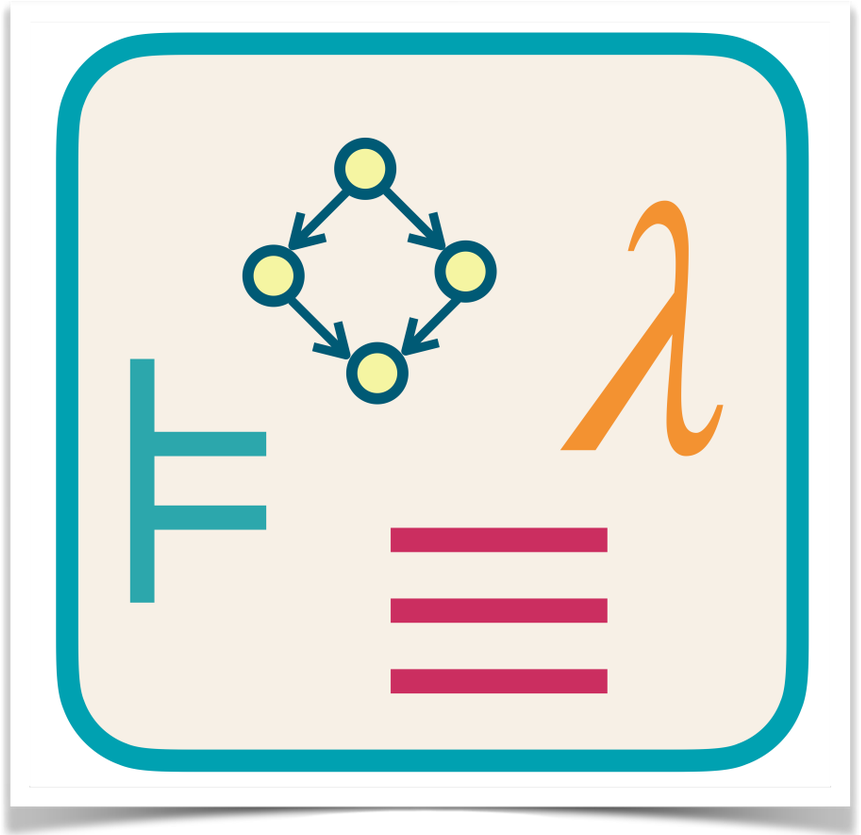


https://didawiki.di.unipi.it/doku.php/magistraleinformatica/mpp/start

MPP 2025/26 (0077A, 9CFU)

Models for Programming Paradigms

Roberto Bruni Filippo Bonchi
http://www.di.unipi.it/~bruni/

11b - Haskell

Haskell: a purely functional programming language

http://www.haskell.org/

Downloads Community Documentation

Haskell

An advanced, purely functional programming language

Declarative, statically typed code.

```
primes = filterPrime [2..]
  where filterPrime (p:xs) =
          p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

Try it!

```
Type Haskell expressions in here.
λ
```

Got 5 minutes?

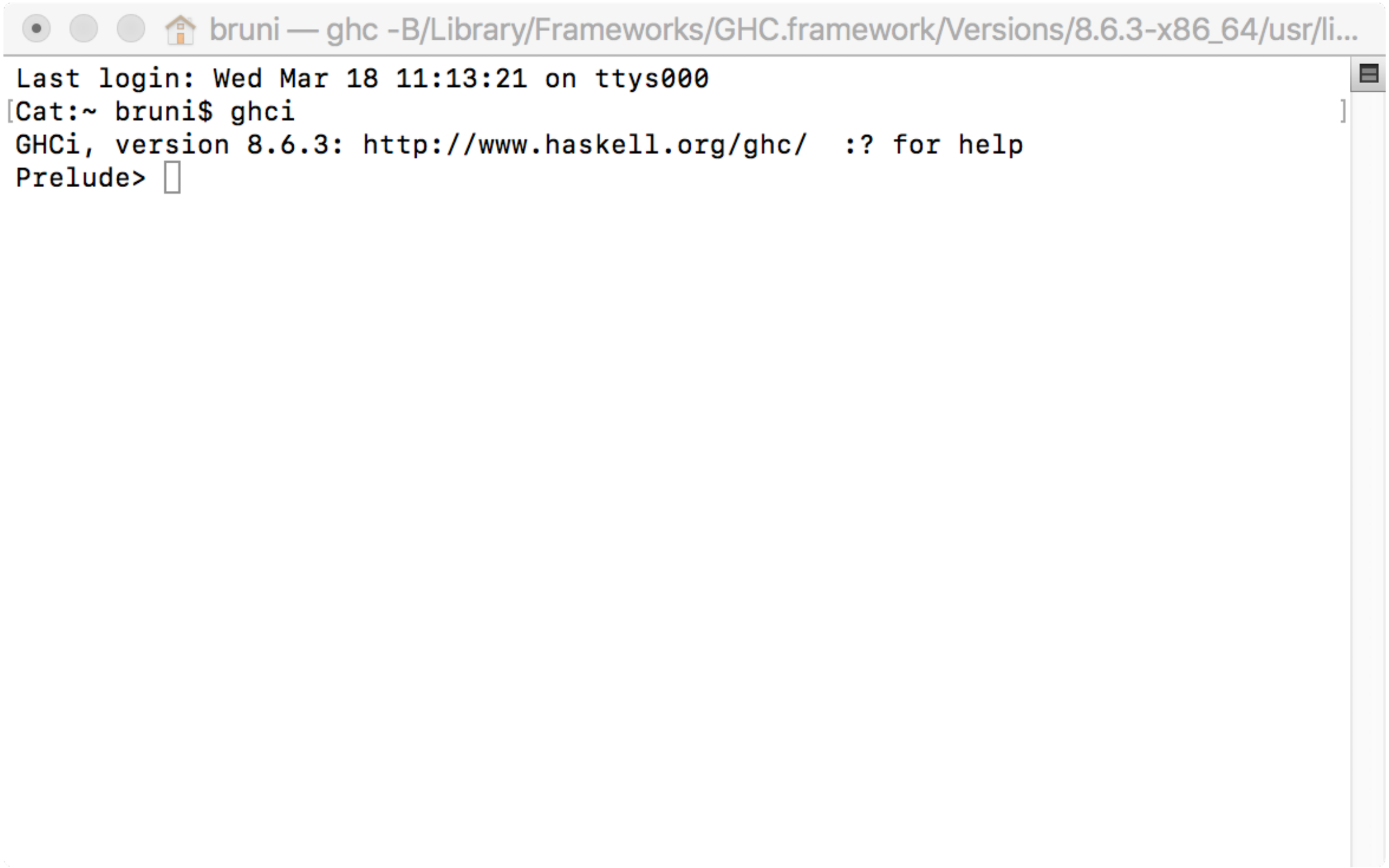
Type `help` to start the tutorial.

Or try typing these out and see what happens (click to insert):

`23 * 36` or `reverse "hello"` or `foldr (:) [] [1,2,3]` or `do line <- getLine; putStrLn line` or `readFile "/welcome"`

These IO actions are supported in this sandbox.

GHCi session



GHCi help

```
% ghci
GHCi, version 9.4.8: https://www.haskell.org/ghc/  :? for help
ghci> :?
 Commands available from the prompt:
                    [with many omissis]
   <statement>                  evaluate/run <statement>
   :                            repeat last command
   :help, :?                    display this list of commands
   :info [<name> ...]           display information about the given names
   :instances <type>            display the class instances available for <type>
   :kind <type>                 show the kind of <type>
   :quit                        exit GHCi
   :type <expr>                 show the type of <expr>

   :set <option> ...            set options
   :unset <option> ...          unset options
Options for ':set' and ':unset':
    +m            allow multiline commands
    +s            print timing/memory stats after each evaluation
    +t            print type after evaluation

   :show bindings               show the current bindings made at the prompt

The User's Guide has more information. An online copy can be found here:

  https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci.html
```

GHCi conditional

```
ghci> ghci> :set +t

ghci> ghci> :set +m

ghci> let max a b = if a>b then a else b
ghci|
max :: Ord a => a -> a -> a

ghci> max 3 4
4
it :: (Ord a, Num a) => a

ghci> :t (max 3)
(max 3) :: (Ord a, Num a) => a -> a
```

GHCi cases

```
ghci> let max a b
ghci|         | a>b = a
ghci|         | a<=b = b
ghci|
max :: Ord a => a -> a -> a
ghci> max 3 4
4
it :: (Ord a, Num a) => a

ghci> let max a b
ghci|         | a>b = a
ghci|         | b>a = b
ghci|
max :: Ord a => a -> a -> a
ghci> max 2 2
*** Exception: <interactive>:(70,5)-(72,17): Non-exhaustive patterns in
function max

ghci> let max a b
ghci|         | a>b = a
ghci|         | otherwise = b
ghci|
max :: Ord a => a -> a -> a
ghci> max 2 2
2
it :: (Ord a, Num a) => a
```

GHCi pattern matching

```
ghci> let empty [] = True
ghci|     empty (x:xs) = False
ghci|
empty :: [a] -> Bool
ghci> empty [1,2,3]
False
it :: Bool
```

```
{- "iterative" factorial -}
ghci> let fact n = product [1 .. n]
ghci|
fact :: (Num a, Enum a) => a -> a
ghci> fact 6
720
it :: (Num a, Enum a) => a
ghci> fact (-29)
1
it :: (Num a, Enum a) => a
```

GHCi recursion

```
ghci> let fac :: Integer -> Integer
ghci|     fac n = if n==0 then 1
ghci|              else if n>0 then n * fac(n-1)
ghci|              else 0
ghci|
fac :: Integer -> Integer
ghci> fac 6
720
it :: Integer
ghci> fac (-29)
0
it :: Integer
ghci> fac 1000
402387260077093773543702433923003985719374864210714632543799910429938512398629020592044208486969404800479988610197196058631666872994808558901323829669944590997424504087073759918823627727188732519779505950995276120874975462497043601418278094646496291056393887437886487337119181045825783647849977012476632889835955735432513185323958463075557409114262417474349347553428646576611667797396668820291207379143853719588249808126867838374559731746136085379534524221586593201928090878297308431392844403281231558611036976801357304216168747609675871348312025478589320767169132448426236131412508780208000261683151027341827977704784635868170164365024153691398281264810213092761244896359928705114964975419909342221566832572080821333186116811553615836546984046708975602900950537616475847728421889679646244945160765353408198901385442487984959953319 ... 000000000000000000000000000000000
it :: Integer
```

GHCi recursion

```
ghci> let fac :: Integer -> Integer
ghci|     fac 0 = 1
ghci|     fac n | n>0 = n * fac(n-1)
ghci|           | otherwise = 0
ghci|
fac :: Integer -> Integer
ghci> fac 1000
40238726007709377354370243392300398571937486421071463254379991042993851239862902059204420848696940480047998861019719605863166872994808558901323829669944590997424504087073759918823627727188732519779505950995276120874975462497043601418278094646496291056393887437886487337119181045825783647849977012476632889835955735432513185323958463075557409114262417474349347553428646576611667797396668820291207379143853719588249808126867838374559731746136085379534524221586593201928090878297308431392844403281231558611036976801357304216168747609675871348312025478589320767169132448426236131412508780208000261683151027341827977704784635868170164365024153691398281264810213092761244896359928705114964975419909342221566832572080821333186116811553615836546984046708975602900950537616475847728421889679646244945160765353408198901385442487984959953319 ... 000000000000000000000000000000000
it :: Integer
```

GHCi let-in

```
ghci> let cylinder r h = let sideArea = 2 * pi * r * h
ghci|                        topArea = pi * r^2
ghci|                    in sideArea + 2 * topArea
ghci|
cylinder :: Floating a => a -> a -> a
ghci> cylinder 3 4
131.94689145077132
it :: Floating a => a
```

Exercises

```
ghci> -- implement replicate :: Integer -> a -> [a]

ghci> let myrep n x
ghci|           | (n<=0) = []
ghci|           | otherwise = x : (myrep (n-1) x)
ghci|
myrep :: (Ord t1, Num t1) => t1 -> t2 -> [t2]
ghci> myrep 3 5
[5,5,5]
it :: Num t2 => [t2]
```

Exercises

```
ghci> -- implement take

ghci> let mytake n _ | (n<=0) = []
ghci|     mytake _ [] = []
ghci|     mytake n (x:xs) = x : (mytake (n-1) xs)
ghci|
mytake :: (Ord t, Num t) => t -> [a] -> [a]
ghci> mytake 10 [1..40]
[1,2,3,4,5,6,7,8,9,10]
it :: (Num a, Enum a) => [a]
```

Exercises

```
ghci> -- implement repeat

ghci> let myrep x = x : (myrep x)
ghci|
myrep :: t -> [t]
ghci> take 10 (myrep 4)
[4,4,4,4,4,4,4,4,4,4]
it :: Num a => [a]
```

Exercises

```
ghci> -- implement elem

ghci> let myelem x [] = False
ghci|     myelem x (y:xs) = (x==y) || (myelem x xs)
ghci|
myelem :: Eq t => t -> [t] -> Bool
ghci> myelem 6 [1..]
True
it :: Bool
```

Exercises

```
ghci> -- implement zip

ghci> let myzip _ [] = []
ghci|     myzip [] _ = []
ghci|     myzip (x:xs) (y:ys) = (x,y) : myzip xs ys
ghci|
myzip :: [a] -> [b] -> [(a, b)]
ghci> myzip [1..10] "a..d"
[(1,'a'),(2,'.'),(3,'.'),(4,'d')]
it :: (Num a, Enum a) => [(a, Char)]
```

Exercises

```
ghci> -- given a list checks if it is ordered

ghci> let sorted xs =  length [x | (x,y) <- zip xs (tail xs) , x>y] == 0
sorted :: Ord a => [a] -> Bool
ghci> sorted [1..12]
True
it :: Bool
ghci> sorted ([1..12] ++ [3])
False
it :: Bool
```

Exercises

```
ghci> -- get the list of all positions of one element x in a list

ghci> let pos x xs = [i | (y,i) <- zip xs [1..] , x==y]
ghci|
pos :: (Num a1, Enum a1, Eq a2) => a2 -> [a2] -> [a1]
ghci> pos 'e' "this is a sentence"
[12,15,18]
it :: (Num a1, Enum a1) => [a1]
```

Exercises

```
ghci> -- insert an element in a sorted list

ghci> let insert x [] = [x]
ghci|     insert x (y:xs)
ghci|             | x<y = x : y : xs
ghci|             | otherwise = y : (insert x xs)
ghci|
insert :: Ord t => t -> [t] -> [t]
ghci> insert 3 [1,2,4,5]
[1,2,3,4,5]
it :: (Ord t, Num t) => [t]
```

```
ghci> -- then, implement insertion sort

ghci> let isort [] = []
ghci|     isort (x:xs) = insert x (isort xs)
ghci|
isort :: Ord a => [a] -> [a]
ghci> isort [15,14..1]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
it :: (Ord a, Num a, Enum a) => [a]
```

Exercises

```
ghci> -- implement quick sort

ghci> let qsort [] = []
ghci|     qsort (x:xs) = smaller ++ [x] ++ greater
ghci|           where smaller = qsort [a | a <- xs , a<=x]
ghci|                 greater = qsort [a | a <- xs , a>x]
ghci|
qsort :: Ord a => [a] -> [a]
ghci> qsort ([50,48..1] ++ [1,3..49])
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,
30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50]
it :: (Ord a, Num a, Enum a) => [a]
```

Exercises

```
ghci> :t map
map :: (a -> b) -> [a] -> [b]

ghci> -- implement map

ghci> let myMap _ [] = []
ghci|     myMap f (x:xs) = (f x) : myMap f xs
ghci|
myMap :: (t -> a) -> [t] -> [a]
ghci> myMap (*2) [1..50]
[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50,52,54,5
6,58,60,62,64,66,68,70,72,74,76,78,80,82,84,86,88,90,92,94,96,98,100]
it :: (Num a, Enum a) => [a]
```

Exercises

```
ghci> :t filter
filter :: (a -> Bool) -> [a] -> [a]

ghci> -- implement filter

ghci> let myFilter _ [] = []
ghci|     myFilter p (x:xs) | p x = x : rest
ghci|                       | otherwise = rest
ghci|                      where rest = myFilter p xs
ghci|
myFilter :: (a -> Bool) -> [a] -> [a]
ghci> myFilter even [1..40]
[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40]
it :: Integral a => [a]
```

Exercises

```
ghci> -- implement the extraction of the factors of a number

ghci> let factors :: Integer -> [Integer]
ghci|     factors n = [ d | d <- [1..n] , n `mod` d==0  ]
ghci|
factors :: Integer -> [Integer]

ghci> let factors n = filter (\d -> n `mod` d==0)  [1..n]
ghci|
factors :: Integral a => a -> [a]
ghci> factors 36
[1,2,3,4,6,9,12,18,36]
it :: Integral a => [a]

ghci> -- then, checkif a number is prime
ghci> let prime n = (factors n) == [1,n]
ghci|
prime :: Integral a => a -> Bool
ghci> prime 7
True
it :: Bool
ghci> prime 27
False
it :: Bool
```

Exercises

```
ghci> -- generate the list of all prime numbers

ghci> let primes = filter prime (2:[3,5..])
ghci|
primes :: Integral a => [a]
ghci> take 30 primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103
,107,109,113]
it :: Integral a => [a]
```

Folds

```
ghci> -- sum [] = 0
ghci> -- sum (x:xs) = x + (sum xs)
ghci> -- sum [1,2,3] = 1 + (2 + (3 + 0))

ghci> -- prod [] = 1
ghci> -- prod (x:xs) = x * (prod xs)
ghci> -- prod [1,2,3] = 1 * (2 * (3 * 1))

ghci> -- and [] = True
ghci> -- and (x:xs) = x && (and xs)
ghci> -- and [True,False,True] = True && (False && (True && True))
```

```
ghci> -- the general schema
ghci> -- rop [] = v
ghci> -- rop (x:xs) = x op (rop xs)
ghci> --      [a,b,c] = a : (b : (c : []))
```

Folds

```
ghci> -- foldr :: (a -> b -> b) -> b -> [a] -> b
ghci> -- a -> b -> b: given the current element and the previous result
ghci> --              computes the next result
ghci> -- b: previous result
ghci> -- [a]: list of elements to traverse
ghci> -- b: final result

ghci> let foldr f acc [] = acc
ghci|     foldr f acc (x:xs) = x `f` (foldr f acc xs)
ghci|
foldr :: (t1 -> t2 -> t2) -> t2 -> [t1] -> t2
ghci> let sumr = foldr (+) 0
ghci|
sumr :: Num t2 => [t2] -> t2

ghci> let prodr = foldr (*) 1
ghci|
prodr :: Num t2 => [t2] -> t2

ghci> let andr = foldr (&&) True
ghci|
andr :: [Bool] -> Bool
```

Folds

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

ghci> let foldl f acc [] = acc
ghci|     foldl f acc (x:xs) = let acc' = acc `f` x
ghci|                          in foldl f acc' xs
ghci|
foldl :: (t -> a -> t) -> t -> [a] -> t
ghci> --      [1,2,3] = 1 : (2 : (3 : []))
ghci> -- sum [1,2,3] = ((0 + 1) + 2) + 3
ghci>
ghci> let suml = foldl (+) 0
ghci|
suml :: Num a => [a] -> a
```

Folds

```
ghci> -- foldr :: (a -> [a] -> [a]) -> [a] -> [a] -> [a]
ghci> let elemr y = foldr (\x acc -> (x==y) || acc) False
ghci|
elemr :: Eq a => a -> [a] -> Bool

ghci> -- foldr :: (a -> [a] -> [a]) -> [a] -> [a] -> [a]
ghci> let filterr p = foldr (\x acc -> if p x then x : acc else acc) []
ghci|
filterr :: (a -> Bool) -> [a] -> [a]
ghci> filterr (even) [1..10]
[2,4,6,8,10]
it :: Integral a => [a]

ghci> -- foldl :: (b -> a -> b) -> b -> [a] -> b
ghci> let filterl p = foldl (\acc x -> if p x then x : acc else acc) []
ghci|
filterl :: (a -> Bool) -> [a] -> [a]
ghci> filterl (even) [1..10]
[10,8,6,4,2]
it :: Integral a => [a]

ghci> let reversel = foldl (flip (:)) []
ghci|
reversel :: [a] -> [a]
ghci> reversel [1..10]
[10,9,8,7,6,5,4,3,2,1]
it :: (Num a, Enum a) => [a]
```

Fix

```
ghci> -- implement fix :: (t -> t) -> t

ghci> let fix f = f (fix f)
ghci|
fix :: (t -> t) -> t

ghci> let fix f = (let x = f x in x)
ghci|
fix :: (t -> t) -> t
```

Fix: factorial

```
ghci> let fac = \n -> if n<=0 then 1 else n * fac(n-1)
ghci|
fac :: (Ord t, Num t) => t -> t
ghci> let fac = (\f n -> if n<=0 then 1 else n * f(n-1)) fac
ghci|
fac :: (Ord t, Num t) => t -> t
ghci> let gamma_fac = (\f n -> if n<=0 then 1 else n * f(n-1))
ghci|
gamma :: (Ord t, Num t) => (t -> t) -> t -> t
ghci> -- obviously, fac = gamma_fac fac

ghci> gamma_fac succ 3
9
it :: (Ord t, Num t, Enum t) => t
ghci> gamma_fac (2*) 3
12
it :: (Ord t, Num t) => t
ghci> let factorial = fix gamma_fac
ghci|
fact :: (Ord t, Num t) => t -> t
ghci> factorial 3
6
it :: (Ord t, Num t) => t
ghci> gamma_fac factorial 3
6
it :: (Ord t, Num t) => t
```

Fix: fibonacci

```
ghci> let fib = \n -> if n<2 then n else fib(n-1) + fib(n-2))
ghci|
fac :: (Ord t, Num t) => t -> t
ghci> let fib = (\f n -> if n<2 then n else f(n-1) + f(n-2)) fib
ghci|
fac :: (Ord t, Num t) => t -> t
ghci> let gamma_fib = (\f n -> if n<2 then n else f(n-1) + f(n-2))
ghci|
gamma :: (Ord t, Num t) => (t -> t) -> t -> t
ghci> -- obviously, fib = gamma_fib fib

ghci> gamma_fib succ 8
15
it :: (Ord t, Num t, Enum t) => t
ghci> gamma_fib (2*) 8
26
it :: (Ord t, Num t) => t
ghci> let fibonacci = fix gamma_fib
ghci|
fact :: (Ord t, Num t) => t -> t
ghci> fibonacci 8
21
it :: (Ord t, Num t) => t
ghci> gamma_fib fibonacci 8
21
it :: (Ord t, Num t) => t
```

Fix: triangular numbers

```
ghci> -- triangular n = sum [1 .. n] = (n * (n+1)) / 2
ghci> -- for which gamma we can set
ghci> -- let triangular = fix gamma?
ghci> let gamma = (\f n -> ...)

ghci> let gamma = (\f n -> if n<=0 then 0 else n + f(n-1))
ghci|
gamma :: (Ord t, Num t) => (t -> t) -> t -> t

ghci> gamma succ 5
10
it :: (Ord t, Num t, Enum t) => t
ghci> gamma (2*) 5
13
it :: (Ord t, Num t) => t
ghci> let triangular = fix gamma
ghci|
fact :: (Ord t, Num t) => t -> t
ghci> triangular 5
15
it :: (Ord t, Num t) => t
ghci> gamma triangular 5
15
it :: (Ord t, Num t) => t
```

Function application

```
ghci> -- function application: $ :: (a -> b) -> a -> b
ghci> -- f $ x = f x
ghci> -- avoid brackets!
ghci> -- f (g (h z)) => f $ g $ h z

ghci> sum (filter (> 10) (map (*2) [1..10]))
80
it :: (Num a, Ord a, Enum a) => a

ghci> sum $ filter (> 10) $ map (*2) [1..10]
80
it :: (Num a, Ord a, Enum a) => a
```

```
ghci> map ($ 5) [(5+), (5*),(^5)]
[10,25,3125]
it :: Num b => [b]
```

Function application

```
ghci> -- function composition: . :: (b -> c) -> (a -> b) -> a -> c
ghci> -- (g . f) x = g (f x)
ghci> -- g . f = \x -> g(f x)

ghci> map (\x -> negate(abs(x))) [-10..10]
[-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,-1,-2,-3,-4,-5,-6,-7,-8,-9,-10]
it :: (Num b, Enum b) => [b]

ghci> map (negate . abs) [-10..10]
[-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,-1,-2,-3,-4,-5,-6,-7,-8,-9,-10]
it :: (Num b, Enum b) => [b]
```

```
ghci> sum (filter odd (map (^2) [1..100]))
166650
it :: Integral a => a
ghci> sum . filter odd . map (^2) $ [1..100]
166650
it :: Integral a => a
ghci> sum . filter odd . map (^2) [1..100]
<interactive>:69:20: error:
    • Couldn't match expected type: a -> [c]
                  with actual type: [b0]
    • Possible cause: ‘map’ is applied to too many arguments
      ...
ghci> :q
Leaving GHCi.
%
```