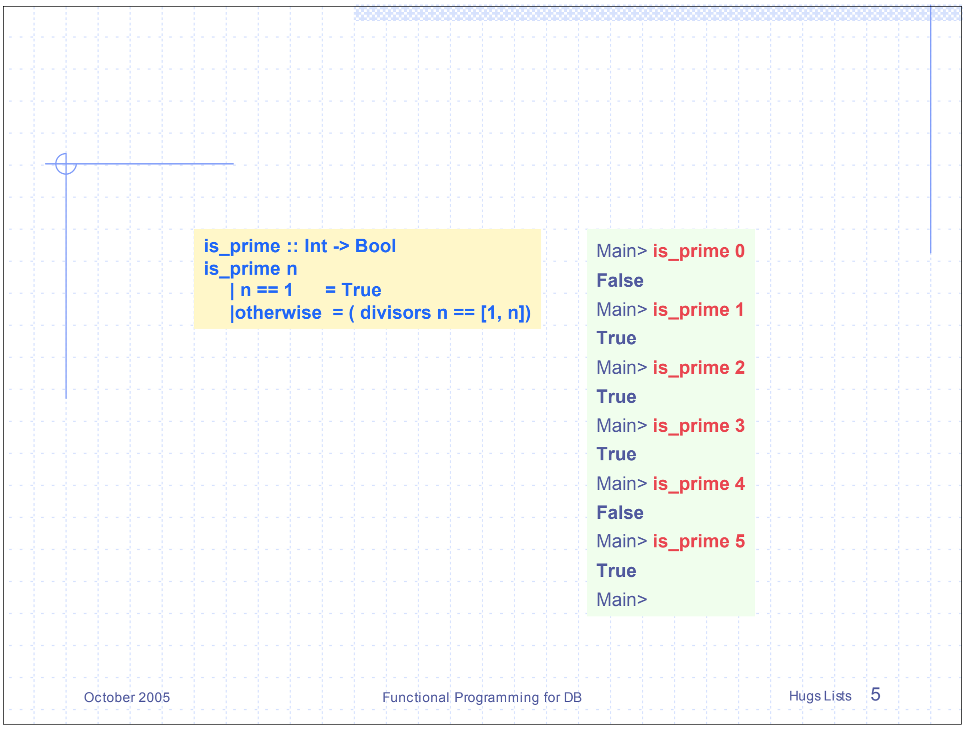
```
getDigits :: [Char] -> [Char]
getDigits s = [c | c <-s, isDigit c]
-- isDigit c :: Char -> Bool is a Prelude function
```

```
Main> getDigits "a12b3"
"123"
Main>
```



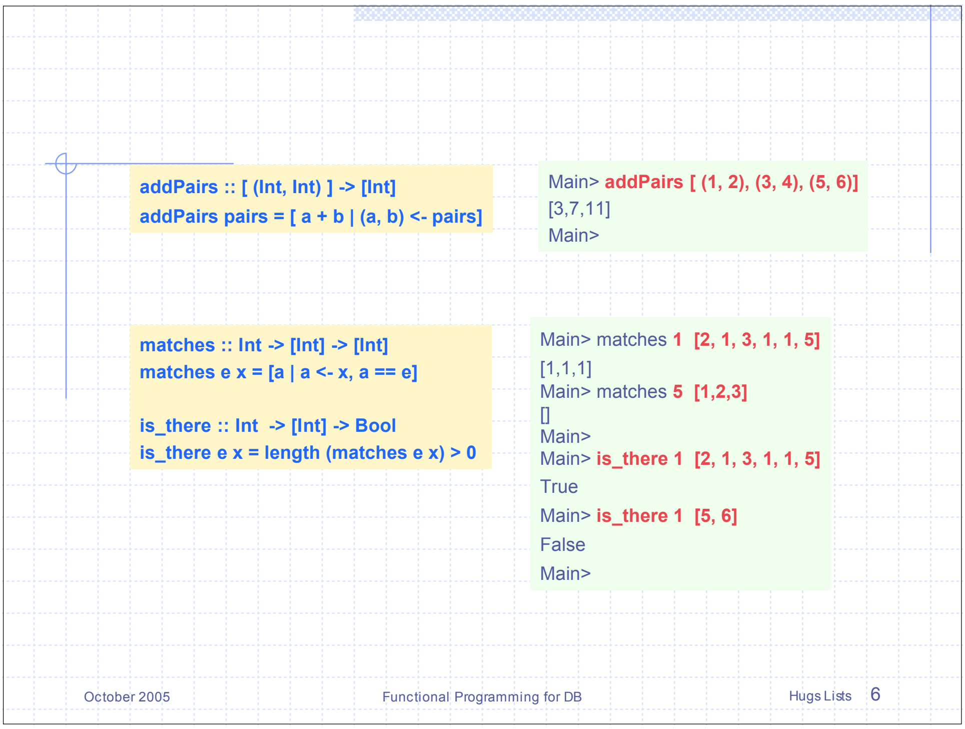
```
divisors :: Int  -> [Int]
divisors n = [d | d <- [1 .. n],  mod n d == 0]
```

```
Main> divisors 1
[1]
Main> divisors 4
[1,2,4]
Main> divisors 6
[1,2,3,6]
Main> divisors 9
[1,3,9]
Main> divisors 13
[1,13]
Main>
```



```
is_prime :: Int -> Bool
is_prime n
    | n == 1      = True
    |otherwise  = ( divisors n == [1, n])
```

```
Main> is_prime 0
False
Main> is_prime 1
True
Main> is_prime 2
True
Main> is_prime 3
True
Main> is_prime 4
False
Main> is_prime 5
True
Main>
```

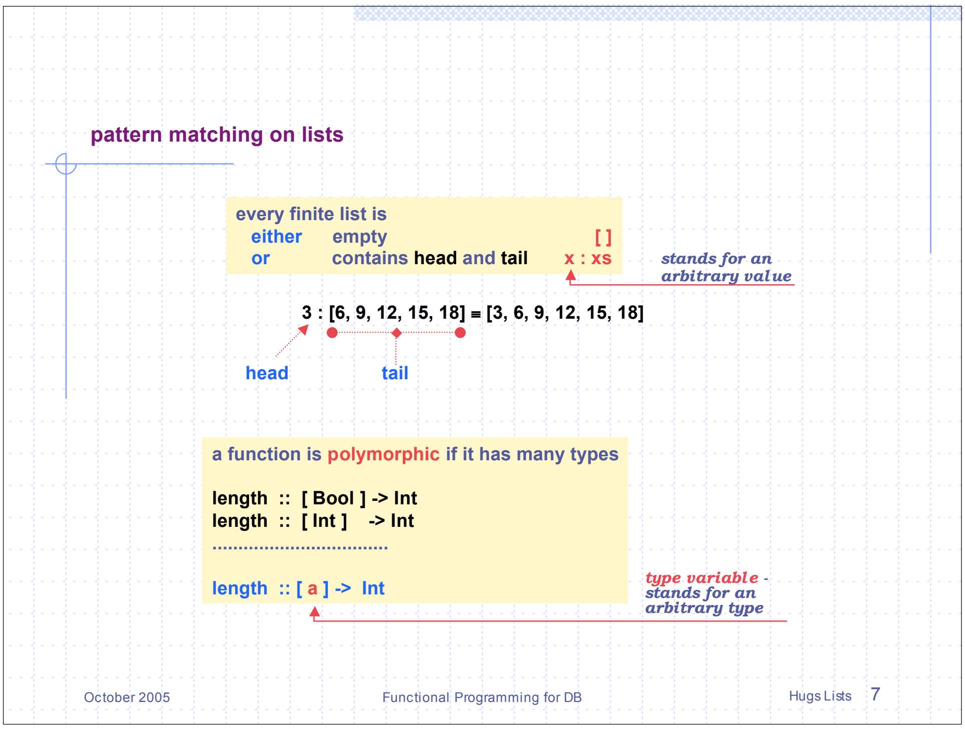


```
addPairs :: [ (Int, Int) ] -> [Int]
addPairs pairs = [ a + b | (a, b) <- pairs]
```

```
Main> addPairs [ (1, 2), (3, 4), (5, 6)]
[3,7,11]
Main>
```

```
matches :: Int -> [Int] -> [Int]
matches e x = [a | a <- x, a == e]

is_there :: Int  -> [Int] -> Bool
is_there e x = length (matches e x) > 0
```

```
Main> matches 1  [2, 1, 3, 1, 1, 5]
[1,1,1]
Main> matches 5  [1,2,3]
[]
Main>
Main> is_there 1  [2, 1, 3, 1, 1, 5]
True
Main> is_there 1  [5, 6]
False
Main>
```



pattern matching on lists

every finite list is
either empty [ ]
or contains head and tail x : xs

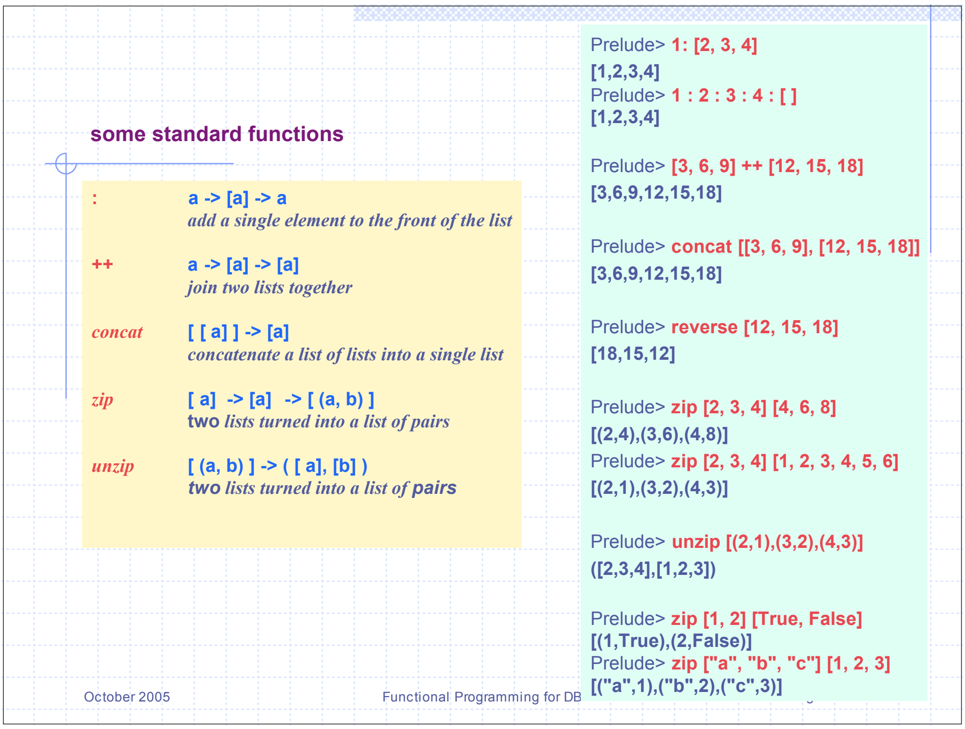
stands for an arbitrary value

3 : [6, 9, 12, 15, 18] ≡ [3, 6, 9, 12, 15, 18]

a function is polymorphic if it has many types

```
length  ::  [ Bool ] -> Int
length  ::  [ Int ]    -> Int
...............................

length  :: [ a ] ->  Int
```

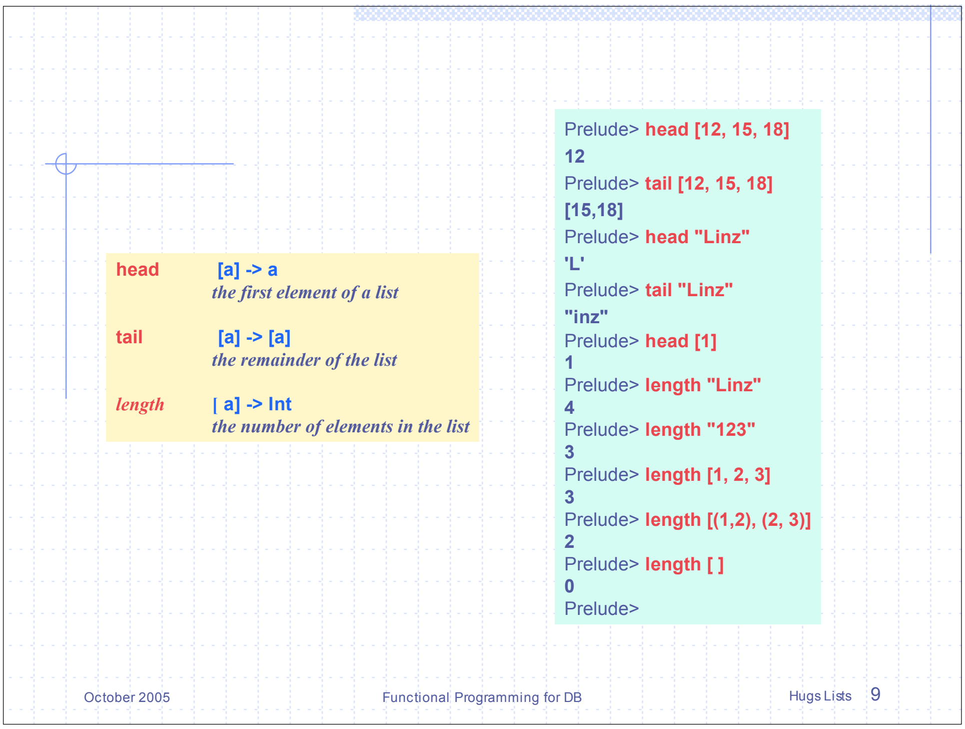
type variable - stands for an arbitrary type



some standard functions

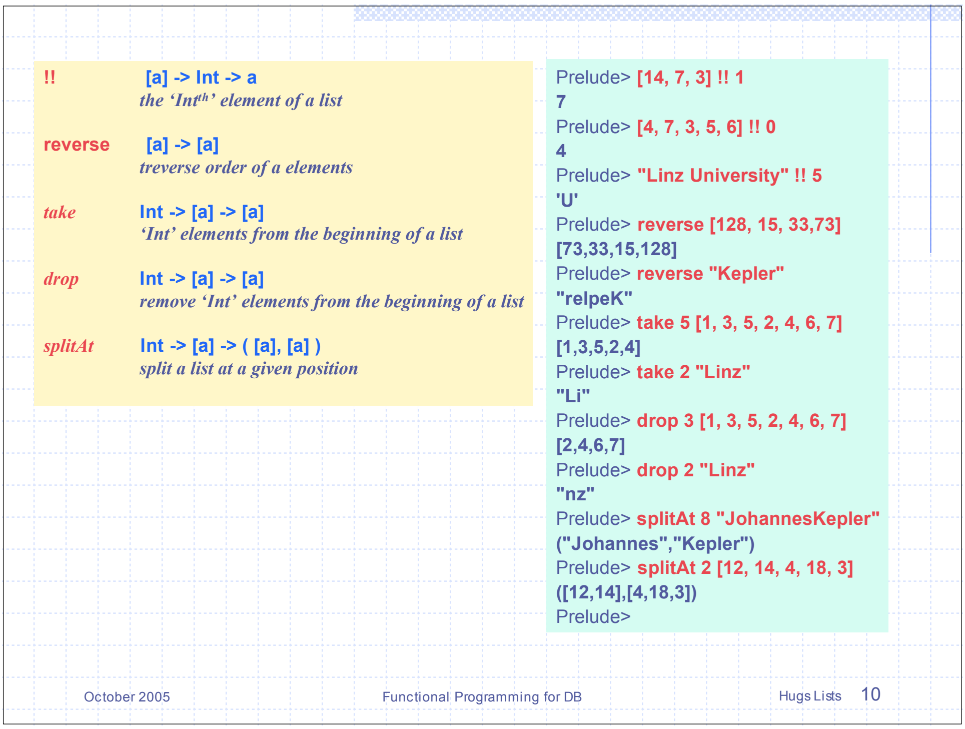
:	a -> [a] -> a *add a single element to the front of the list*
++	a -> [a] -> [a] *join two lists together*
concat	[[a]] -> [a] *concatenate a list of lists into a single list*
zip	[a] -> [a] -> [(a, b)] two *lists turned into a list of pairs*
unzip	[(a, b)] -> ([a], [b]) *two lists turned into a list of* pairs

```
Prelude> 1: [2, 3, 4]
[1,2,3,4]
Prelude> 1 : 2 : 3 : 4 : [ ]
[1,2,3,4]

Prelude> [3, 6, 9] ++ [12, 15, 18]
[3,6,9,12,15,18]

Prelude> concat [[3, 6, 9], [12, 15, 18]]
[3,6,9,12,15,18]

Prelude> reverse [12, 15, 18]
[18,15,12]

Prelude> zip [2, 3, 4] [4, 6, 8]
[(2,4),(3,6),(4,8)]
Prelude> zip [2, 3, 4] [1, 2, 3, 4, 5, 6]
[(2,1),(3,2),(4,3)]

Prelude> unzip [(2,1),(3,2),(4,3)]
([2,3,4],[1,2,3])

Prelude> zip [1, 2] [True, False]
[(1,True),(2,False)]
Prelude> zip ["a", "b", "c"] [1, 2, 3]
[("a",1),("b",2),("c",3)]
```



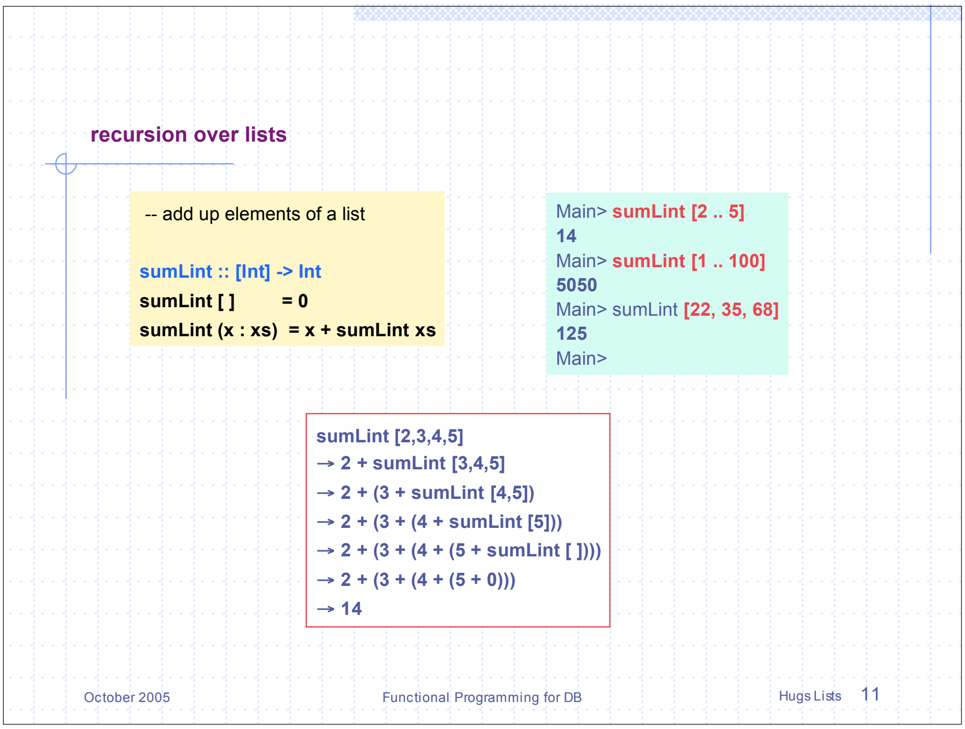
head	[a] -> a *the first element of a list*
tail	[a] -> [a] *the remainder of the list*
length	[a] -> Int *the number of elements in the list*

```
Prelude> head [12, 15, 18]
12
Prelude> tail [12, 15, 18]
[15,18]
Prelude> head "Linz"
'L'
Prelude> tail "Linz"
"inz"
Prelude> head [1]
1
Prelude> length "Linz"
4
Prelude> length "123"
3
Prelude> length [1, 2, 3]
3
Prelude> length [(1,2), (2, 3)]
2
Prelude> length [ ]
0
Prelude>
```



!!	[a] -> Int -> a *the ‘Intth’ element of a list*
reverse	[a] -> [a] *treverse order of a elements*
take	Int -> [a] -> [a] *‘Int’ elements from the beginning of a list*
drop	Int -> [a] -> [a] *remove ‘Int’ elements from the beginning of a list*
splitAt	Int -> [a] -> ([a], [a]) *split a list at a given position*

```
Prelude> [14, 7, 3] !! 1
7
Prelude> [4, 7, 3, 5, 6] !! 0
4
Prelude> "Linz University" !! 5
'U'
Prelude> reverse [128, 15, 33,73]
[73,33,15,128]
Prelude> reverse "Kepler"
"relpeK"
Prelude> take 5 [1, 3, 5, 2, 4, 6, 7]
[1,3,5,2,4]
Prelude> take 2 "Linz"
"Li"
Prelude> drop 3 [1, 3, 5, 2, 4, 6, 7]
[2,4,6,7]
Prelude> drop 2 "Linz"
"nz"
Prelude> splitAt 8 "JohannesKepler"
("Johannes","Kepler")
Prelude> splitAt 2 [12, 14, 4, 18, 3]
([12,14],[4,18,3])
Prelude>
```

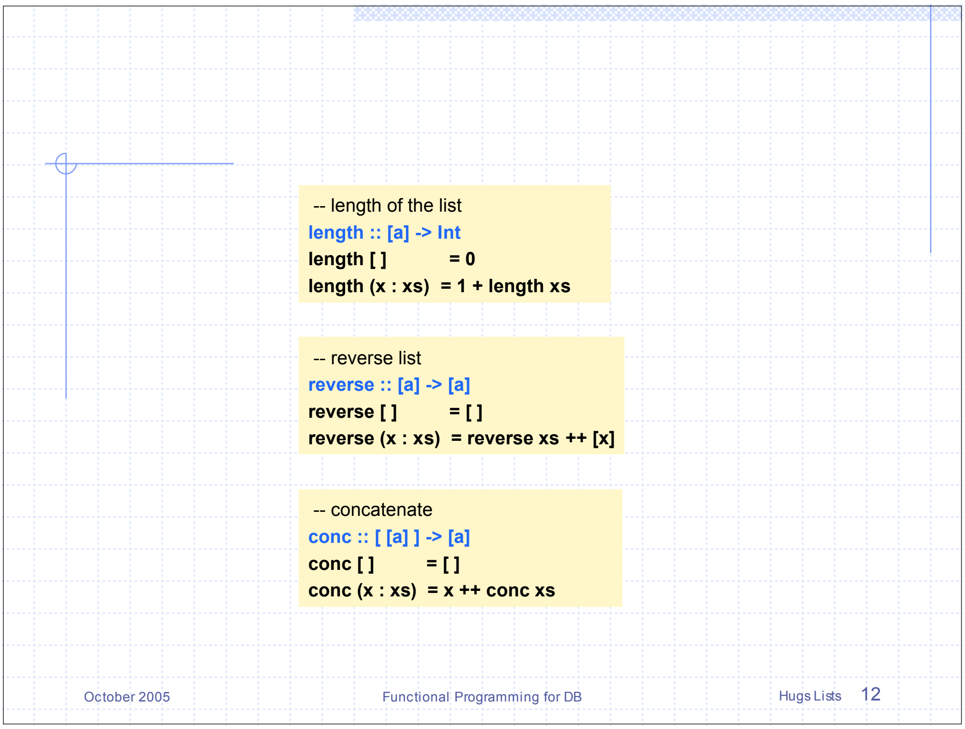


recursion over lists

```
-- add up elements of a list

sumLint :: [Int] -> Int
sumLint [ ]         = 0
sumLint (x : xs)  = x + sumLint xs
```

```
Main> sumLint [2 .. 5]
14
Main> sumLint [1 .. 100]
5050
Main> sumLint [22, 35, 68]
125
Main>
```

```
sumLint [2,3,4,5]
→ 2 + sumLint [3,4,5]
→ 2 + (3 + sumLint [4,5])
→ 2 + (3 + (4 + sumLint [5]))
→ 2 + (3 + (4 + (5 + sumLint [ ])))
→ 2 + (3 + (4 + (5 + 0)))
→ 14
```



```
-- length of the list
length :: [a] -> Int
length [ ]           = 0
length (x : xs)  = 1 + length xs
```

```
-- reverse list
reverse :: [a] -> [a]
reverse [ ]          = [ ]
reverse (x : xs)  = reverse xs ++ [x]
```

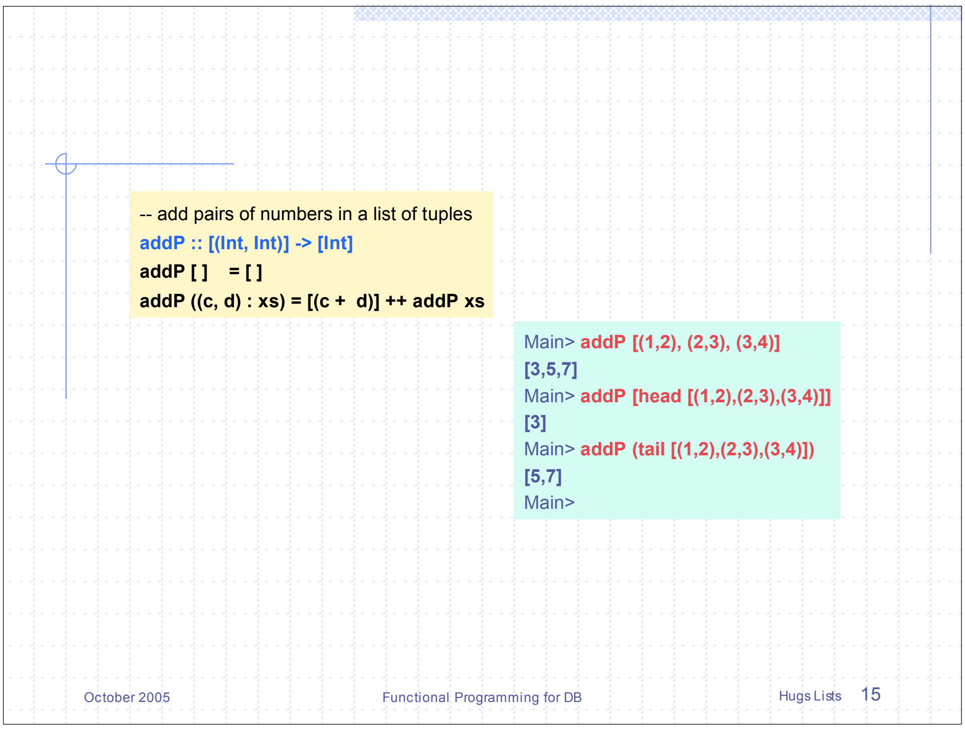
```
-- concatenate
conc :: [ [a] ] -> [a]
conc [ ]          = [ ]
conc (x : xs)  = x ++ conc xs
```

```
-- conjunction of elements within list
andL :: [Bool] -> Bool
andL [ ]       = True
andL (x : xs) = x && andL xs
```

```
Main> andL [True, False]
False
Main> andL [True, True]
True
Main> andL [True, True, False]
False
Main> andL [5==4,  25/2 >= 10]
False
Main> andL [5==5,  25/2 >= 10]
True
Main>
```

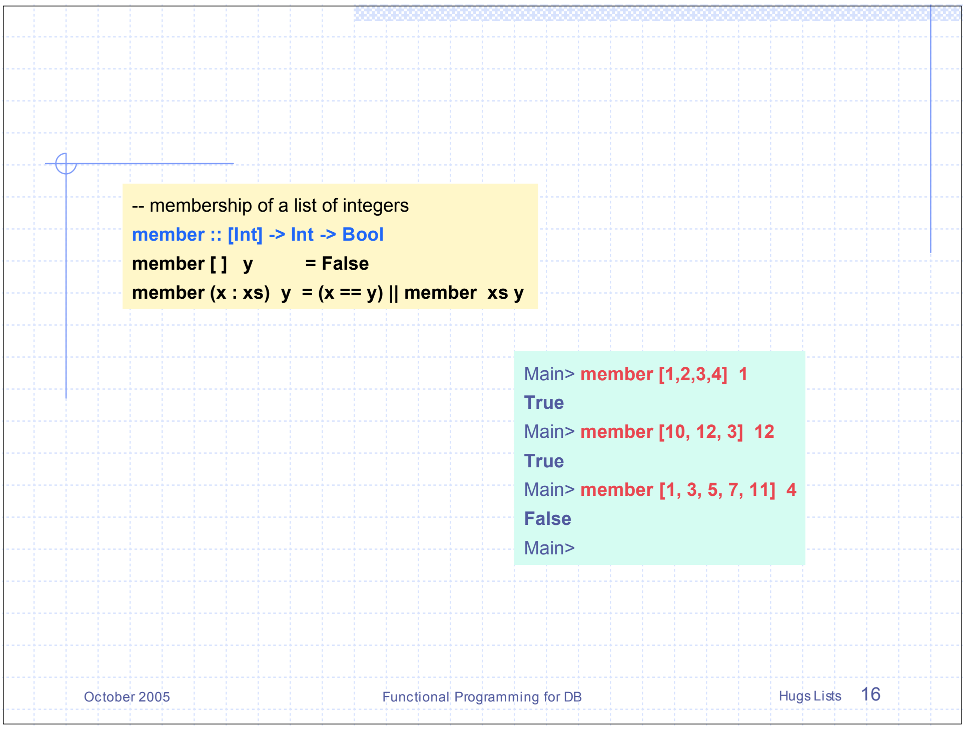
```
-- product of elements
timesL :: [Int] -> Int
timesL [ ]         = 1
timesL (x : xs) = x * timesL xs
```

```
Main> timesL [1,3,5]
15
Main> timesL [2,5,7]
70
Main> timesL [1 .. 5]
120
Main>
```



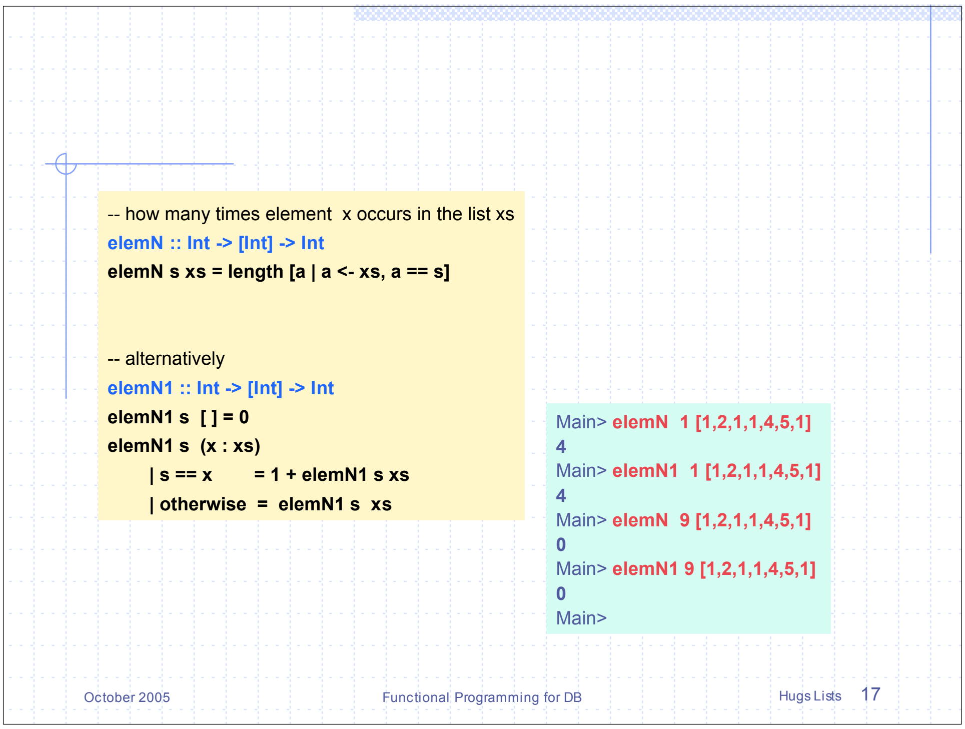
```
-- add pairs of numbers in a list of tuples
addP :: [(Int, Int)] -> [Int]
addP [ ]    = [ ]
addP ((c, d) : xs) = [(c +  d)] ++ addP xs
```

```
Main> addP [(1,2), (2,3), (3,4)]
[3,5,7]
Main> addP [head [(1,2),(2,3),(3,4)]]
[3]
Main> addP (tail [(1,2),(2,3),(3,4)])
[5,7]
Main>
```



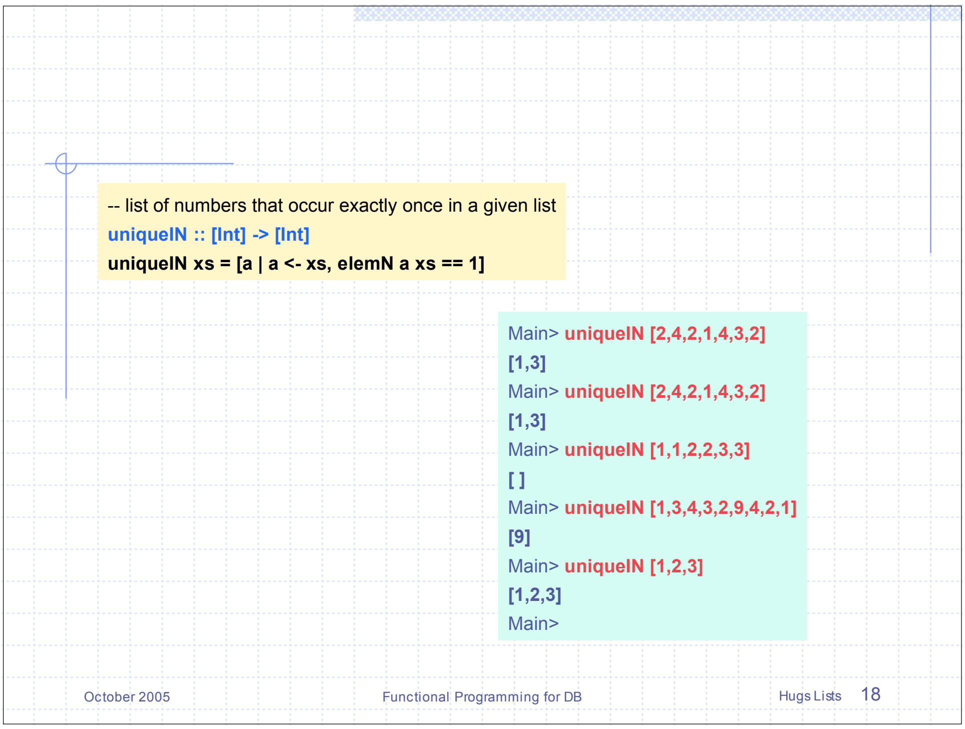
```
-- membership of a list of integers
member :: [Int] -> Int -> Bool
member [ ]   y          = False
member (x : xs)  y  = (x == y) || member  xs y
```

```
Main> member [1,2,3,4]  1
True
Main> member [10, 12, 3]  12
True
Main> member [1, 3, 5, 7, 11]  4
False
Main>
```



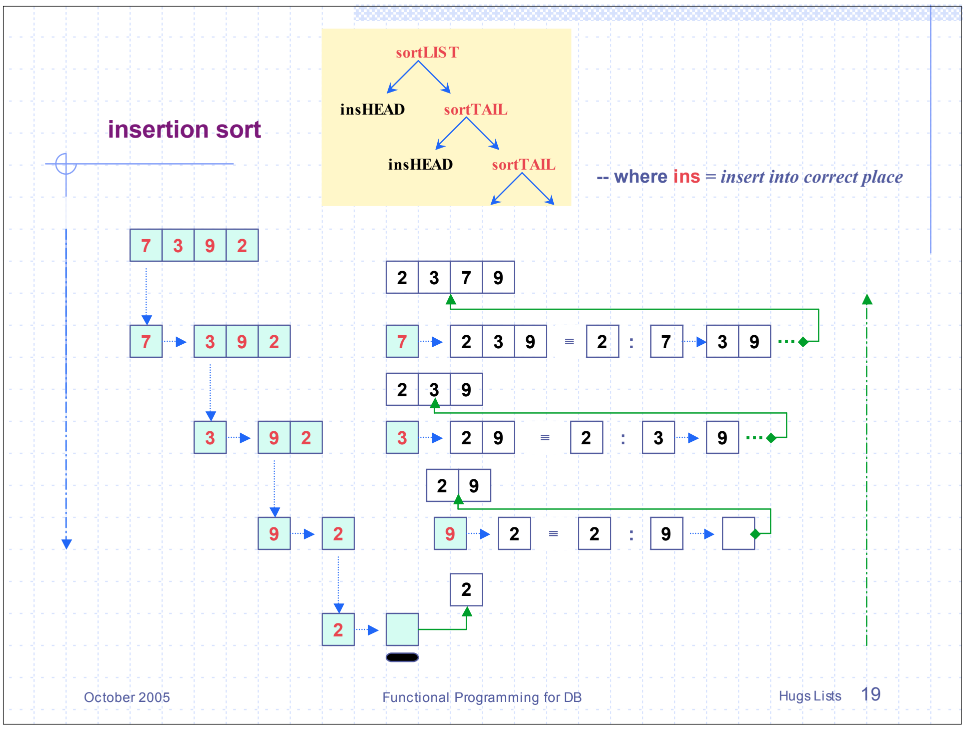
```
-- how many times element  x occurs in the list xs
elemN :: Int -> [Int] -> Int
elemN s xs = length [a | a <- xs, a == s]


-- alternatively
elemN1 :: Int -> [Int] -> Int
elemN1 s  [ ] = 0
elemN1 s  (x : xs)
       | s == x        = 1 + elemN1 s xs
       | otherwise  =  elemN1 s  xs
```

```
Main> elemN  1 [1,2,1,1,4,5,1]
4
Main> elemN1  1 [1,2,1,1,4,5,1]
4
Main> elemN  9 [1,2,1,1,4,5,1]
0
Main> elemN1 9 [1,2,1,1,4,5,1]
0
Main>
```



```
-- list of numbers that occur exactly once in a given list
uniqueIN :: [Int] -> [Int]
uniqueIN xs = [a | a <- xs, elemN a xs == 1]
```

```
Main> uniqueIN [2,4,2,1,4,3,2]
[1,3]
Main> uniqueIN [2,4,2,1,4,3,2]
[1,3]
Main> uniqueIN [1,1,2,2,3,3]
[ ]
Main> uniqueIN [1,3,4,3,2,9,4,2,1]
[9]
Main> uniqueIN [1,2,3]
[1,2,3]
Main>
```



insertion sort

sortLIST

insHEAD sortTAIL

insHEAD sortTAIL

-- where ins = *insert into correct place*

7 3 9 2

7 → 3 9 2

3 → 9 2

9 → 2

2 → □

2 3 7 9

7 → 2 3 9 = 2 : 7 → 3 9

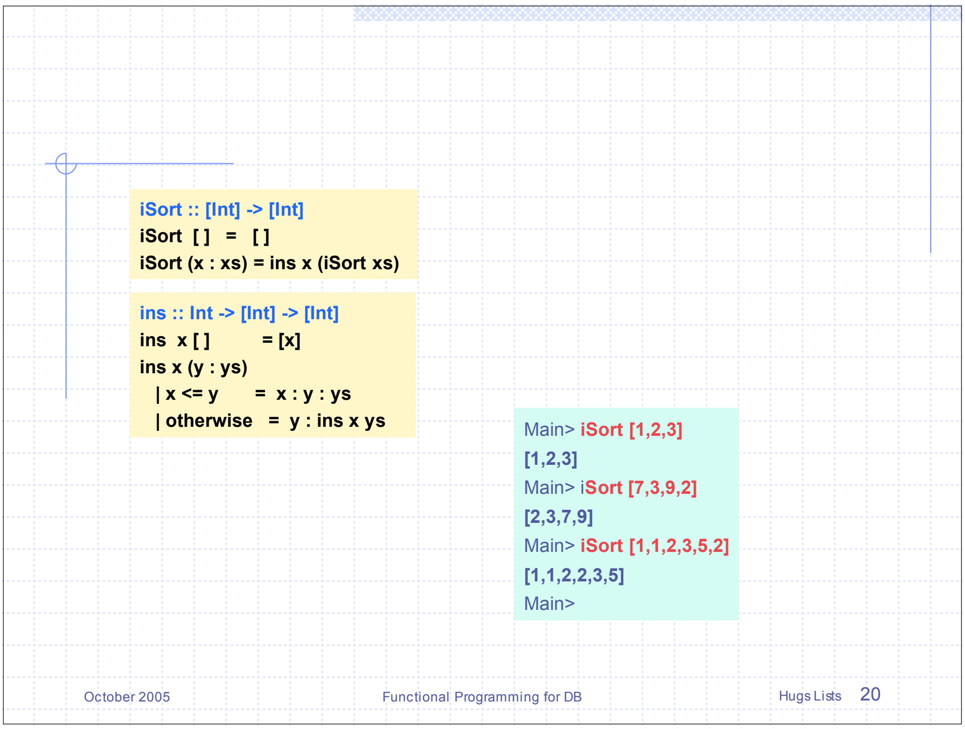
2 3 9

3 → 2 9 = 2 : 3 → 9

2 9

9 → 2 = 2 : 9 → □

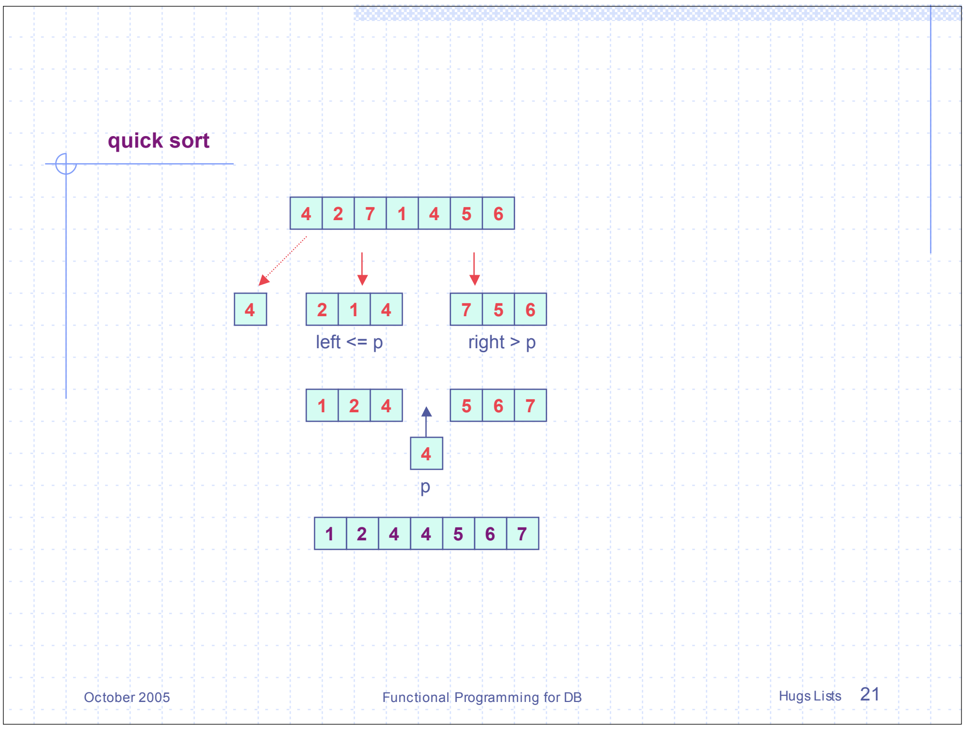
2



```
iSort :: [Int] -> [Int]
iSort  [ ]   =   [ ]
iSort (x : xs) = ins x (iSort xs)
```

```
ins :: Int -> [Int] -> [Int]
ins  x [ ]          = [x]
ins x (y : ys)
   | x <= y       =  x : y : ys
   | otherwise   =  y : ins x ys
```

```
Main> iSort [1,2,3]
[1,2,3]
Main> iSort [7,3,9,2]
[2,3,7,9]
Main> iSort [1,1,2,3,5,2]
[1,1,2,2,3,5]
Main>
```



quick sort

4 2 7 1 4 5 6

4

2 1 4

left <= p

7 5 6

right > p

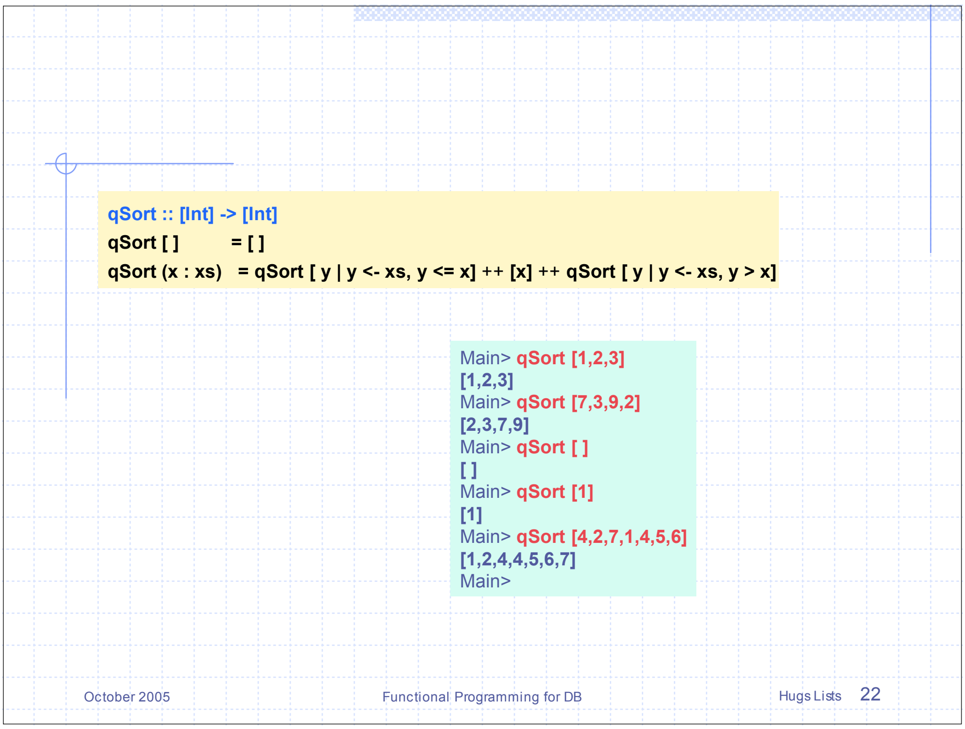
1 2 4

5 6 7

4

p

1 2 4 4 5 6 7



```
qSort :: [Int] -> [Int]
qSort [ ]        = [ ]
qSort (x : xs)   = qSort [ y | y <- xs, y <= x] ++ [x] ++ qSort [ y | y <- xs, y > x]
```

```
Main> qSort [1,2,3]
[1,2,3]
Main> qSort [7,3,9,2]
[2,3,7,9]
Main> qSort [ ]
[ ]
Main> qSort [1]
[1]
Main> qSort [4,2,7,1,4,5,6]
[1,2,4,4,5,6,7]
Main>
```