

MPP 2026/27 (0077A, 9CFU)

Models for Programming Paradigms

Roberto Bruni

Filippo Bonchi

<http://www.di.unipi.it/~bruni/>



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

IIa - Haskell

Lambda notation, again

Bound variables

```
int f(int x) { return x^2 + 2*x + 5 }  
int f(int y) { return y^2 + 2*y + 5 }
```

$$f \triangleq \lambda x. x^2 + 2x + 5$$
$$f \triangleq \lambda y. y^2 + 2y + 5$$

```
let f x = x^2 + 2*x + 5  
let f y = y^2 + 2*y + 5
```

they are all the same!

local names of bound variables are not important:

it's the principle of **alpha-conversion**

bound names can be renamed as we like

Free variables

$$x^2 + 2x + 5$$

$$y^2 + 2y + 5$$

they are not the same!

names of global variables matter

$$\lambda x. x^2 + 2x + 5$$

$$\lambda x. y^2 + 2y + 5$$

the same enclosing context
can make a difference



$$\lambda x. t$$

we say it **binds the free occurrences** of x in t

$$\lambda x. x^2 + 2z + 5$$

$$\lambda y. y^2 + 2z + 5$$

$$\lambda z. z^2 + 2z + 5$$

are they all equivalent
(by alpha-conversion)?

Free variables: formally

$$\text{fv}(\lambda x. x^2 + 2z + 5) = \{z\}$$

$$\text{fv}(\lambda y. y^2 + 2z + 5) = \{z\}$$

$$\text{fv}(\lambda z. z^2 + 2z + 5) = \emptyset$$

$$t ::= x \mid \lambda x. t \mid t t \mid \dots$$

$$\text{fv} : LTerms \rightarrow \wp(Var)$$

$$\begin{aligned} \text{fv}(x) &\triangleq \{x\} \\ \text{fv}(\lambda x. t) &\triangleq \text{fv}(t) \setminus \{x\} \\ \text{fv}(t_1 t_2) &\triangleq \text{fv}(t_1) \cup \text{fv}(t_2) \end{aligned}$$

Alpha-conversion, again

$$\lambda x. t \equiv \lambda y. (t[y/x]) \quad \text{if } y \notin \text{fv}(\lambda x. t)$$

$$\lambda x. x^2 + 2z + 5 \equiv \lambda y. ((x^2 + 2z + 5)[y/x]) = \lambda y. y^2 + 2z + 5$$

$$\lambda x. x^2 + 2z + 5 \not\equiv \lambda z. ((x^2 + 2z + 5)[z/x]) \text{ because } z \in \text{fv}(\lambda x. x^2 + 2z + 5)$$

Beta rule, again

$$(\lambda x. t) e \equiv t[e/x]$$

we cannot just use plain syntactic substitution that would replace every free occurrence of x with e
we need to use **capture-avoiding substitutions**

how is (capture-avoiding) substitution defined?
and why is it called “capture-avoiding”?

Capture-avoiding substitution

Substitution, 1st try

$$y[e/x] \triangleq \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases}$$

$$(t_1 \ t_2)[e/x] \triangleq t_1[e/x] \ (t_2[e/x])$$

$$(\lambda y. t)[e/x] \triangleq \begin{cases} \lambda y. t & \text{if } y = x \\ \lambda y. (t[e/x]) & \text{otherwise} \end{cases}$$

$$t_1 \triangleq \lambda x. \lambda y. x^2 + 2y + 5$$


$$t_2 \triangleq y$$

$$t_1 \ t_2 \equiv (\lambda x. \lambda y. x^2 + 2y + 5) \textcolor{blue}{y}$$


free

$$\equiv (\lambda y. x^2 + 2y + 5)[\textcolor{blue}{y}/x]$$

$$\equiv \lambda y. ((x^2 + 2y + 5)[\textcolor{blue}{y}/x])$$

$$\equiv \lambda y. \textcolor{blue}{y}^2 + 2y + 5$$


captured variable!

Capture-avoiding

free variables occurring in e
should remain free after the application of $[^e/x]$

solution: **alpha-convert bound names before substituting!**

$$\begin{aligned}(\lambda y. x^2 + 2y + 5)[^y/x] &\equiv (\lambda z. (x^2 + 2y + 5)[^z/y])[^y/x] \\&\equiv (\lambda z. x^2 + 2z + 5)[^y/x] \\&\equiv \lambda z. ((x^2 + 2z + 5)[^y/x]) \\&\equiv \lambda z. y^2 + 2z + 5\end{aligned}$$

↖ free

Substitution, 2nd try

$$y[e/x] \triangleq \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases}$$

$$(t_1 \ t_2)[e/x] \triangleq t_1[e/x] \ (t_2[e/x])$$

~~$$(\lambda y. t)[e/x] \triangleq \begin{cases} \lambda y. t & \text{if } y = x \\ \lambda y. (t[e/x]) & \text{otherwise} \end{cases}$$~~

$$(\lambda y. t)[e/x] \triangleq \begin{cases} \lambda y. t & \text{if } y = x \\ \lambda z. (t[z/y][e/x]) & \text{otherwise, with} \\ & z \notin \text{fv}(e) \cup \text{fv}(\lambda y. t) \cup \{x\} \end{cases}$$

superfluous: no free occurrences to replace

Substitution, final

$$y[e/x] \triangleq \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases}$$

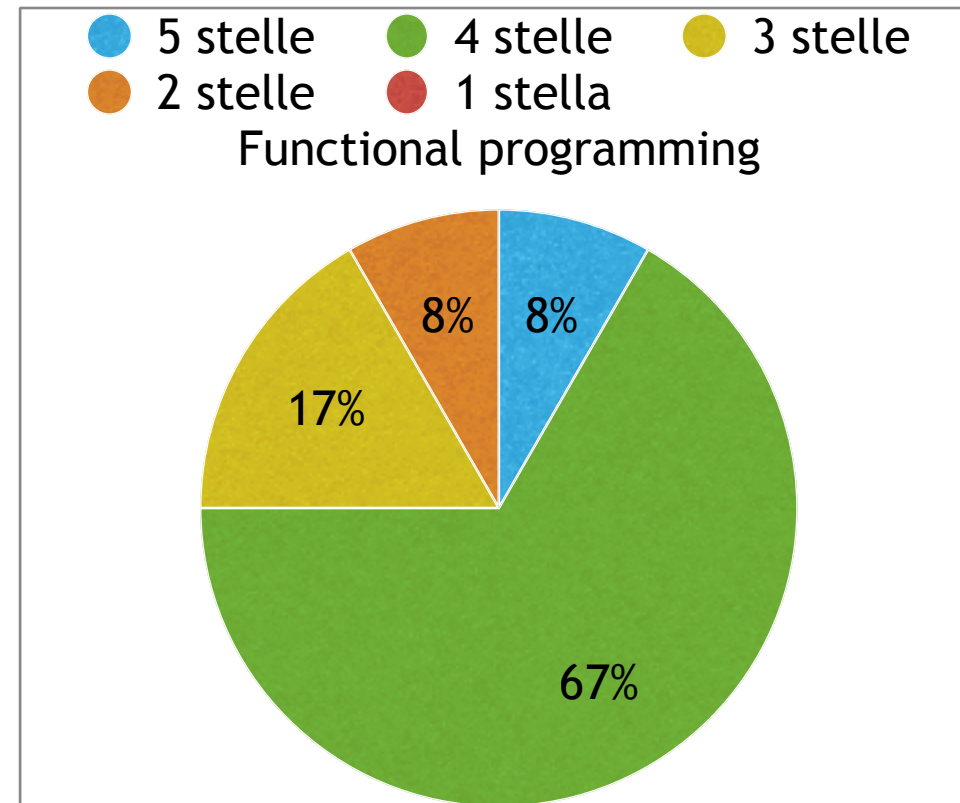
$$(t_1 \ t_2)[e/x] \triangleq t_1[e/x] \ (t_2[e/x])$$

$$(\lambda y. \ t)[e/x] \triangleq \lambda z. \ (t[z/y][e/x]) \quad \text{with } z \notin \text{fv}(e) \cup \text{fv}(\lambda y. \ t) \cup \{x\}$$

Higher Order Functional Languages

Haskell

From your forms



(over 12 answers)

Imperative vs Functional

Imperative style

tell the machine **how** to compute;
a sequence of tasks to execute;
manipulation of mutable states

Purely functional
style

tell the machine **what** to compute;
declarative style;
define what functions are,
not how to compute them;
functions have no side effects;
can't set and change variable's content;
manipulation of values

Declarative style

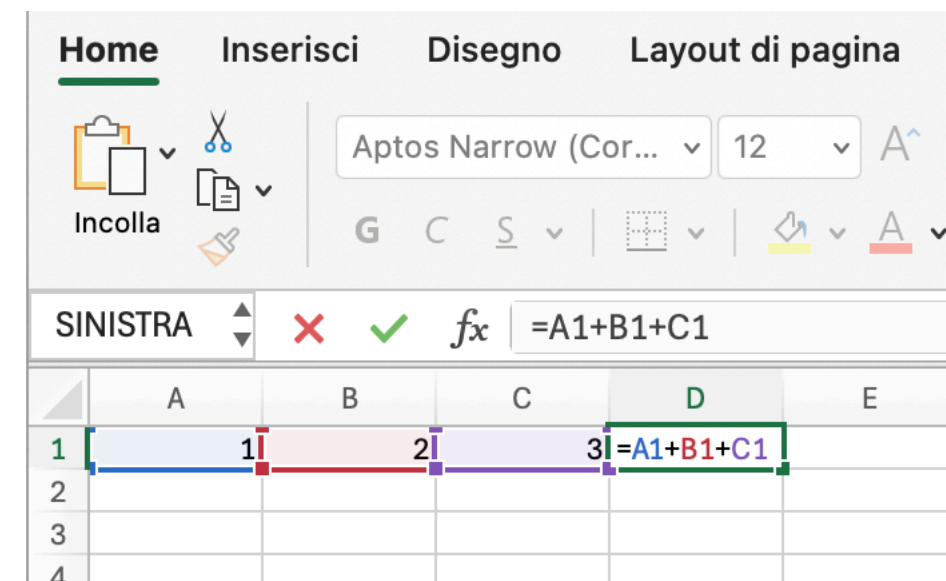
Any experience of functional programming?

Have you ever used a spreadsheet?

The value of a cell is defined in terms of those of other cells:
what is to be computed, not how it must be computed

we do not specify the order in which cells are calculated:
values are computed according to their dependencies

we do not decide how to allocate memory:
only cells in use are allocated



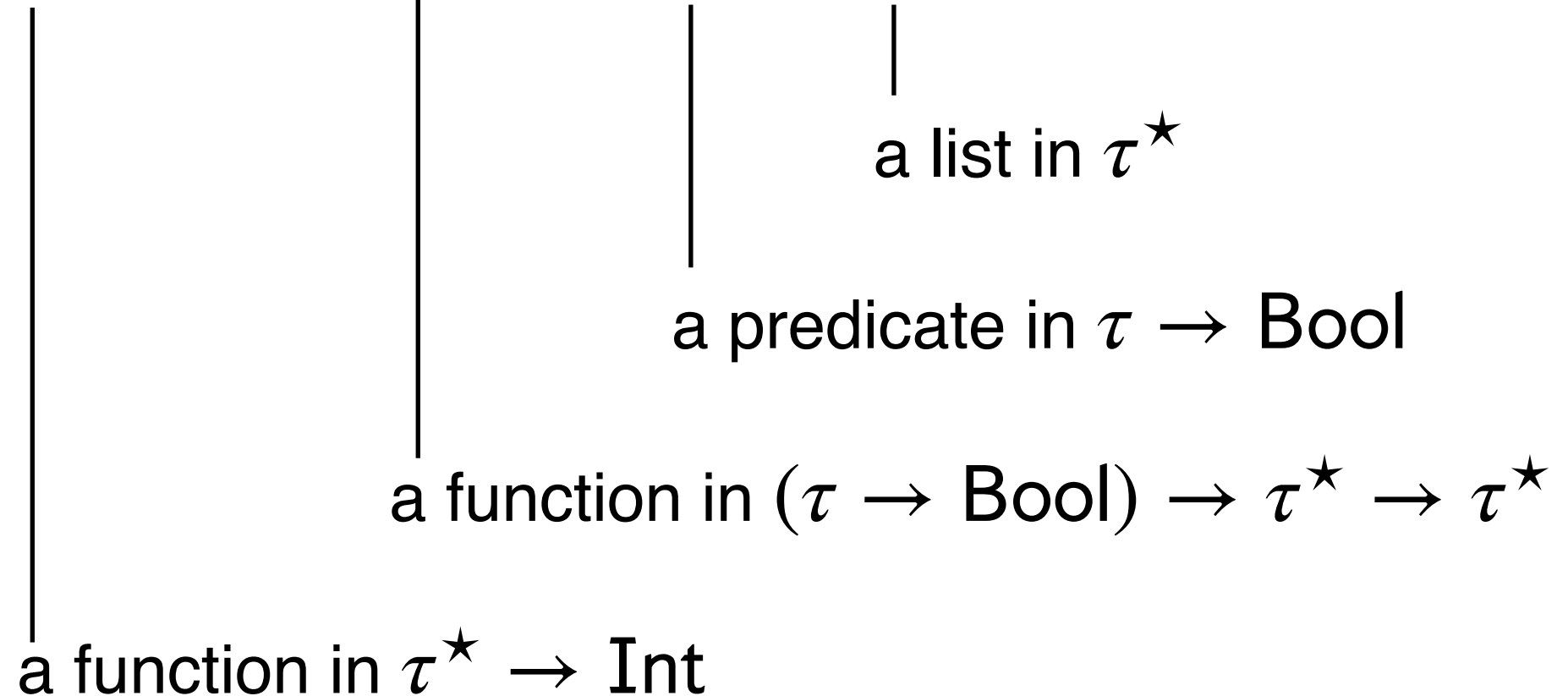
Functional style: HO

Higher-Order:

functions as values,
functions as parameters,
functions are returned,
functions are composed

how many elements of a
list will pass the test?

length (filter test xs)



Functional style: HO

Higher-Order:

functions as values,

functions as parameters,

functions are returned,

functions are composed

how many elements of a
list will pass the test?

`length (filter test xs)`

a list in $\tau^\star$

a predicate in $\tau \rightarrow \text{Bool}$

a function in $(\tau \rightarrow \text{Bool}) \rightarrow \tau^\star \rightarrow \tau^\star$

a function in $\tau^\star \rightarrow \text{Int}$

Functional style: HO

Higher-Order:

functions as values,
functions as parameters,
functions are returned,
functions are composed

how many elements of a
list will pass the test?

`length (filter test xs)`

a list in $\tau^\star$

a predicate in $\tau \rightarrow \text{Bool}$

a function in $(\tau \rightarrow \text{Bool}) \rightarrow \tau^\star \rightarrow \tau^\star$

a function in $\tau^\star \rightarrow \text{Int}$

Purity: no side effects

the result of a function is determined only by its input

a variable is just a name bound to some (HO) value:
shorthands for expressions

variables do not vary

programs are typically shorter, maybe less efficient;
closer to semantics, ease verification of correctness;
more robust, easier to maintain

Haskell: a purely functional programming language

<http://www.haskell.org/>

[Downloads](#)

[Community](#)

[Documentation](#)



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.

Haskell: origins

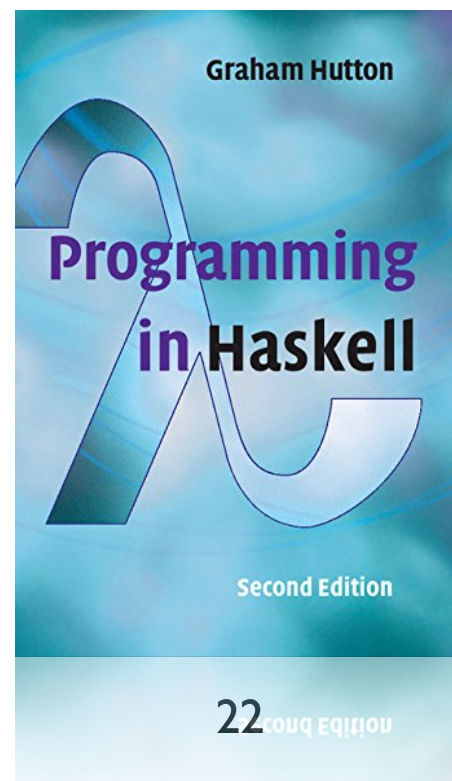
named after mathematical logician Haskell B. Curry

1987: Haskell project begun

1998: first version appear

2003: the Haskell Report was published
(first stable version)

Graham Hutton, “Programming in Haskell”, ch.1-8,14,15



Features

- Referential transparency** if a function is called twice with the same argument, it returns the same result;
compiler can reason on program's behaviour;
one can deduce a function is correct and build more complex functions by composition
- Statically typed** **type inference**: you don't have to label all data, their types can be figured out;
many possible errors are caught at compile time
- Polymorphism** one definition of function works for many types
- Overloading** different definitions of the same function-name for different types
- Laziness** **calculation starts only if some result is needed**;
infinite data structures can be manipulated

I'M NOT LAZY



**I'M JUST SAVING MY
ENERGY FOR WHEN I
REALLY NEED IT.**

More features (less bugs)

Purity: no side effects

Typeful: types are pervasive, no dubious use of types

Concise: shorter programs, less typing (on the keyboard)

High level: closer to the algorithm description

Memory managed: programmers can focus on the algorithm

Compositionality: solve problems by composing functions that solve smaller problems

Modules and type classes: data encapsulation and polymorphism not exclusive to object oriented programming

A taste of Haskell

math. notation

$$f(x) = 2x + 3$$

$$g(x, y) = x^2 + xy + y^2$$

$$abs(x) = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise} \end{cases}$$

$$abs(f(g(2, 3)))$$

set comprehension

$$\{x \mid x \in X \wedge f(x) > 5\}$$

Haskell notation

$$f \ x = 2 * x + 3$$

$$g \ (x, y) = x^2 + x * y + y^2$$

$$\begin{array}{lcl} abs \ x & & \\ | & x \geq 0 & = \ x \\ | & otherwise & = -x \end{array}$$

$$abs \ (f \ (g \ (2, 3)))$$

list comprehension

$$[\ x \mid x \leftarrow X, \ f \ x > 5 \]$$

The power of recursion

No assignments: no loops

(loops over lists exist: *list comprehension*)

Recursion is used in place of loops

```
power2 n
| n==0 = 1
| n>0  = 2 * power2 (n-1)
```

Haskell: some principles

evaluate *expressions* (syntactic terms)

to yield *values* (abstract entities regarded as answers)

every value has an associated *type*

the association is called *typing*

you can think of types as sets of values

as expressions denote values

types are denoted by type expressions

values are first-class (passed around, returned as results)

types are not first-class

Haskell: GHCi

Interactive shell or interpreter, executing read-eval-print loop

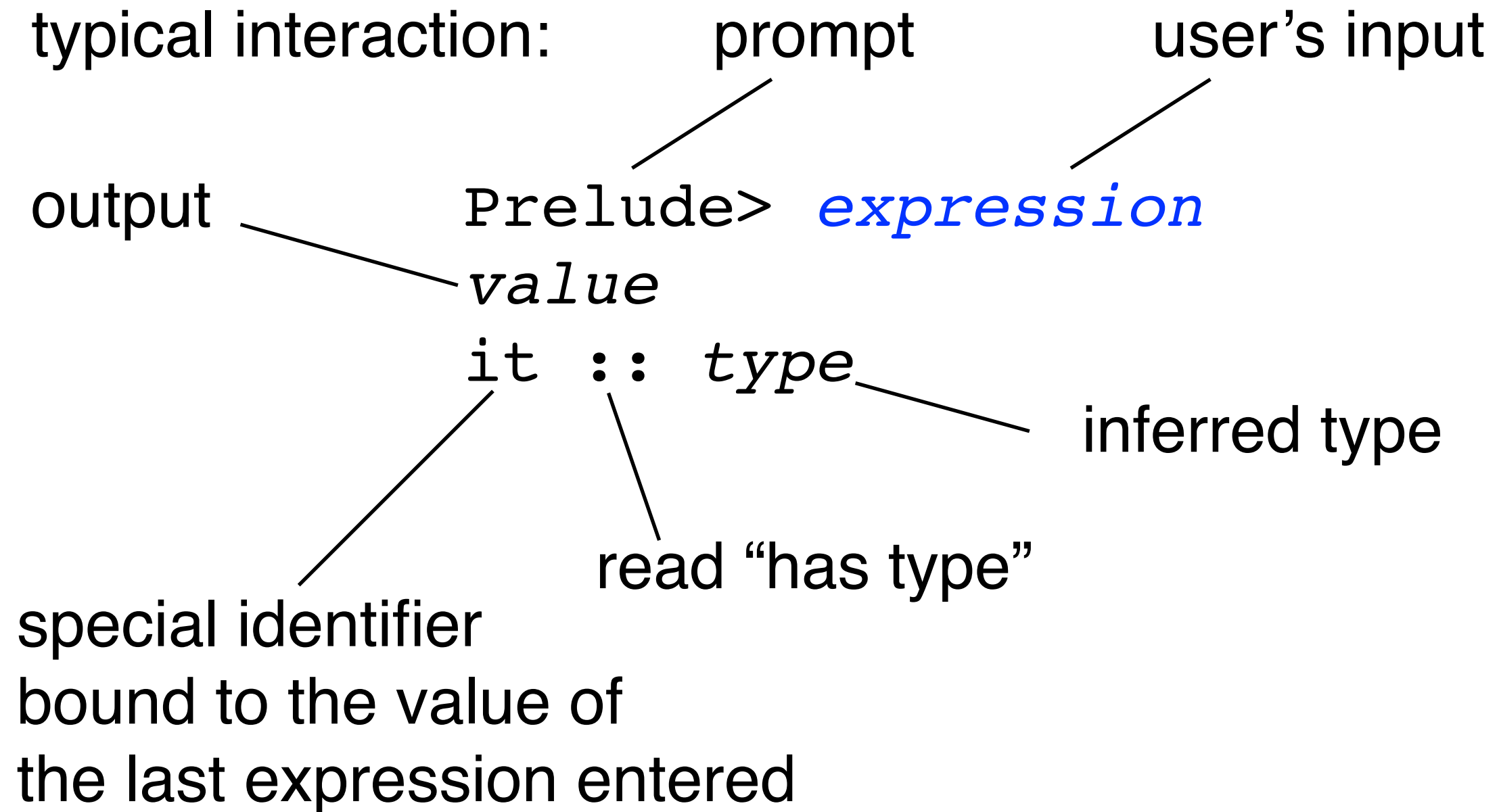
programmers enter expressions/declarations one at a time

they are type checked, compiled and executed

if an expression does not parse correctly
or does not pass the type-checking phase of the compiler,
no code is generated and no code is executed

once an identifier is defined it is available at subsequent lines

GHCi expressions



GHCi declarations

typical interaction:

keyword
Prelude> *let id = expression*
id :: type
defining symbol

GHCi declarations

more generally:

function name

formal parameters

```
Prelude> let id arguments = expression  
id :: argtype -> restype
```

arguments types

result type

The diagram illustrates the structure of a GHCi declaration. It shows a code snippet with four labels and arrows pointing to specific parts: 'function name' points to 'id', 'formal parameters' points to 'arguments', 'arguments types' points to 'argtype', and 'result type' points to 'restype'.

GHCi session

A terminal window titled 'bruni — ghc -B/Library/Frameworks/GHC.framework/Versions/8.6.3-x86_64/usr/li...'. The window shows the output of a login session and the start of a GHCi session. The text inside the terminal is: 'Last login: Wed Mar 18 11:13:21 on ttys000', '[Cat:~ bruni\$ ghci]', 'GHCi, version 8.6.3: http://www.haskell.org/ghc/ :? for help', and 'Prelude>'. The cursor is at the end of the 'Prelude>' line.

```
bruni — ghc -B/Library/Frameworks/GHC.framework/Versions/8.6.3-x86_64/usr/li...
Last login: Wed Mar 18 11:13:21 on ttys000
[Cat:~ bruni$ ghci
GHCi, version 8.6.3: http://www.haskell.org/ghc/ :? for help
Prelude> ]
```

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 basics

```
ghci> -- a comment
```

```
ghci> {- a block -}
```

```
ghci> :set +t
```

```
ghci> 1 + 2 ^ 3 * 4
```

```
33
```

```
it :: Num a => a
```

```
ghci> 1 + (2 ^ 3) * 4
```

```
33
```

```
it :: Num a => a
```

```
ghci> 1 + 2 ^ (3 * 4)
```

```
4097
```

```
it :: Num a => a
```

```
ghci> 3 ^ 3 ^ 3
```

```
7625597484987
```

```
it :: Num a => a
```

```
ghci> 3 ^ (3 ^ 3)
```

```
7625597484987
```

```
it :: Num a => a
```

```
ghci> (3 ^ 3) ^ 3
```

```
19683
```

```
it :: Num a => a
```

GHCi basics

```
ghci> it + 10
```

```
19693
```

```
it :: Num a => a
```

```
ghci> 7 / 2
```

```
3.5
```

```
it :: Fractional a => a
```

```
ghci> div 7 2
```

```
3
```

```
it :: Integral a => a
```

```
ghci> 7 `div` 2
```

```
3
```

```
it :: Integral a => a
```

```
ghci> 4 * 5
```

```
20
```

```
it :: Num a => a
```

```
ghci> (*) 4 5
```

```
20
```

```
it :: Num a => a
```

```
ghci> (*) 4
```

```
<interactive>:16:1: error:
```

- No instance for (Show (Integer -> Integer))
arising from a use of 'print'
(maybe you haven't applied a function to enough arguments?)
- In a stmt of an interactive GHCi command: print it

GHCI Bool

```
ghci> 2 == 3
```

```
False
```

```
it :: Bool
```

```
ghci> 2 < 3 && 4 < 3
```

```
False
```

```
it :: Bool
```

```
ghci> 2 < 3 || 4 < 3
```

```
True
```

```
it :: Bool
```

```
ghci> not (2<3)
```

```
False
```

```
it :: Bool
```

```
ghci> odd 4
```

```
False
```

```
it :: Bool
```

```
ghci> even 4
```

```
True
```

```
it :: Bool
```

GHCi session

```
ghci> succ 5
```

```
6
```

```
it :: (Enum a, Num a) => a
```

```
ghci> succ False
```

```
True
```

```
it :: Bool
```

```
ghci> pred 5
```

```
4
```

```
it :: (Enum a, Num a) => a
```

```
ghci> pred True
```

```
False
```

```
it :: Bool
```

```
ghci> min 3 4
```

```
3
```

```
it :: (Ord a, Num a) => a
```

```
ghci> 3 `min` 4
```

```
3
```

```
it :: (Ord a, Num a) => a
```

```
ghci> if 2 < 3 then 1 else 2
```

```
1
```

```
it :: Num a => a
```

GHCI let

```
ghci> let x = 1 + 2 ^ 3 * 4
```

```
x :: Num a => a
```

```
ghci> x
```

```
33
```

```
it :: Num a => a
```

```
ghci> x + x
```

```
66
```

```
it :: Num a => a
```

```
ghci> x
```

```
33
```

```
it :: Num a => a
```

GHCi partial application

```
ghci> :t (*)
(*) :: Num a => a -> a -> a
ghci> :t (*) 4
(*) 4 :: Num a => a -> a
ghci> :t (*4)
(*4) :: Num a => a -> a
ghci> :t (/4)
(/4) :: Fractional a => a -> a
ghci> :t (4/)
(4/) :: Fractional a => a -> a
```

```
ghci> let twice = (2*)
twice :: Num a => a -> a
ghci> twice 3
6
it :: Num a => a
```

```
ghci> (1,2) -- pairing
(1,2)
it :: (Num a, Num b) => (a, b)
ghci> (1,True,'a') -- tupling
(1,True,'a')
it :: Num a => (a, Bool, Char)
```

GHCi lists

```
ghci> [1,2,3,4]
[1,2,3,4]
it :: Num a => [a]
ghci> []
[]
it :: [a]
```

```
ghci> :t (++)
(++) :: [a] -> [a] -> [a]
ghci> [1,2] ++ [3,4]
[1,2,3,4]
it :: Num a => [a]
```

```
ghci> :t (:)
(:) :: a -> [a] -> [a]
ghci> 1 : [2,3,4]
[1,2,3,4]
it :: Num a => [a]
```

```
ghci> [1,2,3,4] == 1:2:3:4:[]
True
it :: Bool
ghci> [] /= [ [] ]
True
it :: Bool
```

GHCI list ops

```
ghci> :t length
```

```
length :: Foldable t => t a -> Int
```

```
ghci> length [1,2,3]
```

```
3
```

```
it :: Int
```

```
ghci> [1,2,3,4] <= [1,2,3,4,5]
```

```
True
```

```
it :: Bool
```

```
ghci> :t elem
```

```
elem :: (Foldable t, Eq a) => a -> t a -> Bool
```

```
ghci> elem 3 [1,2,3,4]
```

```
True
```

```
it :: Bool
```

```
ghci> elem 6 [1,2,3,4]
```

```
False
```

```
it :: Bool
```

```
ghci> 6 `elem` [1,2,3]
```

```
False
```

```
it :: Bool
```

GHCi list ops

```
ghci> :t head
```

```
head :: GHC.Stack.Types.HasCallStack => [a] -> a
```

```
ghci> head [1,2,3]
```

```
1
```

```
it :: Num a => a
```

```
ghci> :t tail
```

```
tail :: GHC.Stack.Types.HasCallStack => [a] -> [a]
```

```
ghci> tail [1,2,3]
```

```
[2,3]
```

```
it :: Num a => [a]
```

```
ghci> :t last
```

```
head :: GHC.Stack.Types.HasCallStack => [a] -> a
```

```
ghci> last [1,2,3,4]
```

```
4
```

```
it :: Num a => a
```

```
ghci> :t init
```

```
tail :: GHC.Stack.Types.HasCallStack => [a] -> [a]
```

```
ghci> init [1,2,3,4]
```

```
[1,2,3]
```

```
it :: Num a => [a]
```

GHCI list ops

```
ghci> :t minimum
```

```
minimum :: (Foldable t, Ord a) => t a -> a
```

```
ghci> minimum [3,2,1,4]
```

```
1
```

```
it :: (Ord a, Num a) => a
```

```
ghci> :t maximum
```

```
maximum :: (Foldable t, Ord a) => t a -> a
```

```
ghci> maximum [3,4,2,1]
```

```
4
```

```
it :: (Ord a, Num a) => a
```

```
ghci> :t sum
```

```
sum :: (Foldable t, Num a) => t a -> a
```

```
ghci> sum [1,2,3,4]
```

```
10
```

```
it :: Num a => a
```

```
ghci> :t product
```

```
product :: (Foldable t, Num a) => t a -> a
```

```
ghci> product [1,2,3,4]
```

```
24
```

```
it :: Num a => a
```

GHCI list ranges

```
ghci> [1 .. 99]
```

```
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99]
```

```
it :: (Num a, Enum a) => [a]
```

```
ghci> ['a' .. 'z']
```

```
"abcdefghijklmnopqrstuvwxyz"
```

```
it :: [Char]
```

```
ghci> [1,3 .. 99]
```

```
[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47,49,51,53,55,57,59,61,63,65,67,69,71,73,75,77,79,81,83,85,87,89,91,93,95,97,99]
```

```
it :: (Num a, Enum a) => [a]
```

```
ghci> [100, 98 .. 1]
```

```
[100,98,96,94,92,90,88,86,84,82,80,78,76,74,72,70,68,66,64,62,60,58,56,54,52,50,48,46,44,42,40,38,36,34,32,30,28,26,24,22,20,18,16,14,12,10,8,6,4,2]
```

```
it :: (Num a, Enum a) => [a]
```

```
ghci> [100, 99, 97 .. 1]
```

```
<interactive>:55:14: error: parse error on input '..'
```

GHCi list ops

```
ghci> :t take
```

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

```
ghci> take 12 [0 .. ]
```

```
[0,1,2,3,4,5,6,7,8,9,10,11]
```

```
it :: (Num a, Enum a) => [a]
```

```
ghci> :t drop
```

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

```
ghci> drop 12 [0 .. 20]
```

```
[12,13,14,15,16,17,18,19,20]
```

```
it :: (Num a, Enum a) => [a]
```

```
ghci> :t cycle
```

```
cycle :: GHC.Stack.Types.HasCallStack => [a] -> [a]
```

```
ghci> take 20 (cycle [1,2,3])
```

```
[1,2,3,1,2,3,1,2,3,1,2,3,1,2,3,1,2,3,1,2]
```

```
it :: Num a => [a]
```

```
ghci> :t repeat
```

```
repeat :: a -> [a]
```

```
ghci> take 20 (repeat 2)
```

```
[2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2]
```

```
it :: Num a => [a]
```

```
ghci> :t replicate
```

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

```
ghci> replicate 10 3
```

```
[3,3,3,3,3,3,3,3,3,3]
```

```
it :: Num a => [a]
```

GHCI list comprehension

```
ghci> [x^2 | x <- [1..10]]  
[1,4,9,16,25,36,49,64,81,100]
```

```
it :: (Num a, Enum a) => [a]
```

```
ghci> [x^2 | x <- [1..10] , even x]  
[4,16,36,64,100]
```

```
it :: Integral a => [a]
```

```
ghci> [x^2 | x <- [1..10] , even (x^2)]  
[4,16,36,64,100]
```

```
it :: Integral a => [a]
```

```
ghci> [(x,y) | x<-[1..5], y<-[6..10]]
```

```
[(1,6),(1,7),(1,8),(1,9),(1,10),(2,6),(2,7),(2,8),(2,9),(2,10),(3,6),(3,7),  
(3,8),(3,9),(3,10),(4,6),(4,7),(4,8),(4,9),(4,10),(5,6),(5,7),(5,8),(5,9),  
(5,10)]
```

```
it :: (Num a, Num b, Enum a, Enum b) => [(a, b)]
```

```
ghci> [x*y | x<-[1..10], y<-[1..10]]
```

```
[1,2,3,4,5,6,7,8,9,10,2,4,6,8,10,12,14,16,18,20,3,6,9,12,15,18,21,24,27,30,4,8,  
12,16,20,24,28,32,36,40,5,10,15,20,25,30,35,40,45,50,6,12,18,24,30,36,42,48,54,  
60,7,14,21,28,35,42,49,56,63,70,8,16,24,32,40,48,56,64,72,80,9,18,27,36,45,54,6  
3,72,81,90,10,20,30,40,50,60,70,80,90,100]
```

```
it :: (Enum a, Num a) => [a]
```

```
ghci> [(x,y,x*y) | x<-[1..5], y<-[1..5]]
```

```
[(1,1,1),(1,2,2),(1,3,3),(1,4,4),(1,5,5),(2,1,2),(2,2,4),(2,3,6),(2,4,8),  
(2,5,10),(3,1,3),(3,2,6),(3,3,9),(3,4,12),(3,5,15),(4,1,4),(4,2,8),(4,3,12),  
(4,4,16),(4,5,20),(5,1,5),(5,2,10),(5,3,15),(5,4,20),(5,5,25)]
```

```
it :: (Enum c, Num c) => [(c, c, c)]
```

GHCI more with lists

```
ghci> [s | s<-"goodbye", not (elem s "aeiou")]  
"gdb"  
it :: [Char]
```

```
ghci> :t zip  
zip :: [a] -> [b] -> [(a, b)]  
ghci> zip [1,2,3] "abcd"  
[(1,'a'),(2,'b'),(3,'c')]  
it :: Num a => [(a, Char)]  
ghci> zip [1,2,3,4] "abc"  
[(1,'a'),(2,'b'),(3,'c')]  
it :: Num a => [(a, Char)]
```

```
ghci> let predecessors xs = zip xs (tail xs)  
predecessors :: [b] -> [(b, b)]  
ghci> predecessors [1,2,3,4]  
[(1,2),(2,3),(3,4)]  
it :: Num b => [(b, b)]  
ghci> let predecessors = \xs -> zip xs (tail xs)  
ghci> predecessors [1,2,3,4]  
[(1,2),(2,3),(3,4)]  
it :: Num b => [(b, b)]
```

GHCi lambda

```
ghci> pi
```

```
3.141592653589793
```

```
it :: Floating a => a
```

```
ghci> :t \ r -> 2 * pi * r
```

```
\r -> 2 * pi * r :: Floating a => a -> a
```

```
ghci> \ r -> 2 * pi * r
```

```
<interactive>:20:1: error:
```

- No instance for (Show (Double -> Double))
arising from a use of 'print'
(maybe you haven't applied a function to enough arguments?)
- In a stmt of an interactive GHCi command: print it

```
ghci> (\ r -> 2 * pi * r) 4
```

```
25.132741228718345
```

```
it :: Floating a => a
```

```
ghci> :set +m
```

```
ghci> let circ :: Double -> Double
```

```
ghci|     circ = \ r -> 2 * pi * r
```

```
ghci|
```

```
circ :: Double -> Double
```

```
ghci> circ 3
```

```
18.84955592153876
```

```
it :: Double
```

GHCI lambda

```
ghci> let rect :: (Double,Double) -> Double
ghci|      rect = \ (b,h) -> 2 * b + 2 * h
ghci|
rect :: (Double, Double) -> Double
ghci> rect 3 4
```

<interactive>:34:1: error:

- Couldn't match expected type 't0 -> t' with actual type 'Double'
 - The function 'rect' is applied to two value arguments, but its type '(Double, Double) -> Double' has only one
- In the expression: rect 3 4
In an equation for 'it': it = rect 3 4
- Relevant bindings include it :: t (bound at <interactive>:34:1)

```
ghci> rect (3,4)
14.0
it :: Double
```

```
ghci> let rect = \ b h -> 2 * b + 2 * h
ghci|
rect :: Num a => a -> a -> a
ghci> :t rect 3
rect 3 :: Num a => a -> a
```

GHCI functions

```
ghci> let percent = (/ 100)
```

```
ghci|
```

```
percent :: Fractional a => a -> a
```

```
ghci> percent 50
```

```
0.5
```

```
it :: Fractional a => a
```

```
ghci> let isUpper c = c elem ['A'..'Z']
```

```
ghci|
```

```
isUpper ::
```

```
(Foldable t1, Eq a) => ((a -> t1 a -> Bool) -> [Char] -> t2) -> t2
```

```
ghci> let isUpper c = (`elem` ['A'..'Z']) c
```

```
ghci|
```

```
isUpper :: Char -> Bool
```

```
ghci> let isUpper = (`elem` ['A'..'Z'])
```

```
isUpper :: Char -> Bool
```

```
ghci> isUpper 'D'
```

```
True
```

```
it :: Bool
```

GHCI functions

```
ghci> let twice :: (a -> a) -> a -> a
ghci|     twice f x = f(f(x))
ghci|
twice :: (a -> a) -> a -> a
ghci> twice succ 0
2
it :: (Enum a, Num a) => a
ghci> :t twice succ
twice succ :: Enum a => a -> a
ghci> :t (`twice` 2)
(`twice` 2) :: Num a => (a -> a) -> a
ghci> (`twice` 2) succ
4
it :: (Num a, Enum a) => a

ghci> twice tail [1,2,3,4,5]
[3,4,5]
it :: Num a => [a]
ghci> :t twice tail
twice tail :: [a] -> [a]
ghci> :t (`twice` [1,2,3,4])
(`twice` [1,2,3,4]) :: Num a => ([a] -> [a]) -> [a]
ghci> (`twice` [1,2,3,4]) init
[1,2]
it :: Num a => [a]
```

GHCi functions

```
ghci> let flip :: (a -> b -> c) -> b -> a -> c
ghci|     flip f x y = f y x
ghci|
flip :: (a -> b -> c) -> b -> a -> c
```

```
ghci> flip (/) 2 6
```

```
3.0
```

```
it :: Fractional c => c
```

```
ghci> flip elem [1,2,3,4,5] 4
```

```
True
```

```
it :: Bool
```

```
ghci> flip (:) [2,3,4,5] 1
```

```
[1,2,3,4,5]
```

```
it :: Num a => [a]
```

```
ghci> flip take [1 .. 10] 5
```

```
[1,2,3,4,5]
```

```
it :: (Num a, Enum a) => [a]
```

```
ghci> let oneto = flip take [1 ..]
```

```
ghci|
```

```
oneto :: (Num a, Enum a) => Int -> [a]
```

```
ghci> oneto 12
```

```
[1,2,3,4,5,6,7,8,9,10,11,12]
```

```
it :: (Num a, Enum a) => [a]
```

```
ghci> :q
```

```
Leaving GHCi.
```

```
%
```