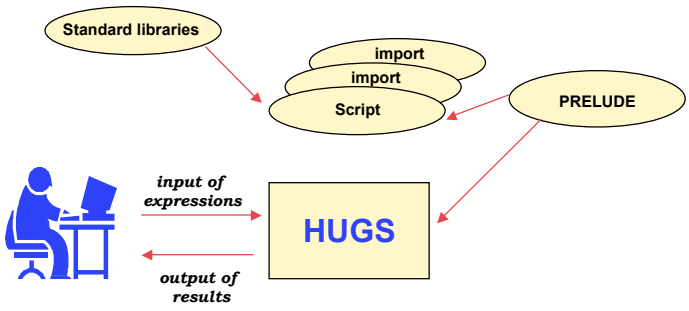
Haskell through HUGS
THE BASICS

Algorithmic Imperative Languages

- variables
- assignment
- if condition then action1 else action2
- loop
  - while condition do action
  - repeat action until condition
  - for i = start to finish do action
- block begin program end
- data types (extendible)
- record
- data structures (for algorithms)
  - predefined | user defined
  - ordered sets od data
- abstract data types
  - data structure + basic operations
  - forming a conceptual machine
- procedures
  - extend the language expressive power
  - increase transparency of the program
- recursion

Functional Programming Languages

- declarativeness
  encoding rules of tranformation
  rather than prescribing execution
- every object
  is a function - which itself can be an object
- application as the main operation
- evaluation (β-reduction)
- referential transparency -
  replace names by their values at any
  time (substituting equals by equals)
- higher order functions
  function(function) → function
- polymorphism
  same algorithm works on many kinds of inputs
- recursion



a program in Haskell (script) is a list of function definitions
it is recorded in a file loaded into the system by :load

the programme can be changed and reloaded by :reload

programs may import other programs

```
__   __ __  __  ____   ___     ____________________________________
||   || ||  || ||  || ||__     Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||    Copyright (c) 1994-2001
||---||         ___||          World Wide Web: http://haskell.org/hugs
||   ||                        Report bugs to: hugs-bugs@haskell.org
||   || Version: December 2001 ____________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Reading file "Macintosh HD:Desktop Folder:LinzFPDB:hugs98:lib:Prelude.hs":

Hugs session for:
Macintosh HD:Desktop Folder:LinzFPDB:hugs98:lib:Prelude.hs
Type :? for help

Prelude>
Prelude> 1 + 2
3
Prelude> (+1) 2
3
Prelude> + 1 2
ERROR - Syntax error in expression (unexpected token)
Prelude> 5*6
30
Prelude> 5*6/2
15.0
Prelude> (((1 + 3) * 5 - 2 * 3 + 17/2) * (3 + 8) - 17/2) * 2.5
597.5
```



```
Prelude> pi
3.14159
Prelude> sin pi
-8.74228e-08
Prelude> log pi
1.14473
Prelude> sqrt 5
2.23607
Prelude> sqrt 100
10.0
Prelude> sin 0
0.0
Prelude> log (sin (pi/4))
-0.346574
```

```
Prelude> False
False
Prelude> True
True
Prelude> not False
True
Prelude> not True
False
Prelude> not (not True)
True
Prelude> not (not False)
False
Prelude> not False && True
True
Prelude> True || False
True
Prelude> not (True && False)
True
Prelude> 5 == 4
False
Prelude> 5 /= 5
False
Prelude> 5 == 5
True
```



```
Prelude> reverse "Stefan"
"nafetS"
Prelude> reverse (reverse "Stefan")
"Stefan"
Prelude> even 3
False
Prelude> even 4
True
Prelude> sum [1..10]
55
Prelude> filter even [1..10]
[2,4,6,8,10]
Prelude> odd  2
False
Prelude> odd  3
True
```

```
Prelude> filter odd [1..10]
[1,3,5,7,9]
Prelude> sum (filter even [1..10])
30
Prelude> sum (filter odd [1..10])
25
Prelude> reverse (filter odd [1..10])
[9,7,5,3,1]
Prelude> map (1+) [1..10]
[2,3,4,5,6,7,8,9,10,11]
Prelude> map (1+) (map (1+) [1..10])
[3,4,5,6,7,8,9,10,11,12]
Prelude> filter even $$
[4,6,8,10,12]
Prelude> putStr "hello"
hello
Prelude> print "hello"
"hello"
Prelude> > putStr "How " >> putStr "are " >> putStr "you"
How are you
Prelude>:q
```



```
Prelude> :?
LIST OF COMMANDS:  Any command may be abbreviated to :c
                   where c is the first character in the full name.

:load <filenames>     load modules from specified files
:load                 clear all files except prelude
:also <filenames>     read additional modules
:reload               repeat last load command
:project <filename>   use project file
:edit <filename>      edit file
:edit                 edit last module
:module <module>      set module for evaluating expressions
<expr>                evaluate expression
:type <expr>          print type of expression
:?                    display this list of commands
:set <options>        set command line options
:set                  help on command line options
:names [pat]          list names currently in scope
:info <names>         describe named objects
:browse <modules>     browse names defined in <modules>
:find <name>          edit module containing definition of name
:!command             shell escape
:cd dir               change directory
:gc                   force garbage collection
:version              print Hugs version
:quit                 exit Hugs interpreter
```

```
-- content of First.hs

size :: Int
size = 10 + 20

square :: Int -> Int
square n = n*n

double :: Int -> Int
double  n = 2*n

example :: Int
example = double (size - square (2+2))

plus5 :: Int -> Int
plus5 a = a + 5

plus :: Int -> Int  -> Int
plus a  b = a + b

power4 :: Int -> Int
power4 n = square (square(n))

power3 n = square(n) * n

-- end of file
```

```
Prelude> :l First
Reading file "First.hs":

Hugs session for:
Macintosh HD:Desktop
Folder:LinzFPDB:hugs98:lib:Prelude.hs
First.hs
Main> size
30
Main> square 3
9
Main> double 5
10
Main> example
28
Main> plus 5 7
12
Main> plus 3 5
8
Main> power4 2
16
Main> power3 2
8
Main>
```



```
Main> power3 (plus5 example)
35937
Main> double (plus 5 4)
18
Main> (plus 5 4) * (5 - 4)
9
Main> double (square ((plus 5 4) * (5 - 4) - example))
722
Main> $$
722
Main> div 13 5
2
Main> mod 13 5
3
Main> 13 / 5
2.6
Main> :t square
square :: Int -> Int
Main>:q
```

style

function definitions are equations and should be preceeded by type declarations (which can be omitted)

filename.hs

```
-- this is a definition of my plus

plus :: Int -> Int  -> Int
plus a  b = a + b
```

conventional style

filename.lhs

```
this is a definition of my plus

> plus :: Int -> Int  -> Int
> plus a  b = a + b
```

literate style

every value and function has a type

every script is checked before the execution, hence type errors are not possible at run time

currying

functions of multiple arguments are usually curried

```
plus          :: (Int, Int) -> Int
plus (x, y)  =  x + y
```

conventional

```
plusC       :: Int -> Int -> Int
plusC x y   =  x + y
```

curried

function plusC can be applied to one argument - (plusC 3) takes a number and adds 3 to it

left associativity

function application has the highest priority

plusC x y means (plusC x) y rather than plus (x y)

square square 3 means (square square) 3

```
Main> square square 3
ERROR - Type error in application
*** Expression     : square square 3
*** Term           : square
*** Type           : Int -> Int
*** Does not match : a -> b -> c

Main> :type square
square :: Int -> Int
```

```
Main>
Main> square (square 3)
81
Main>
```


function composition

if **f: A → B** and **g: B → C**
then **(f . g) : A → C (f . g)x = f (g x)**

```
Main> (square . square) 3
81
Main>
```

```
Main> square . square 3
ERROR - Type error in application
*** Expression     : square . square 3
*** Term           : square 3
*** Type           : Int
*** Does not match : a -> b
```

Boolean

Constants	**True**, **False**
Logical operators	**&&** **||** **not** and or not
Relational operators	**==** **/=** **>** **>=** **<** **<=**

== and **/=** are used for both **integers** and **booleans;** the operators are termed **overloaded**

```
Main> (1==2) && (1/0 > 5)
False

Main> 1/0
Program error: {primDivDouble 1.0 0.0}
Main >
```

evaluated as
(1 == 2) && whatever
False && whatever
False

lazy evaluation


```
xOR      :: Bool -> Bool -> Bool
xOR x y  = (x || y) && not (x && y)
```

```
nAND     :: Bool -> Bool -> Bool
nAND x y = not (x && y)
```

```
same3          :: Int -> Int -> Int -> Bool
same3 m n p    = (m == n) && (n == p)
```

```
same3 1 1 7
→(m == n) && (n == p)
→(1 == 1) && (1 == 7)
→True && False
→False
```

```
same4          :: Int -> Int -> Int -> Int -> Bool
same4 m n p q  = (m == n) && (n == p) && (p == q)
```

```
same4          :: Int -> Int -> Int -> Int -> Bool
same4 m n p q  = same3 m n p && (p == q)
```

guards and conditionals

```
bigger      :: Int -> Int -> Int
bigger  x y
        | x >= y            = x
        | otherwise         = y
```

```
biggest3    :: Int -> Int -> Int -> Int
biggest3  x y z
        | x >= y && x >= z              = x
        | y >= z                        = y
        | otherwise                     = z
```

```
bigif       ::  Int -> Int -> Int
bigif  x y
            = if x >y then x else y
```


ASCII codes

```
Prelude> :type ord
ord :: Char -> Int


Prelude> ord 'a'
97
Prelude> ord 'b'
98
Prelude> ord 'z'
122
Prelude> ord 'A'
65
Prelude> ord 'B'
66
Prelude> ord 'Z'
90
Prelude> ord '\t'
9
Prelude> ord '\n'
10
Prelude>
```

```
Prelude> :type chr
chr :: Int -> Char


Prelude> chr  97
'a'
Prelude> chr  98
'b'
Prelude> chr  122
'z'
Prelude> chr  65
'A'
Prelude> chr  66
'B'
Prelude> chr  90
'Z'
Prelude> chr  9
'\t'
Prelude> chr  10
'\n'
Prelude>
```

```
Prelude> a
ERROR - Undefined variable "a"
Prelude> 'a'
'a'
Prelude> 3
3
Prelude> 3 == 3
True

Prelude> '3' == 3
ERROR - Illegal Haskell 98 class constraint in inferred type
*** Expression : '3' == 3
*** Type       : Num Char => Bool

Prelude> 'a' == 3
ERROR - Illegal Haskell 98 class constraint in inferred type
*** Expression : 'a' == 3
*** Type       : Num Char => Bool

Prelude> 'a' == '3'
False
Prelude>
```



```
-- digit to its value; zero for non-digits

digitTOnum :: Char -> Int
digitTOnum c
        | 1 < n && n <= 9       = n
        | otherwise             = 0
            where n = ord c - ord '0'
```

```
Main> digitTOnum '2'
2
Main> digitTOnum '9'
9
Main> digitTOnum '0'
0
Main> digitTOnum 'a'
0
Main>
```

```
-- how many roots in  ax^2 + bx + c = 0

howmany :: Float -> Float -> Float -> Int
howmany a b c
        | discriminant > 0      = 2
        | discriminant  == 0    = 1
        | discriminant < 0       = 0
            where discriminant = b^2 - 4 * a * c
```

local definition

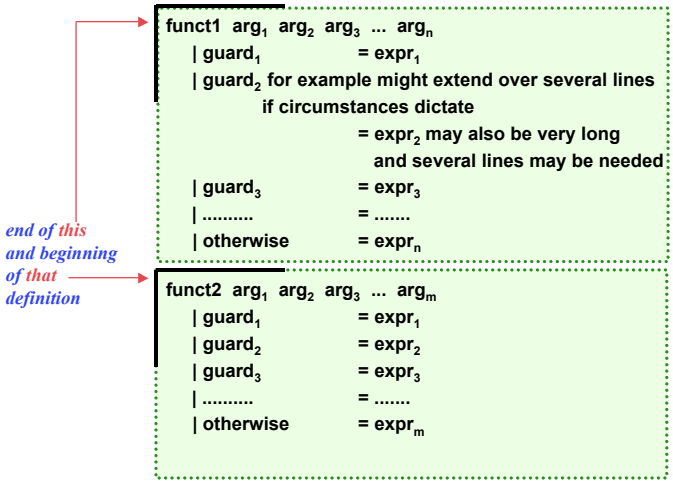
```
Main> howmany 1 (-2) 1
1
Main> howmany 5 4 3
0
Main> howmany 1 0 (-4)
2
Main>
```

x^2 -2x +1 = (x-1)^2

5x^2 + 4x 3

x^2 - 4 = (x-2)(x+2)

layout rule (blocks show the structure)



simple recursion

```
-- factorial

fac  :: Int -> Int
fac n
  | n == 0       = 1
  | n > 0        = fac (n - 1) * n
```

```
Main> fac 0
1
Main> fac 1
1
Main> fac 2
2
Main> fac 5
120
Main> fac 9
362880
Main> fac (-1)

Program error: {fac (-1)}

Main>
```



```
-- multiplication

times :: Int -> Int -> Int
times m n
      | n == 0       = 0
      | n > 0        = times m (n-1) + m


-- exponentiation

power :: Int -> Int -> Int
power  m n
       | n == 0        = 1
       | n > 0         =  times (power m (n - 1)) m
```

```
Main> power 2  4
16
Main> power 2  5
32
Main> power 2  10
1024
Main> power 3  2
9
Main> power 3  4
81
Main> power 5  4
625
Main> power 10  10
1410065408
Main> power 0  0
1
Main> power 0  1
0
Main> power 1  0
1
Main>
```

```
-- nth Fibonacci number

fib :: Int -> Int
fib n
  | n == 0    = 0
  | n == 1    = 1
  | n > 1     = fib (n - 2) + fib (n -1)
```

0	1	1	2	3	5	8	13	21
0	1	2	3	4	5	6	7	8

```
Main> fib 0
0
Main> fib 1
1
Main> fib 2
1
Main> fib 7
13
Main> fib 8
21
Main>
```


names

identifiers - alphanumeric strings, starting with a letter

functions & variables must start with a lower-case letter

types, type constructors, type classes start with a capital letter

reserved words

case	class	data	default	deriving	do
else	if	infix	infix1	infixr	instance
let	module	newtype	of	then	type
where					

tuples (records)

student-record

name id mark

("Hans", "s887655", 92) :: (String, String, Int)

belongs to the **tuple type**

```
type Student = (String, String, Int)
hans :: Student
hans = ("Hans", "s887655", 92)
```

```
type Cohort = [Student]
[("Hans", "s887655", 92), ("Mary", "s887123", 65), ("Anne", s8870091", 94)]
```

list of students, each elemrnt of the list is of the same type

functions over tuples - pattern matching

```
projection
first, second :: (Int, Int) -> Int
first (x, y) = x
second (x, y) = y
```

```
Prelude> fst ("john", "mary")
"john"
Prelude> snd ("john", "mary")
"mary"
Prelude> fst (18, 20)
18
Prelude> snd (18, 20)
20
```

```
type Student = (String, String, Int)
getID :: Student -> String
getID (name, id, mark) = id
```

```
Main> getID ("Mary", "s887123", 65)
"s887123"
Main>
```



```
addPair :: (Int, Int) -> Int
addPair (x, y) = x+y

min3, max3 :: Int -> Int -> Int -> Int
min3 x y z
   | x <= y && x <= z    = x
   | y <= z              = y
   | otherwise           = z

max3 x y z
   | x >= y && x >= z    = x
   | y >= z              = y
   | otherwise           = z


middle :: Int -> Int -> Int -> Int
middle x y z
   | between x y z       = x
   | between y x z       = y
   | otherwise           = z

between :: Int -> Int -> Int -> Bool
between x y z = (x >= y &&  x<=z) || (x >=z &&  x <= y)

orderTriple :: (Int, Int, Int) -> (Int, Int, Int)
orderTriple (x, y, z) = (min3 x y z, middle x y z,  max3 x y z)
```

```
Main> orderTriple (18, 12, 30)
(12,18,30)
Main> orderTriple (1,2,3)
(1,2,3)
Main> orderTriple (3,2,1)
(1,2,3)
Main> orderTriple (2,1,3)
(1,2,3)
Main> orderTriple (18, 120, 34)
(18,34,120)
```

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
```

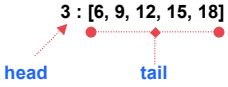
```
Main> matches 1  [2, 1, 3, 1, 1, 5]
[1,1,1]
Main> matches 5  [1,2,3]
[]
Main>
Main> is_there 1  [2, 1, 3, 1, 1, 5]
True
Main> is_there 1  [5, 6]
False
Main>
```

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
```

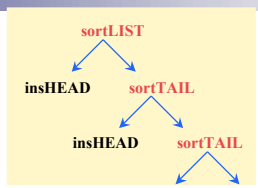
```
Main> elemN  1 [1,2,1,1,4,5,1]
4
Main> elemN1  1 [1,2,1,1,4,5,1]
4
Main> elemN  9 [1,2,1,1,4,5,1]
0
Main> elemN1 9 [1,2,1,1,4,5,1]
0
Main>
```



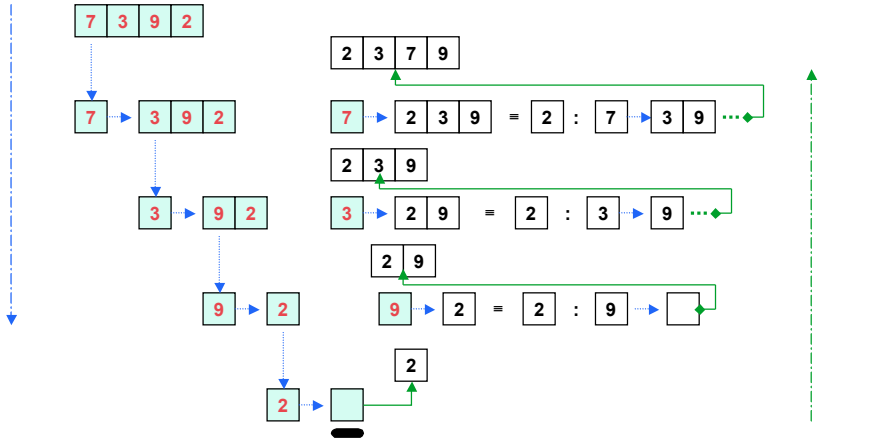
```
-- list of numbers that occur exactly once in a given list
uniqueIN :: [Int] -> [Int]
uniqueIN xs = [a | a <- xs, elemN a xs == 1]
```

```
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



-- where ins = *insert into correct place*

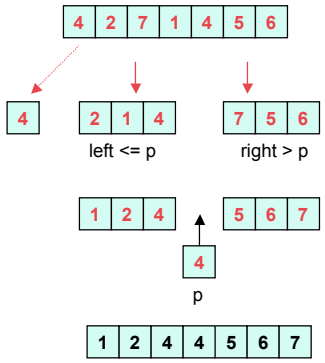



```
iSort :: [Int] -> [Int]
iSort  [ ]  =  [ ]
iSort (x : xs) = ins x (iSort xs)
```

```
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




```
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>
```