

# Overview

---

## Programming Paradigms I

# Programming Paradigms

---

- Types, guard equations, pattern matching, etc.
- **Recursion**
- **List comprehensions**
- Data declarations
- High-order functions

# Recursion

---

- Recursive definitions (inductive definitions)
- The basic mechanism for looping in Haskell
- Base case and recursive case

# List Comprehensions

---

- Sets and lists
- Specifying sets
- Generators and guards
- Can be combined with recursion (e.g., quicksort example)

Overview

---

# The End

# Types and Programming I

---

Mechanism for Defining Functions

# Defining Functions

---

## Basic mechanisms

- Conditional expressions
- Guard equations
- Pattern matching
- List patterns

# Conditional Expressions

---

As in most programming languages, functions can be defined using **conditional expressions**.

```
abs :: Int → Int  
abs n = if n ≥ 0 then n else -n
```

abs takes an integer  $n$  and returns  $n$  if it is non-negative and  $-n$  otherwise.



- Conditional expressions can be nested:

```
signum :: Int → Int  
signum n = if n < 0 then -1 else  
           if n == 0 then 0 else 1
```

- Note:
  - In Haskell, conditional expressions must **always** have an else branch, which avoids any possible ambiguity problems with nested conditionals.

# Guarded Equations

---

As an alternative to conditionals, functions can also be defined using **guarded equations**.

```
abs n | n ≥ 0    = n  
      | otherwise = -n
```



As previously but using guarded equations

- Guarded equations can be used to make definitions involving multiple conditions easier to read:

```
signum n | n < 0    = -1  
         | n == 0   = 0  
         | otherwise = 1
```

- Note:
  - The catch-all condition **otherwise** is defined in the prelude by `otherwise = True`.

# Pattern Matching

---

Many functions have a particularly clear definition using **pattern matching** on their arguments.

```
not :: Bool → Bool  
not False = True  
not True  = False
```



not maps False to True and True to False.

- Functions can often be defined in many different ways using pattern matching; for example:

```
(&&) :: Bool → Bool → Bool
True && True = True
True && False = False
False && True = False
False && False = False
```

can be defined more compactly by

```
True && True = True
_ && _ = False
```

- However, the following definition is more efficient, because it avoids evaluating the second argument if the first argument is False:

```
True  && b = b  
False && _ = False
```

- Note:
  - The underscore symbol `_` is a **wildcard** pattern that matches any argument value.

- Patterns are matched **in order**; for example, the following definition always returns False:

```
_ && _ = False  
True && True = True
```

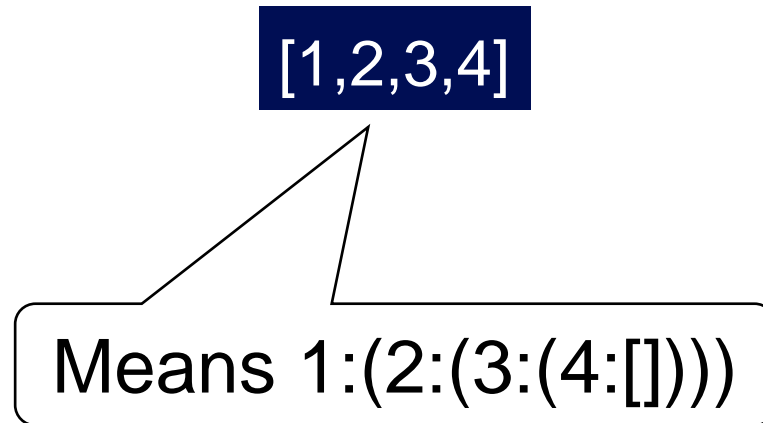
- Patterns may not **repeat** variables; for example, the following definition gives an error:

```
b && b = b  
_ && _ = False
```

# List Patterns

---

Internally, every nonempty list is constructed by repeated use of an operator ( $:$ ) called “**cons**” that adds an element to the start of a list.



$$[] = []$$

$$4:[] = [4]$$

$$3:(4:[]) = [3,4]$$

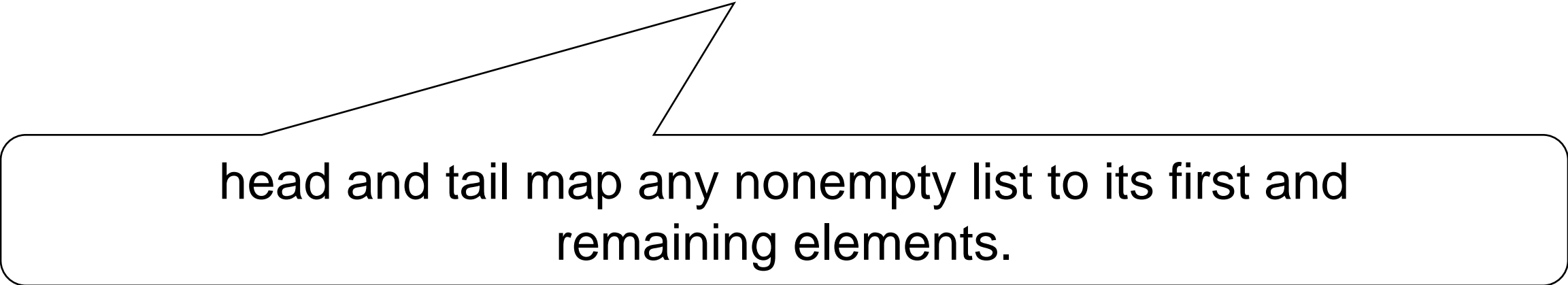
:



Functions on lists can be defined using  **$x:xs$**  patterns.

```
head :: [a] → a  
head (x:_) = x
```

```
tail :: [a] → [a]  
tail (_:xs) = xs
```



head and tail map any nonempty list to its first and remaining elements.

- Note:
  - `x:xs` patterns only match **nonempty** lists:

```
> head []  
*** Exception: empty list
```

- `x:xs` patterns must be **parenthesized** because application has priority over `(:)`; for example, the following definition gives an error:

(x:xs)  
↑  
head x:\_ = x

Types and Programming I

---

# The End

# Types and Programming II

---

Currying, Polymorphic Functions, and Lambda Functions

# Curried Functions

Functions with multiple arguments are also possible by returning **functions as results**:

$\text{add}' :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$   
 $\text{add}' x y = x + y$

$\rightarrow \text{add}' :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$   
*Same meaning*

$\text{add}'$  takes an integer  $x$  and returns a function  **$\text{add}' x$** . In turn, this function takes an integer  $y$  and returns the result  $x+y$ .

Note:

- `add` and `add'` produce the same final result, but `add` takes its two arguments at the same time, whereas `add'` takes them one at a time:

```
add :: (Int,Int) → Int
```

```
add' :: Int → (Int → Int)
```

- Functions that take their arguments one at a time are called **curried** functions, celebrating the work of Haskell Curry on such functions

$add' :: Int \rightarrow Int \rightarrow Int$

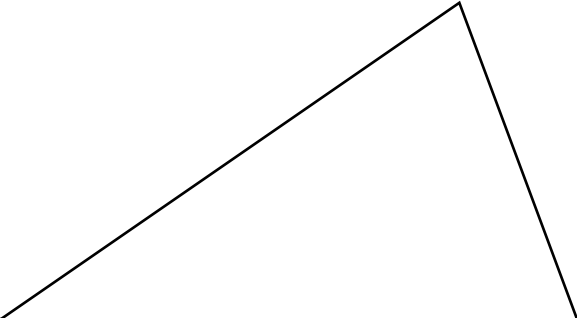
$add\ x\ y = x + y$

$add\ x$

$add\ x\ y = x + y$

Functions with more than two arguments can be curried by returning nested functions:

```
mult :: Int → (Int → (Int → Int))  
mult x y z = x*y*z
```



mult takes an integer  $x$  and returns a function **mult x**, which in turn takes an integer  $y$  and returns a function **mult x y**, which finally takes an integer  $z$  and returns the result  $x*y*z$ .

# Why Is Currying Useful?

---

- Curried functions are more flexible than functions on tuples because useful functions can often be made by **partially applying** a curried function
- For example:

```
add' 1 :: Int → Int
```

```
take 5 :: [Int] → [Int]
```

```
drop 5 :: [Int] → [Int]
```



# Currying Conventions

---

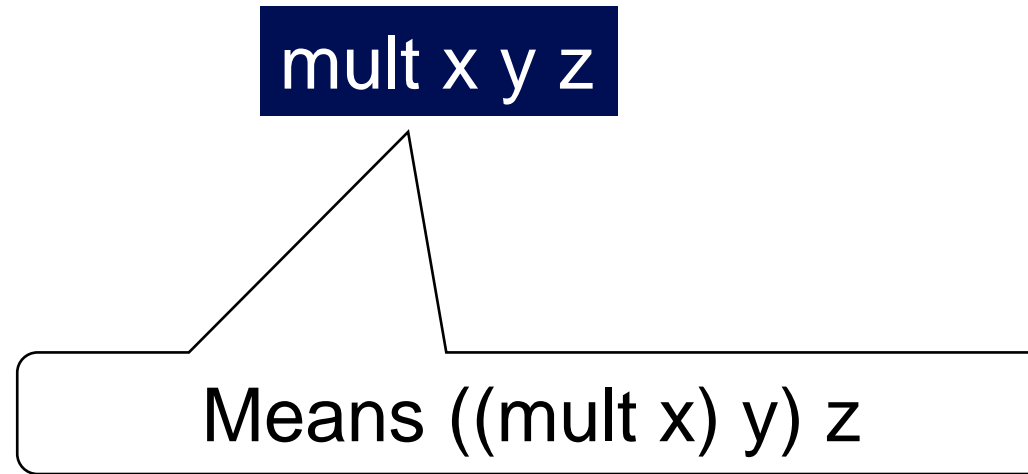
To avoid excess parentheses when using curried functions, two simple conventions are adopted.

- The arrow  $\rightarrow$  associates to the **right**.

$\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

Means  $\text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}))$

- As a consequence, it is then natural for the function application to associate to the **left**.



- Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form.

# Polymorphic Functions

---

A function is called **polymorphic** (“of many forms”) if its type contains one or more type variables.

`length :: [a] → Int`



For any type `a`, `length` takes a list of values of type `a` and returns an integer.

Note:

- Type variables can be instantiated to different types in different circumstances:

```
> length [False,True]  
2
```

a = Bool

```
> length [1,2,3,4]  
4
```

a = Int

- Type variables must begin with a lowercase letter and are usually named a, b, c, etc.

Many of the functions defined in the standard prelude are polymorphic; for example:

```
fst :: (a,b) → a
```

```
head :: [a] → a
```

```
take :: Int → [a] → [a]
```

```
zip :: [a] → [b] → [(a,b)]
```

```
id :: a → a
```

# Overloaded Functions

---

A polymorphic function is called **overloaded** if its type contains one or more class constraints.

$(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$

For any numeric type  $a$ ,  $(+)$  takes two values of type  $a$  and returns a value of type  $a$ .

Note:

- Constrained type variables can be instantiated to any types that satisfy the constraints:

```
> 1 + 2  
3
```

```
> 1.0 + 2.0  
3.0
```

```
> 'a' + 'b'  
ERROR
```

a = Int

a = Float

Char is not a  
numeric type

$(+) :: (\text{Num } a) \Rightarrow a \rightarrow a \rightarrow a$

$\Rightarrow a \rightarrow a \rightarrow a$

Haskell has a number of type classes, including:

**Num** - Numeric types

**Eq** - Equality types

**Ord** - Ordered types

For example:

```
(+) :: Num a => a -> a -> a
```

```
(==) :: Eq a => a -> a -> Bool
```

```
(<) :: Ord a => a -> a -> Bool
```



# Lambda Expressions

---

Functions can be constructed without naming the functions by using **lambda expressions**.


$$\lambda x \rightarrow x + x$$

the nameless function that takes a number  $x$  and returns the result  $x + x$ .

Note:

- The symbol  $\lambda$  is the Greek letter **lambda** and is typed at the keyboard as a backslash \.
- In mathematics, nameless functions are usually denoted using the  $\alpha$  symbol, as in  $x \alpha x + x$ .
- In Haskell, the use of the  $\lambda$  symbol for nameless functions comes from the **lambda calculus**, the theory of functions on which Haskell is based.

# Why Are Lambdas Useful?

---

- Lambda expressions can be used to give a formal meaning to functions defined using **currying**
- For example:

`add x y = x + y`

means

$(\lambda x \rightarrow (\lambda y \rightarrow x + y))$

`add =  $\lambda x \rightarrow (\lambda y \rightarrow x + y)$`

- Lambda expressions are also useful when defining functions that return **functions as results**
- For example:

```
const :: a → b → a  
const x _ = x
```

is more naturally defined by

```
const :: a → (b → a)  
const x = λ_ → x
```

- Lambda expressions can be used to avoid naming functions that are only **referenced once**
- For example:

```
odds n = map f [0..n-1]
      where
        f x = x*2 + 1
```

can be simplified to

```
odds n = map ( $\lambda x \rightarrow x*2 + 1$ ) [0..n-1]
```

$\hookrightarrow f \rightarrow$

if  $n = 5$

$[0, 1, 2, 3, 4]$

$\downarrow$  returns

$[f_0, f_1, \dots, f_5]$

# Operator Sections

---

- An operator written **between** its two arguments can be converted into a curried function written **before** its two arguments by using parentheses
- For example:

```
> 1+2
```

```
3
```

```
> (+) 1 2
```

```
3
```

- This convention also allows one of the arguments of the operator to be included in the parentheses
- For example:

$$\begin{array}{l} > (1+) 2 \\ 3 \end{array}$$
$$\begin{array}{l} > (+2) 1 \\ 3 \end{array}$$

- In general, if  $\oplus$  is an operator, then functions of the form  $(\oplus)$ ,  $(x\oplus)$  and  $(\oplus y)$  are called **sections**

# Why Are Sections Useful?

---

Useful functions can sometimes be constructed in a simple way using sections; for example:

$(1+)$  - Successor function

$(1/)$  - Reciprocation function

$(*2)$  - Doubling function

$(/2)$  - Halving function



# Hints and Tips

---

- When defining a new function in Haskell, it is useful to begin by writing down its type.
- Within a script, it is good practice to state the type of every new function defined.
- When stating the types of polymorphic functions that use numbers, equality, or orderings, take care to include the necessary class constraints.

Types and Programming II

---

# The End

# Recursion I

---

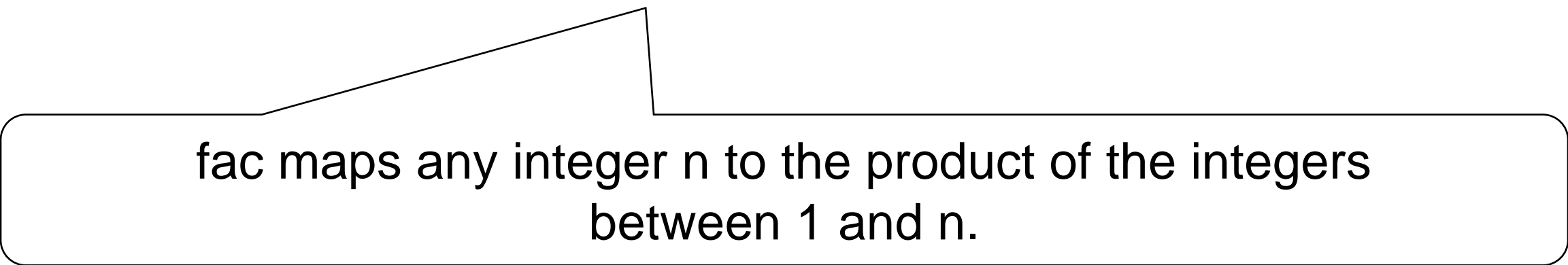
## Recursive Functions

# Introduction

---

As we have seen, many functions can naturally be defined in terms of other functions.

```
fac :: Int → Int  
fac n = product [1..n]
```



fac maps any integer  $n$  to the product of the integers between 1 and  $n$ .

- Expressions are **evaluated** by a stepwise process of applying functions to their arguments
- For example:

fac 4  
=  
product [1..4]  
=  
product [1,2,3,4]  
=  
1\*2\*3\*4  
=  
24

# Recursive Functions

---

In Haskell, functions can also be defined in terms of themselves. Such functions are called **recursive**.

```
fac 0 = 1  
fac n = n * fac (n-1)
```

fac maps 0 to 1 and any other integer to the product of itself and the factorial of its predecessor.

For example:

$$\begin{aligned} & \text{fac } 3 \\ = & 3 * \text{fac } 2 & \text{--- (fac. 2)} \\ = & 3 * (2 * \text{fac } 1) & \text{--- (fac. 2)} \\ = & 3 * (2 * (1 * \text{fac } 0)) & \text{--- (fac 2)} \\ = & 3 * (2 * (1 * 1)) & \text{--- (fac. 1)} \\ = & 3 * (2 * 1) \\ = & 3 * 2 \\ = & 6 \end{aligned}$$

Note:

- $\text{fac } 0 = 1$  is appropriate because 1 is the identity for multiplication:  $1 * x = x = x * 1$
- The recursive definition **diverges** on integers less than 0 because the base case is never reached:

```
> fac (-1)
```

```
*** Exception: stack overflow
```



# Why Is Recursion Useful?

---

- Some functions, such as factorial, are **simpler** to define in terms of other functions.
- As we shall see, however, many functions can **naturally** be defined in terms of themselves.
- Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of **induction**.

# Recursion on Lists

---

Recursion is not restricted to numbers but can also be used to define functions on **lists**.

```
product :: Num a => [a] → a  
product []    = 1  
product (n:ns) = n * product ns
```

product maps the empty list to 1 and any nonempty list to its head multiplied by the product of its tail.

For example:

$$\begin{aligned} & \text{product } [2,3,4] \\ = & 2 * \text{product } [3,4] \\ = & 2 * (3 * \text{product } [4]) \\ = & 2 * (3 * (4 * \text{product } [])) \\ = & 2 * (3 * (4 * 1)) \\ = & 24 \end{aligned}$$

Using the same pattern of recursion as in `product`, we can define the **length** function on lists.

```
length :: [a] → Int  
length []    = 0  
length (_:xs) = 1 + length xs
```

`length` maps the empty list to 0, and any nonempty list to the successor of the length of its tail.

For example:

$$\begin{aligned} & \text{length } [1,2,3] \\ = & 1 + \text{length } [2,3] \\ = & 1 + (1 + \text{length } [3]) \\ = & 1 + (1 + (1 + \text{length } [])) \\ = & 1 + (1 + (1 + 0)) \\ = & 3 \end{aligned}$$

Using a similar pattern of recursion, we can define the **reverse** function on lists.

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

reverse maps the empty list to the empty list and any nonempty list to the reverse of its tail appended to its head.

For example:

`reverse [1,2,3]`  
=  
`reverse [2,3] ++ [1]`  
=  
`(reverse [3] ++ [2]) ++ [1]`  
=  
`((reverse [] ++ [3]) ++ [2]) ++ [1]`  
=  
`(([] ++ [3]) ++ [2]) ++ [1]`  
=  
`[3,2,1]`

# Multiple Arguments

---

Functions with more than one argument can also be defined using recursion; for example:

- Zipping the elements of two lists:

```
zip :: [a] → [b] → [(a,b)]  
zip [] _      = []  
zip _ []      = []  
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```



- Remove the first n elements from a list:

```
drop :: Int → [a] → [a]
drop 0 xs    = xs
drop _ []    = []
drop n (_:xs) = drop (n-1) xs
```

- Appending two lists:

```
(++) :: [a] → [a] → [a]
[]    ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Recursion I

---

# The End

# Recursion II

---

Defining/Specifying a Function by Recursion

# Introduction

---

- The design and specification of quicksort
- General principles
- Advise

# Quicksort

---

The **quicksort** algorithm for sorting a list of values can be specified by the following two rules:

1. The empty list is already sorted.
2. Nonempty lists can be sorted by sorting the tail values less than or equal to the head, sorting the tail values greater than the head, and then appending the resulting lists on either side of the head value.



$q(\text{quicksort})$

$q [] = []$

# Quicksort

The **quicksort** algorithm for sorting a list of values can be specified by the following two rules:

1. The empty list is already sorted.
2. Nonempty lists can be sorted by sorting the tail values less than or equal to the head, sorting the tail values greater than the head, and then appending the resulting lists on either side of the head value.

$$q(x:xs) = [ \overset{q}{\uparrow} ] ++ [x] ++ [ \overset{q}{\dots} ]$$

$y \text{ from } xs \quad y \leq x \qquad z \text{ from } xs, \quad z > x$

Using recursion, this specification can be translated directly into an implementation:

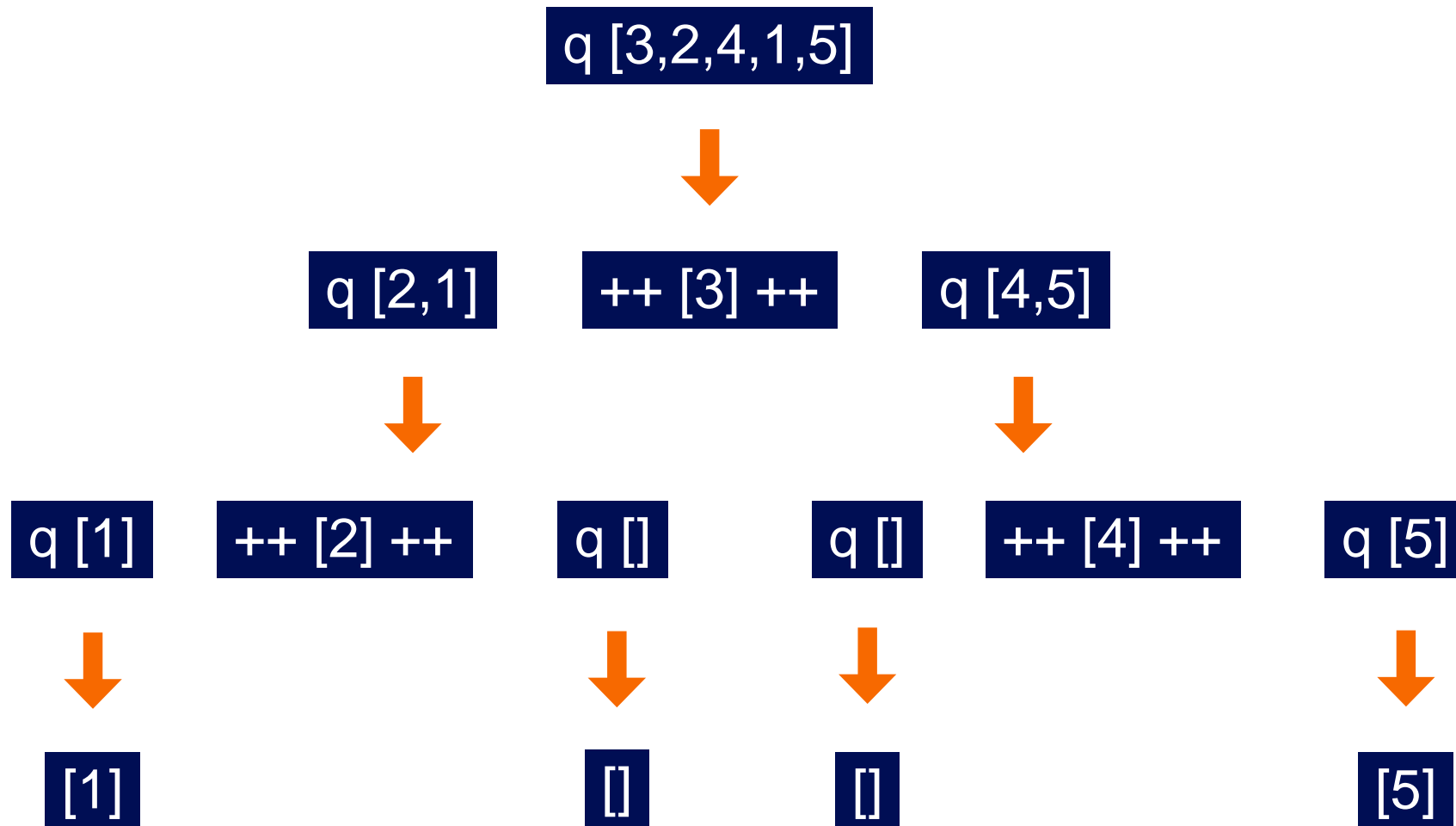
① →  
② →

```
qsort :: Ord a => [a] → [a]
qsort []    = []
qsort (x:xs) =
  qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [a | a ← xs, a ≤ x]
    larger  = [b | b ← xs, b > x]
```

Note:

- This is probably the **simplest** implementation of quicksort in any programming language!

For example (abbreviating qsort as q):





# Recall: Hints and Tips

---

- When defining a new function in Haskell, it is useful to begin by writing down its type.
- Within a script, it is good practice to state the type of every new function defined.
- When stating the types of polymorphic functions that use numbers, equality, or orderings, take care to include the necessary class constraints.

In quicksort, we can use it for  $[a]$   
in Ord

# Advice

---

- Define the type.
- Enumerate the cases.
- Define the simple cases.
- Define the other cases.
- Generalize and simplify.

# Practice

Steps	Quicksort <i>q</i>
Define the type.	✓
Enumerate the cases.	<i>[ ]</i> <i>(x:xs)</i>
Define the simple cases.	<i>q [ ]</i>
Define the other cases.	<i>q (x:xs)</i>
Generalize and simplify.	<i>...</i>

---

$\text{drop} :: \text{Int} \rightarrow [a] \rightarrow [a]$

$k \quad xs$

$xs$

$\dots$

$\vdash \downarrow -1$

drop  
the  
first  $k$   $elts$

---

$\swarrow$   
 $\text{drop} :: \text{Int} \rightarrow [a] \rightarrow [a]$

$\text{drop } 0 \ [ ] = []$

$\text{drop } 0 \ (x:xs) = x:xs$

$\text{drop } n \ [ ] = []$

$\text{drop } n \ (x:xs) = \text{drop } (n-1) \ xs$

Work for Int or Integer  $\rightarrow$  Integer.

Recursion II

---

# The End

# List Comprehensions I

---

Sets, Set Builder Notation, and Lists

# Set Comprehensions

In mathematics, the **comprehension** notation can be used to construct new sets from old sets.

transform each  
elt to  
the new  
form

$$\{x^2 \mid x \in \{1 \dots 5\}\}$$

generate elements  
from a set

The set  $\{1, 4, 9, 16, 25\}$  of all numbers  $x^2$  such that  $x$  is an element of the set  $\{1 \dots 5\}$ .



# Lists Comprehensions

---

In Haskell, a similar comprehension notation can be used to construct new **lists** from old lists.

`[x^2 | x ← [1..5]]`

The list [1,4,9,16,25] of all numbers  $x^2$  such that  $x$  is an element of the list [1..5].

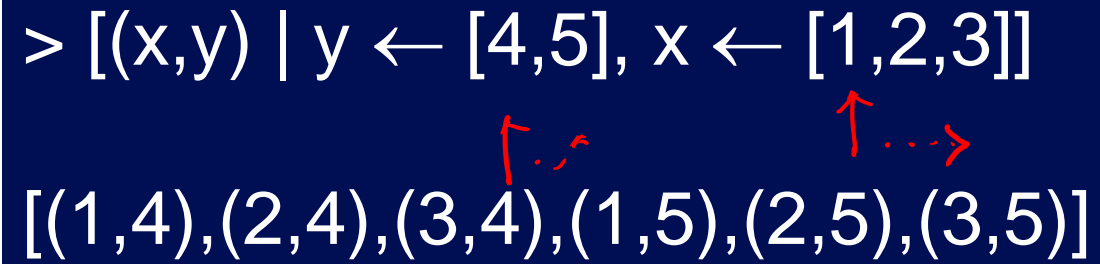
Note:

- The expression  $x \leftarrow [1..5]$  is called a **generator**, as it states how to generate values for  $x$
- Comprehensions can have **multiple** generators, separated by commas; for example:

```
> [(x,y) | x ← [1,2,3], y ← [4,5]]  
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

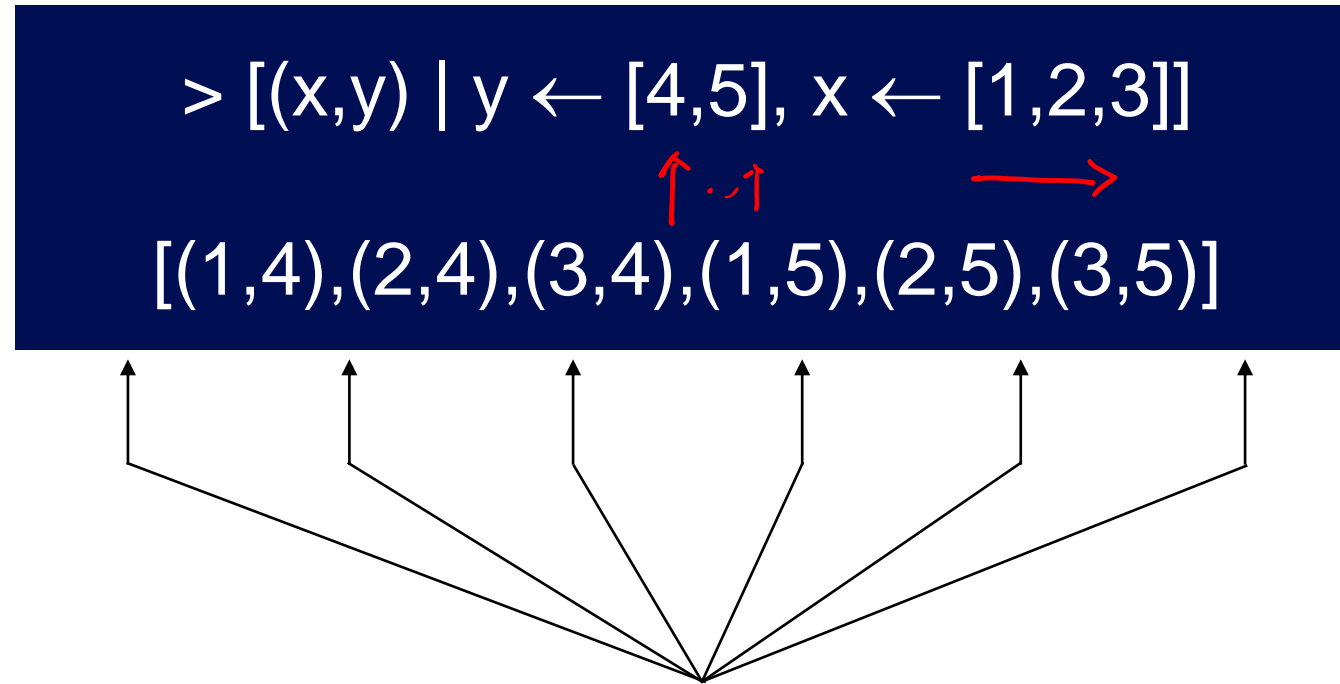
- Changing the **order** of the generators changes the order of the elements in the final list:

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]  
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```



- Multiple generators are like **nested loops**, with later generators as more deeply nested loops whose variables change value more frequently

For example:



$x \leftarrow [1, 2, 3]$  is the last generator, so the value of the  $x$  component of each pair changes most frequently.

# Dependent Generators

---

Later generators can **depend** on the variables that are introduced by earlier generators.

$[(x,y) \mid x \leftarrow [1..3], y \leftarrow [x..3]]$

The list  $[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]$   
of all pairs of numbers  $(x,y)$  such that  $x,y$  are elements of the list  
 $[1..3]$  and  $y \geq x$ .

- Using a dependent generator, we can define the library function that **concatenates** a list of lists:

```
concat :: [[a]] → [a]
concat xss = [x | xs ← xss, x ← xs]
```

- For example:

*xs*    *[1, 2, 3]*  
/    ↑  
     x

```
> concat [[1,2,3],[4,5],[6]]
[1,2,3,4,5,6]
```

List Comprehensions I

---

# The End

# List Comprehensions II

---

List and String Comprehensions

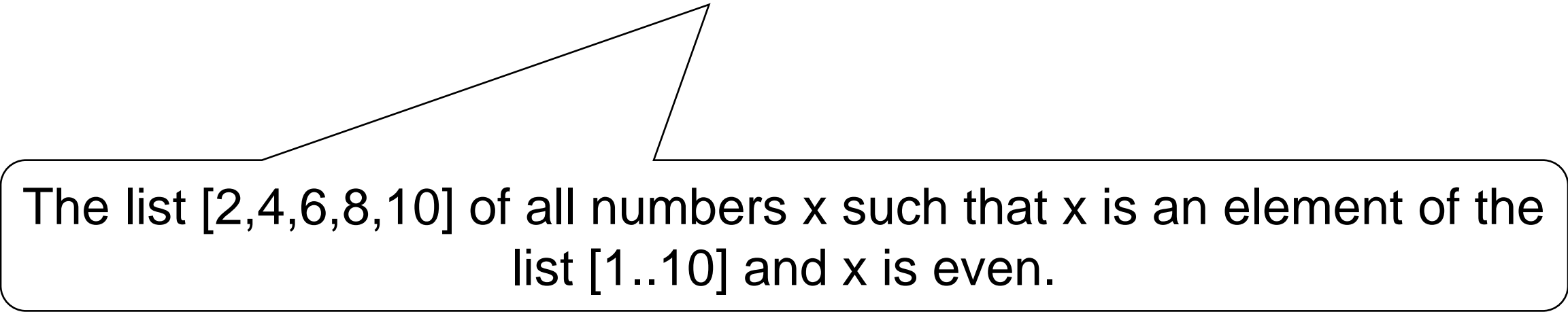


# Guards

---

List comprehensions can use **guards** to restrict the values produced by earlier generators.

```
[x | x ← [1..10], even x]
```



The list [2,4,6,8,10] of all numbers x such that x is an element of the list [1..10] and x is even.

- Using a guard, we can define a function that maps a positive integer to its list of **factors**:

```
factors :: Int → [Int]
factors n =
  [x | x ← [1..n], n `mod` x == 0]
```

- For example:

```
> factors 15
[1,3,5,15]
```

- A positive integer is **prime** if its only factors are 1 and itself; hence, using factors, we can define a function that decides if a number is prime:

```
prime :: Int → Bool  
prime n = factors n == [1,n]
```

- For example:

```
> prime 15  
False  
  
> prime 7  
True
```

- Using a guard, we can now define a function that returns the list of all **primes** up to a given limit:

```
primes :: Int → [Int]
primes n = [x | x ← [2..n], prime x]
```

- For example:

```
> primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
```

# The Zip Function

---

A useful library function is **zip**, which maps two lists to a list of pairs of their corresponding elements.

```
zip :: [a] → [b] → [(a,b)]
```

- For example:

```
> zip ['a','b','c'] [1,2,3,4]  
[('a',1),('b',2),('c',3)]
```

- Using zip, we can define a function returns the list of all **pairs** of adjacent elements from a list:

```

pairs :: [a] → [(a,a)]
pairs xs = zip xs (tail xs)

```

- For example:

```

> pairs [1,2,3,4]
[(1,2),(2,3),(3,4)]

```

Zip [a,b,c] [1..]



[(a,1), (b,2), (c,3)]

Zip [a,b,c] [b,c]

"

[(a,b), (b,c)]

[a,b,c]

↘ [(a,b), (b,c)]

- Using pairs, we can define a function that decides if the elements in a list are **sorted**:

```
sorted :: Ord a => [a] → Bool  
sorted xs = and [x ≤ y | (x,y) ← pairs xs]
```

*✓ combine all cases*


- For example:

```
> sorted [1,2,3,4]  
True
```

```
> sorted [1,3,2,4]  
False
```

- Using zip, we can define a function that returns the list of all **positions** of a value in a list:

```
positions :: Eq a => a -> [a] -> [Int]
positions x xs =
  [i | (x',i) <- zip xs [0..], x == x']
```

- For example: 

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```

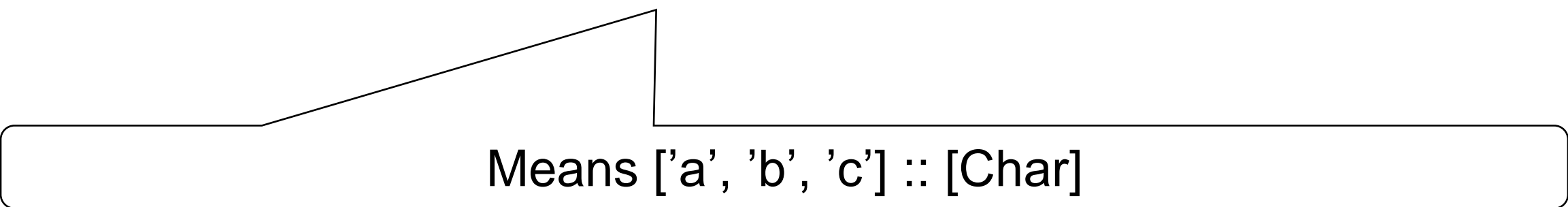


# String Comprehensions

---

A **string** is a sequence of characters enclosed in double quotes. Internally, however, strings are represented as lists of characters.

`"abc" :: String`



Means `['a', 'b', 'c'] :: [Char]`

The diagram consists of a dark blue box containing the text `"abc" :: String`. A line extends from the bottom-left corner of this box, goes down vertically, then left horizontally, and finally up diagonally to point at a rounded rectangular box below. This lower box contains the text `Means ['a', 'b', 'c'] :: [Char]`.

Because strings are just special kinds of lists, any **polymorphic** function that operates on lists can also be applied to strings; for example:

```
> length "abcde"
```

```
5
```

```
> take 3 "abcde"
```

```
"abc"
```

```
> zip "abc" [1,2,3,4]
```

```
[('a',1),('b',2),('c',3)]
```

- Similarly, list comprehensions can also be used to define functions on strings, such as counting how many times a character occurs in a string:

```
count :: Char → String → Int  
count x xs = length [x' | x' ← xs, x == x']
```

- For example:

```
> count 's' "Mississippi"  
4
```

*count*      *type string*

*'s'*

List Comprehensions II

---

# The End

# Weekly Summary

---

## Programming Paradigm I

# Summary, Part I

---

## Specifying/defining functions in Haskell

- Basic mechanisms
- Recursion
- List comprehensions

# Summary, Part II

---

## Basic mechanisms

- Types: include the types for the functions to be defined (reduce type errors)
- Conditional and guard equations (note the ordering of checking conditions and the keyword “otherwise”)
- (Important) Pattern matching for various types

# Summary, Part III

---

Define/specify functions by recursion

- A recursive definition includes both base case(s) and the recursion step(s)
- Verify both sides to void unintended infinite computations
- Pay attention to the pattern matching used in the function definition(s)
- Follow the general advice given by the text



# Summary, Part IV

---

Define/specify functions by list comprehensions

- Specify relevant conditions using set-builder notations and try to translate them into code
- It is often helpful to index a list and use zip function in those cases
- They can be used together with recursion (e.g., quick sort)

Weekly Summary

---

# The End