

# Lists and Abstraction

**David Basin**

Department of Computer Science  
ETH Zurich

# List type

- Types we have seen
  - ▶ Basis types: `Int`, `Char`, `Bool`, . . .
  - ▶ Type constructors:  $(T_1, \dots, T_k)$
- List types, a new type constructor

If  $T$  is a type, then  $[T]$  is a type
- Elements of  $[T]$ 
  - ▶ Empty list  $[] :: [T]$
  - ▶ If  $x :: T$  and  $xs :: [T]$ , then  $(x : xs) :: [T]$
- Short hand:  $1 : (2 : (3 : []))$  written as  $[1, 2, 3]$

## Examples and notation

```
? ['a','a','b'] :: [Char]
"aab" :: [Char]
```

```
? ['a','a','b'] == "aab"
True
```

Haskell supports various abbreviations

```
? [3..6]
[3, 4, 5, 6] :: [Int]
```

```
? [6..3]
[] :: [Int]
```

$[n, p..m]$  means count from  $n$  to  $m$  in steps of  $p - n$

```
? [7,6..3]
[7, 6, 5, 4, 3] :: [Int]
```

```
? [0.0, 0.3 .. 1.0]
[0.0, 0.3, 0.6, 0.9] :: [Double]
```

## Functions on lists — sumList

- Function `sumList :: [Int] -> Int` must specify:
  - ▶ how to compute with the **empty list** `[]`
  - ▶ how to compute with the **non-empty list** `(x : xs)`
- Computation

**Empty list** `[]` : 0

**Non-empty list** `(x : xs)` :  $x + \text{sum of list } xs$

```
sumList [] = 0
```

```
sumList (x:xs) = x + sumList xs
```

```
? sumList [1..100]           -- 100 * 101 / 2  
5050 :: Int
```

# Standard functions on lists

- length

```
length [] = 0
length (x:xs) = 1 + length xs
```

- append (not only for strings!)

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

N.B.:  $(++)$  and  $(:)$  have different types!

```
? [2] ++ [3,4,5] == [2,3,4,5]
True
```

```
? 2 : [3,4,5] == [2,3,4,5]
True
```

```
? [2] : [3,4,5]
... Error ...
```

## Patterns (lists and in general)

- Pattern matching has two purposes
  - ▶ checks if an argument has the proper form
  - ▶ binds values to variables
- Example:  $(x : xs)$  matches with  $[2, 3, 4]$  ( $= 2 : 3 : 4 : [] = 2 : [3, 4]$ )

$$\begin{aligned} x &= 2 \\ xs &= [3, 4] \end{aligned}$$

- Another example (pattern matching during let-binding)

```
? let ([x,y,z],t) = ([1,2,3],(20,30)) in x + y
3 :: Int
```

```
? let ([x,y,z],t) = ([1,2,3,4],(20,30)) in x + y
Pattern match fails...
```

## Patterns — details

- Patterns are inductively defined (additional constructors later)

**Constants:**  $-2$ ,  $'1'$ ,  $True$ ,  $[]$

**Variables:**  $x$ ,  $foo$

**Wild card:**  $_$

**Tuples:**  $(p_1, p_2, \dots, p_k)$ , where  $p_i$  are patterns

**Lists:**  $(p_1 : p_2)$ , where  $p_i$  are patterns

- Moreover, patterns required to be **linear**.

This means that each variable can occur at most once

- Examples:  $[(x, foo), -]$ ,  $((x, y), -)$ , and  $1 : (2 : (x, y))$
- Counterexamples:  $(x ++ y, z)$  and  $[x, y, z, x]$

# Pattern matching

- Define pattern  $p$  **matches** term  $a$  by recursion on  $p$ .

**Constant**  $p = c$ : succeeds if  $c = a$

**Variable**  $p = x$ : succeeds and with binding  $x = a$

**Wild card**  $p = \_$ : succeeds but no binding

**Tuple**  $p = (p_1, \dots, p_k)$ : succeeds if  $a = (a_1, \dots, a_k)$  and  $p_i$  **matches**  $a_i$ , for  $i \in \{1, \dots, k\}$

**List**  $p = (p_1 : p_2)$ : succeeds if  $a$  is a nonempty list  $a_1 : a_2$  and  $p_1$  **matches**  $a_1$  and  $p_2$  **matches**  $a_2$

- Successful or not?

►  $([x], y)$  matches  $([1], 2 + 3)$

►  $[x]$  matches  $[1, 2]$ ?       $[x, y]$  matches  $[1]$ ?

►  $x : y$  matches  $[1, 2]$ ?       $x : (y : z)$  matches  $[1, 2]$ ?

►  $[x, x]$  matches  $[1, 1]$



# Examples

- Zipper function

$$\begin{aligned} \text{zip } [2, 3, 4] \ [4, 5, 78] &= [(2, 4), (3, 5), (4, 78)] \\ \text{zip } [2, 3] \ [1, 2, 3] &= [(2, 1), (3, 2)] \end{aligned}$$

N.B.: extra elements in a longer list are discarded.

- Implementing *zip*

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

- Which functions are implemented here?

```
f (False, False) = False
f (_,        _   ) = True
```

```
g (True, _   ) = True
g (_,    True) = True
g (_,    _   ) = False
```

## Intermezzo — advice on recursion

“Defining recursive functions is like riding a bicycle: it looks easy when someone else is doing it; it may seem impossible when you first try to do it yourself, but becomes simple and natural with practice.”  
— G. Hutton, *Programming in Haskell*

Function *drop* removes the first  $n$  elements of a list.

**Step 1:** Define the type:

```
drop :: Int -> [Int] -> [Int]
```

**Step 2:** Enumerate the cases:

```
drop 0 [] = ...
```

```
drop 0 (x:xs) = ...
```

```
drop n [] = ...
```

```
drop n (x:xs) = ...
```

**Step 3:** Define the simple cases:

```
drop 0 [] = []
```

```
drop 0 (x:xs) = x:xs
```

```
drop n [] = []
```

**Step 4:** Define the other cases:

```
drop n (x:xs) = drop (n-1) xs
```

**Step 5:** Generalize and simplify.

**Suggestions?**

## Example: insertion sort

- Insertion sort: [7, 3, 9, 2]
  - ▶ First sort rest: [2, 3, 9]
  - ▶ Insert head: [2, 3, 7, 9]

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

```
ins :: Int -> [Int] -> [Int]
ins a [] = [a]
ins a (x:xs)
  | a <= x    = a : (x : xs)
  | otherwise = x : ins a xs
```

```
isort [3,9,2]
  = ins 3 (isort [9,2])
  = ins 3 (ins 9 (isort [2]))
  = ins 3 (ins 9 (ins 2 (isort [])))
  = ins 3 (ins 9 (ins 2 []))
  = ins 3 (ins 9 ([2]))
  = ins 3 (2: (ins 9 []))
  = ins 3 [2, 9]
  = 2 : (ins 3 [9])
  = 2 : [3,9] = [2,3,9]
```

## Insertion sort — complexity

- Processing list  $[x_1, \dots, x_n]$  of size  $n$  results in

$ins\ x_1\ (ins\ x_2\ (\dots (ins\ x_n\ [])) \dots))$

- Complexity of computing  $ins\ a\ [x_1, x_2, \dots, x_n]$

**Best case:**  $a \leq x_1$ : 1 step

**Worst case:**  $a > x_n$ :  $n$  steps

**Average:**  $n/2$  steps (assuming all input sequences equally likely)

- Complexity of insertion sort:

**Best case:**  $n$  steps, i.e.,  $O(n)$

**Worst case:**  $1 + 2 + \dots + n = n(n + 1)/2$ , i.e.,  $O(n^2)$

**Average:**  $1/2 + 2/2 + \dots + (n - 1)/2 + n/2$ , also  $O(n^2)$

# Quick sort

- Quick sort:  $[7, 3, 8, 2, 9]$ 
  - ▶ Split into head and tail: 7 and  $[3, 8, 2, 9]$
  - ▶ Partition tail into parts  $\leq 7$  and  $> 7$ :  $[3, 2]$  and  $[8, 9]$
  - ▶ Recursively sort:  $[2, 3]$  and  $[8, 9]$
  - ▶ Concatenate with head in the middle:  $[2, 3] ++ [7] ++ [8, 9] = [2, 3, 7, 8, 9]$

```

qsort [] = []
qsort (x:xs) =
  qsort (lesseq x xs) ++ [x] ++ qsort (greater x xs)
  where
    lesseq _ [] = []
    lesseq x (y:ys)
      | (y <= x) = y : lesseq x ys
      | otherwise = lesseq x ys
    greater _ [] = []
    greater x (y:ys)
      | (y > x) = y : greater x ys
      | otherwise = greater x ys

```

- Complexity:  $O(n \log n)$  on average. Worst case?

## List comprehension

- Notation for sequential processing of list elements

► Analogous to set comprehension in set theory  $\{2 \cdot x \mid x \in X\}$

► Haskell notation: `[2*x | x <- xs]`

```
? [2*x | x <- [1,2,3,4,5]]
[2, 4, 6, 8, 10]
```

```
? [n 'mod' 2 == 0 | n <- [2,4,7]]
[True,True,False]
```

- Can be augmented with guards: `[2*x | x <- xs, pred1(x), ...]`

```
? [2*x | x <- [0,1,2,3,4,5,6], x 'mod' 2 == 0, x > 3]
[8, 12]
```

- What is computed here?

```
q [] = []
q (p:xs) = q [x | x<-xs, x <= p] ++ [p] ++ q [x | x<-xs, x > p]
```

# Program development with lists

## A larger example

- Objective: a (mini-)library database
  - A design method for programs “in the small”
    1. Specify the requirements
    2. Fix the types (input/output representation)
    3. Implement each function
  - For systems “in the large”, design is substantially more difficult
- Topic for software engineering courses

## Step 1: requirements analysis

Which functionality is required?

1. Given a person  $p$ , which books has  $p$  borrowed?
2. Given a book  $b$ , who has borrowed  $b$ ?

Assumption: many-to-many relation between (copies of) books and persons

3. Is a book lent out?



## Step 2: types

- We define two types to represent books and people

```
type Person = String  -- Note: types always start with  
type Book   = String  --           a capital letter
```

- Database: list of (Person, Book)-pairs

```
type Database = [(Person, Book)]
```

- Example

```
myDB = [("Alice", "Postman Pat"), ("Anna", "All Alone"),  
        ("Alice", "Spot"), ("Rory", "Postman Pat")]
```

## Types (cont.)

- Each kind of functionality implemented by a separate function

1. Which books has person  $p$  borrowed?

`books :: Database -> Person -> [Book]`

2. Who has borrowed book  $b$ ?

`borrowers :: Database -> Book -> [Person]`

3. Is a book  $b$  on loan?

`borrowed :: Database -> Book -> Bool`

- Further functionality could also be specified

E.g., checking out or returning a book

## Step 3: implementation

- `books :: Database -> Person -> [Book]`

```
books db p = [bk | (per,bk) <- db, per == p]
```

- `borrowers :: Database -> Book -> [Person]`

```
borrowers db b = [per | (per,bk) <- db, bk == b]
```

- `borrowed :: Database -> Book -> Bool`

```
borrowed [] _ = False  
borrowed ((_,bk):rest) b = bk == b || borrowed rest b
```

## Complete program and examples

```

type Person    = String
type Book      = String
type Database  = [(Person,Book)]

myDB  = [("Alice", "Postman Pat"), ("Anna", "All Alone"),
        ("Alice", "Spot"), ("Rory", "Postman Pat")]

books      db p = [bk | (per,bk) <- db, per == p]
borrowers db b = [per | (per,bk) <- db, bk == b]

borrowed [] _ = False
borrowed ((_,bk):rest) b = bk == b || borrowed rest b

? books myDB "Alice"
["Postman Pat", "Spot"]
? borrowers myDB "All Alone"
["Anna"]
? borrowed myDB "Postman Pat"
True

```

# Correctness — Induction

## Review: induction over natural numbers

- Rule for induction over  $\mathcal{N}$ .

**Proof by induction:** to prove  $P(n)$  for all natural numbers  $n$ .

**Base case:** prove  $P(0)$

**Step case:** prove  $P(n) \rightarrow P(n + 1)$  for an arbitrary  $n \in \mathcal{N}$ , i.e.,

**Induction hypothesis:**  $P(n)$

**To prove:**  $P(n + 1)$

- Once proven, we know, for example,  $P(17)$  since
  - ▶  $P(0)$  holds, and
  - ▶  $P(0) \rightarrow P(1) \dots$  and  $\dots P(16) \rightarrow P(17)$
  - ▶ since all  $n \in \mathcal{N}$  are reachable in finitely many steps, we know that  $P(n)$ , for every  $n \in \mathcal{N}$ .
- Reflects that  $N$  is the least set such that  $0 \in \mathcal{N}$  and if  $n \in \mathcal{N}$ , then so is  $n + 1 \in \mathcal{N}$ .

## Induction over lists

- How are elements in  $[T]$  constructed?

$[] :: [T]$     and     $(x : xs) :: [T]$ , if  $x :: T$  and  $xs :: [T]$

- Corresponds to following rule

**Proof by induction:** to prove  $P(xs)$  for all  $xs$  in  $[T]$

**Base case:** prove  $P([])$

**Step case:** prove  $P(x : xs)$  under the assumption  $P(xs)$ , for an arbitrary  $xs :: [T]$  and  $x :: T$ , i.e.,

**Induction hypothesis:**  $xs :: [T]$ ,  $x :: T$ , and  $P(xs)$

**To prove:**  $P(x : xs)$

- Like with numbers: induction can be seen as a “machine” that establishes a property for all lists.

E.g.,  $P([3, 20])$  follows since  $P([]) \rightarrow P(20 : []) \rightarrow P(3 : 20 : [])$ .

## Example 1: sum and double

$\text{sumList } [] = 0$                        $\text{double } [] = []$   
 $\text{sumList } (x:xs) = x + \text{sumList } xs$        $\text{double } (x:xs) = (2 \cdot x) : \text{double } xs$

**Proof by induction:**  $P(xs) \equiv \text{sumList } (\text{double } xs) = 2 \cdot \text{sumList } xs$

**Base case:**  $P([])$

$$\begin{aligned}
 \text{sumList } (\text{double } []) &= \text{sumList } [] = 0 \\
 2 \cdot \text{sumList } [] &= 2 \cdot 0 = 0
 \end{aligned}$$

**Step case:**  $P(xs) \rightarrow P(x : xs)$

**Induction hypothesis:**  $\text{sumList } (\text{double } xs) = 2 \cdot \text{sumList } xs$

**To prove:**  $\text{sumList } (\text{double } (x : xs)) = 2 \cdot \text{sumList } (x : xs)$

$$\begin{aligned}
 \text{sumList } (\text{double } (x : xs)) &= \text{sumList } ((2 \cdot x) : \text{double } xs) \\
 &= 2 \cdot x + \text{sumList } (\text{double } xs) \\
 &= 2 \cdot x + 2 \cdot \text{sumList } xs \\
 &= 2 \cdot (x + \text{sumList } xs) \\
 &= 2 \cdot (\text{sumList } (x : xs))
 \end{aligned}$$



## Example 2: associativity of append

$$\begin{aligned} [] & \quad ++ \text{ys} = \text{ys} \\ (x : xs) & \quad ++ \text{ys} = x : (xs ++ \text{ys}) \end{aligned}$$

**Proof by induction:**

$$Q(xs) \equiv (xs ++ ys) ++ zs = xs ++ (ys ++ zs)$$

**Base case:**  $([] ++ ys) ++ zs = ys ++ zs = [] ++ (ys ++ zs)$

**Step case:**

**Induction hypothesis:**  $(xs ++ ys) ++ zs = xs ++ (ys ++ zs)$

**To prove:**  $((x : xs) ++ ys) ++ zs = (x : xs) ++ (ys ++ zs)$

$$\begin{aligned} ((x : xs) ++ ys) ++ zs &= (x : (xs ++ ys)) ++ zs \\ &= x : ((xs ++ ys) ++ zs) \\ &= x : (xs ++ (ys ++ zs)) \\ &= (x : xs) ++ (ys ++ zs) \end{aligned}$$

## Induction: difficulties

- How do we select an induction variable?

$$\begin{aligned} P(xs) &\equiv \text{sumList } (\text{double } xs) = 2 \cdot \text{sumList } xs \\ Q(???) &\equiv (xs ++ ys) ++ zs = xs ++ (ys ++ zs) \end{aligned}$$

- ▶ Induction on  $ys$  or  $zs$  fails, e.g.,  $Q(zs)$
- ▶ In step case we must show  $Q(z : zs)$

$$(xs ++ ys) ++ (z : zs) = xs ++ (ys ++ (z : zs))$$

Not possible either to further simplify or to use  $Q(zs)$ .

- Requires planning in order to use the induction hypothesis
- Auxiliary lemmas are sometimes also needed.

## Example: *rev* and *qrev*

```

rev []      = []
rev (x:xs) = rev xs ++ [x]

qrev xs = aux (xs, [])
aux ([],      ys) = ys
aux ((x:xs),  ys) = aux (xs, x:ys)

```

- Append is  $O(n)$  in first argument

$$\begin{aligned}
 [a_1, a_2, \dots, a_n] ++ x &= a_1 : ([a_2, \dots, a_n] ++ x) \\
 &= \dots \text{ } n - 1 \text{ times } \dots \\
 &= a_1 : (a_2 : \dots : (a_n : x) \dots)
 \end{aligned}$$

- *rev* is  $O(n^2)$  (average append:  $n/2$  steps)

$$\begin{aligned}
 rev [a_1, a_2, \dots, a_n] &= rev [a_2, \dots, a_n] ++ [a_1] \\
 &= \dots \text{ } n - 1 \text{ times } \dots \\
 &= (\dots ([] ++ [a_n]) ++ \dots ++ [a_2]) ++ [a_1]
 \end{aligned}$$

## Example (cont.)

- $qrev$  is  $O(n)$

$$\begin{aligned} qrev [a_1, \dots, a_n] &= aux ([a_1, \dots, a_n], []) \\ &= aux ([a_2, \dots, a_n], [a_1]) \\ &= \dots \text{ } n - 2 \text{ times } \dots \\ &= aux ([], [a_n, \dots, a_2, a_1]) \\ &= [a_n, \dots, a_2, a_1] \end{aligned}$$

## Relationship between *rev* and *qrev*

- Is  $rev\ xs = qrev\ xs$  ?

**Base case:**  $P([])$

$$rev\ [] = [] = aux\ ([], []) = qrev\ []$$

**Step case:**  $P(xs) \rightarrow P(x : xs)$

**Induction hypothesis:**  $rev\ xs = qrev\ xs$

**To prove:**  $rev\ (x : xs) = qrev\ (x : xs)$

$$\begin{aligned} rev\ (x : xs) &= rev\ xs ++ [x] \\ qrev\ (x : xs) &= aux\ (x : xs, []) = aux\ (xs, [x]) \end{aligned}$$

- No proof!

- ▶ We require  $rev\ xs ++ [x] = aux\ (xs, [x])$
- ▶ Induction hypothesis only says  $rev\ xs = qrev\ xs = aux\ (xs, [])$

## Next induction

- To prove:  $rev\ xs ++ [x] = aux\ (xs, [x])$ . Let's try induction again

**Base case:**  $P([])$

$$rev\ [] ++ [x] = [] ++ [x] = [x] = aux\ ([], [x])$$

**Step case:**  $P(xs) \rightarrow P(a : xs)$

**Induction hypothesis:**  $rev\ xs ++ [x] = aux\ (xs, [x])$

**To prove:**  $rev\ (a : xs) ++ [x] = aux\ (a : xs, [x])$

$$\begin{aligned} rev\ (a : xs) ++ [x] &= (rev\ xs ++ [a]) ++ [x] \\ &= rev\ xs ++ ([a] ++ [x]) \\ &= rev\ xs ++ [a, x] \\ aux\ (a : xs, [x]) &= aux\ (xs, [a, x]) \end{aligned}$$

- Problem: Induction hypothesis only proven for accumulator  $[x]$ .  
Rather than another induction, let's try a **generalization**!

# Generalization

- To prove  $rev\ xs \ ++\ [x] = aux\ (xs, [x])$ , we prove

$$rev\ xs \ ++\ ys = aux\ (xs, ys)$$

I.e., prove  $\forall xs. P(xs)$ , where

$$P(xs) \equiv \forall ys. rev\ xs \ ++\ ys = aux\ (xs, ys)$$

- Recall proof rules for  $\forall x.Q(x)$

- ▶  $\forall$ -Introduction

**To prove:**  $\forall x.Q(x)$

**Prove:**  $Q(x)$ , where  $x$  is arbitrary

- ▶  $\forall$ -Elimination

From  $\forall x.Q(x)$  we can conclude  $Q(t)$ , for every  $t$

**Prove:**  $\forall \mathbf{x}s. \forall \mathbf{y}s. \text{rev } \mathbf{x}s ++ \mathbf{y}s = \text{aux } (\mathbf{x}s, \mathbf{y}s)$

**Proof by induction:**  $\forall xs. P(xs)$  by induction

$$P(xs) \equiv \forall ys. \text{rev } xs ++ ys = \text{aux } (xs, ys)$$

**Base case:**  $P([])$

$$\forall ys. \text{rev } [] ++ ys = \text{aux } ([], ys)$$

Using  $\forall$ -Introduction, it is sufficient to prove:

$$\text{rev } [] ++ ys = \text{aux } ([], ys)$$

Holds as  $\text{rev } [] ++ ys = [] ++ ys = ys = \text{aux } ([], ys)$



## Proof (cont.)

**Step case:**  $P(xs) \rightarrow P(x : xs)$

**Induction hypothesis:**  $\forall ys. rev\ xs \ ++\ ys = aux\ (xs, ys)$

**To prove:**  $\forall ys. rev\ (x : xs) \ ++\ ys = aux\ (x : xs, ys)$

Using  $\forall$ -introduction, we reduce the goal (for  $ys$  arbitrary) to

$$rev\ (x : xs) \ ++\ ys = aux\ (x : xs, ys)$$

From the definition and associativity of  $++$ , we know that

$$\begin{aligned} rev\ (x : xs) \ ++\ ys &= (rev\ xs \ ++\ [x]) \ ++\ ys \\ &= rev\ xs \ ++\ ([x] \ ++\ ys) = rev\ xs \ ++\ (x : ys) \\ aux\ (x : xs, ys) &= aux\ (xs, x : ys) \end{aligned}$$

But an instance of the induction hypothesis ( $\forall$ -Elimination) is

$$rev\ xs \ ++\ (x : ys) = aux\ (xs, x : ys)$$

**QED**

# Abstraction

## Until now . . .

- We have only seen simple structuring techniques
  - ▶ sufficient though to construct all programs
- Difference: language **expressiveness** versus **usability/eloquence**
  - ▶ Assembler versus modern programming languages
  - ▶ Quick sort with/without comprehension
 
$$\begin{aligned}
 q \ [] &= [] \\
 q \ (p:xs) &= q \ [x \mid x \leftarrow xs, x \leq p] \ ++ \ [p] \ ++ \ q \ [x \mid x \leftarrow xs, x > p]
 \end{aligned}$$
- We will now examine different ways of
  - ▶ structuring programs
  - ▶ simplifying programs
  - ▶ improving their reusability

# Higher-order functions

**First order:** Arguments are basis types or constructor types

$$\text{Int} \rightarrow [\text{Int}]$$

**Second order:** Arguments can be themselves functions

$$(\text{Int} \rightarrow \text{Int}) \rightarrow [\text{Int}]$$

**Third order:** Arguments may be functions, whose arguments are functions

$$((\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}) \rightarrow [\text{Int}]$$

**Higher-order functions:** Functions of arbitrary order

## Example: **map**

```
double :: [Int] -> [Int]
double []      = []
double (x:xs) = (2*x) : double xs
```

```
triple :: [Int] -> [Int]
triple []      = []
triple (x:xs) = (3*x) : triple xs
```

Same control structure, only different function application

$$\textit{double} \ [x_1, \dots, x_n] = [2 \cdot x_1, \dots, 2 \cdot x_n]$$

$$\textit{triple} \ [x_1, \dots, x_n] = [3 \cdot x_1, \dots, 3 \cdot x_n]$$

## Control structure can be abstracted

```
map f []      = []      -- higher order
map f (x:xs) = f x : map f xs -- (function f is an argument)
```

```
times2 x = 2 * x
times3 x = 3 * x
```

```
double xs = map times2 xs
triple xs = map times3 xs
```

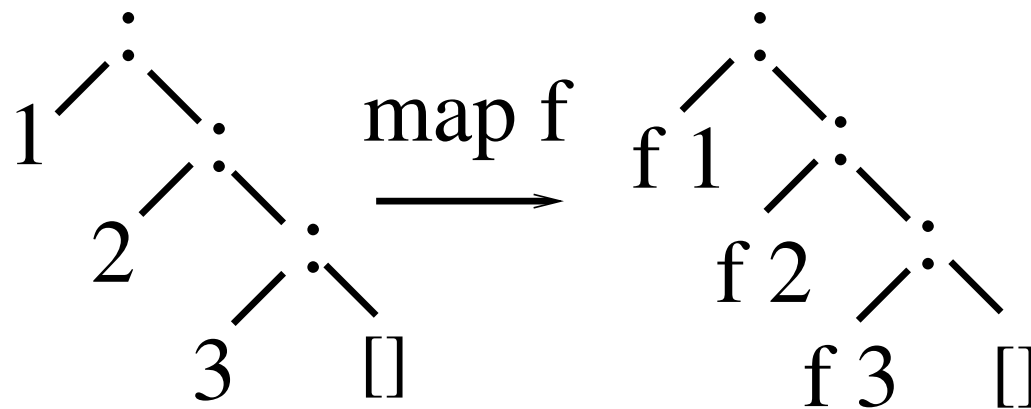
### Example of execution

```
map times2 [2,3]
= times2 2 : map times2 [3]
= 4 : map times2 [3]
= 4 : (times2 3 : map times2 [])
= 4 : (6 : map times2 [])
= 4 : (6 : [])
= [4,6]
```

## Visualizing map

```
map f []      = []
map f (x:xs) = f x : map f xs
```

$$\text{map } f [1, 2, 3] = [f\ 1, f\ 2, f\ 3]$$



Note similarity to list comprehension:  $\text{map } f\ xs = [f\ x \mid x \leftarrow xs]$

# Why functions as arguments?

```
-- without map
double []      = []
double (x:xs) = (2*x) : double xs
```

```
-- with map
double xs = map times2 xs
```

## Advantages:

1. Definition is easier to understand
2. Parts are easier to modify
3. Parts are easier to reuse
4. Correctness is simpler to understand and show



## Example: folding

- Consider `sumList [1,2,3] = 1+2+3`

`sumList [] = 0`

`sumList (x:xs) = x + sumList xs`

**Is this an instance of map?**

- Generalization

$$f [x_1, x_2, \dots, x_k] = f(x_1, f(x_2, \dots, f(x_k, 0) \dots))$$

E.g. `sumList [1,2,3] = f(1, f(2, f(3, 0)))`, for `f = (+)`

- Program

`foldr :: (a -> b -> b) -> b -> [a] -> b`

`foldr f z [] = z`

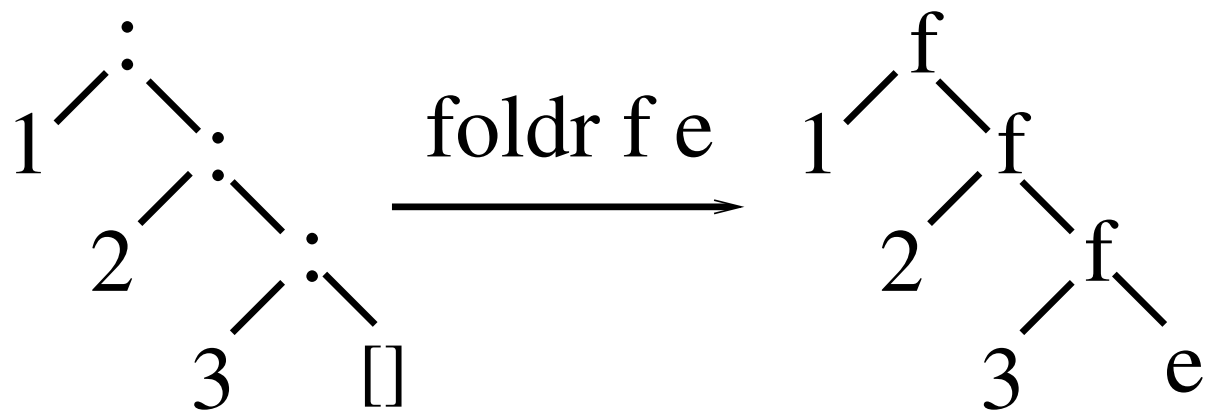
`foldr f z (x:xs) = f x (foldr f z xs)`

`sumList xs = foldr (+) 0 xs`

# Visualizing foldr

```
foldr f z []      = z
foldr f z (x:xs) = f x (foldr f z xs)
```

$$\text{foldr } f \ e \ [1, 2, 3] = f(1, f(2, f(3, e)))$$



## Examples of foldr

We can now easily define some standard Haskell functions:

```
concat xs = foldr (++) [] xs  
? concat [[1,2,3],[4],[5,6]]  
[1,2,3,4,5,6] :: [Int]
```

```
and bs = foldr (&&) True bs  
? and [True,True,False]  
False :: Bool
```

```
or bs = foldr (||) False bs  
? or [True,True,False]  
True :: Bool
```

```
pcons (x,y) (xs,ys) = (x:xs, y:ys)  
unzip xs = foldr pcons ([],[]) xs  
? unzip [(1,2),(3,4),(5,6)]  
([1,3,5],[2,4,6]) :: ([Int],[Int])
```

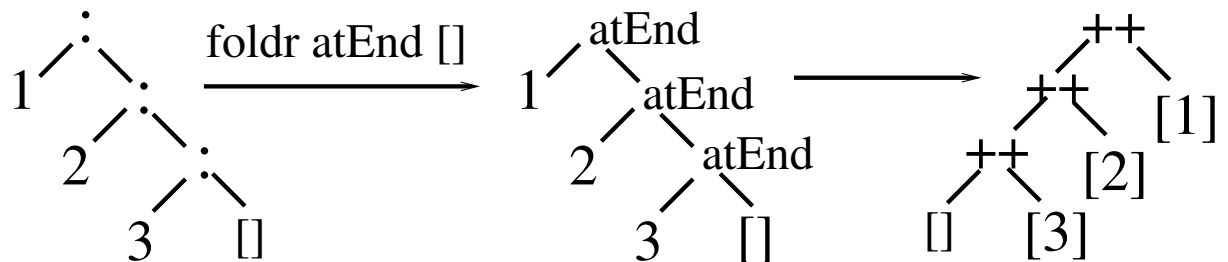
## Example: reverse

```
rev []      = []
rev (x:xs) = rev xs ++ [x]
```

$$\begin{aligned}
 rev\ [1, 2, 3] &= rev\ [2, 3] ++ [1] \\
 &= (rev\ [3] ++ [2]) ++ [1] \\
 &= ((rev\ [] ++ [3]) ++ [2]) ++ [1] \\
 &= ((([] ++ [3]) ++ [2]) ++ [1]) \\
 &= \dots \\
 &= [3, 2, 1]
 \end{aligned}$$

Suggests program:

```
atEnd x xs = xs ++ [x]
rv xs = foldr atEnd [] xs
```



## Example execution

```

atEnd x xs = xs ++ [x]           foldr f z []           = z
rv xs = foldr atEnd [] xs       foldr f z (x:xs) = f x (foldr f z xs)

```

Executes as follows:

```

rv [1,2,3] = foldr atEnd [] [1,2,3]
           = atEnd 1 (foldr atEnd [] [2,3])
           = (foldr atEnd [] [2,3]) ++ [1]
           = (atEnd 2 (foldr atEnd [] [3])) ++ [1]
           = ((foldr atEnd [] [3]) ++ [2]) ++ [1]
           = (atEnd 3 (foldr atEnd [] [])) ++ [2] ++ [1]
           = (((foldr atEnd [] []) ++ [3]) ++ [2]) ++ [1]
           = ([] ++ [3]) ++ [2] ++ [1]
           = ...
           = [3,2,1]

```

Correctness: prove  $\forall xs. \text{rev } xs = \text{rv } xs$

## Example: composition and iteration

- Pattern and control structure for function application

```
fapp f g x = f (g x)
```

```
iter 0 f x = x
```

```
iter n f x = f (iter (n-1) f x)
```

- Examples

```
times2 x = 2 * x
```

```
times3 x = 3 * x
```

```
? fapp times2 times3 5
```

```
30 :: Int
```

```
? iter 4 times2 1
```

```
16 :: Int
```

```
? iter 4 times3 1
```

```
81 :: Int
```

# Polymorphic types and reusability

- Consider following example:

```
length []      = 0
length (x:xs) = 1 + length xs
```

```
? length [17,3,149]
3 :: Int
```

```
? length ["eat","the","potato","Jane"]
4 :: Int
```

- What is type? `[Int] -> Int`, `[String] -> Int`, ...
  - ▶ The type is **polymorphic**: `[t] -> Int`, **for all types  $t$** .
- Often called **parametric polymorphism**. Differs from **subtyping polymorphism** (e.g., in Java), where methods can be applied to objects only of sub-classes.

## Types and reusability (cont.)

- Polymorphic types contain type variables

`length :: [t] -> Int`

- Function typeable for all instances.

**Definition:** A type  $w$  for  $f$  is a **most general** (also called **principal**) **type** iff for all types  $s$  for  $f$ ,  $s$  is an instance of  $w$ .

- Haskell has algorithms for **type checking** and **type reconstruction**
  - ▶ Haskell computes the principal type  $w$ , given a function  $f$
  - ▶ If user provides a type  $t$ , then  $t$  must be an instance of  $w$ .  
I.e., one can only restrict the type
- We will look at this in more detail later



# Polymorphic types — examples

- Functions over lists, e.g.  $(++) :: [t] \rightarrow [t] \rightarrow [t]$
- Functions defined by gluing together other functions

```
? :type iter  
Int -> (a -> a) -> a -> a
```

```
? :type fapp  
(a -> b) -> (c -> a) -> c -> b
```

- Or applying functions to data

```
? :type map  
(a -> b) -> [a] -> [b]
```

# Functions as values

- Functions can be returned as values
- Example: function composition (`fapp`)

$$(f \circ g) x = f (g x)$$

```
(.)           :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x     = f (g x)
```

- Examples

```
twice :: (t -> t) -> (t -> t)
twice f = f . f
```

```
? twice times3 1
9 :: Int
? (twice . twice) times3 1      -- 3^4 = 81
81 :: Int
? twice (times3 . times3) 1
81 :: Int
```

## Functions as values (cont.)

- Iter (higher-order argument)

```
iter 0 f x = x
iter n f x = f (iter (n-1) f x)
```

- Iter (higher-order argument and result)

```
id x = x

iter :: Int -> (a -> a) -> a -> a
iter 0 f = id
iter n f = f . (iter (n-1) f)
```

- Functions cannot be displayed, but they can be applied

```
? iter 2 times2
... Error ...
```

```
? let f = iter 2 times2 in f 5
20 :: Int
```

## Evaluation example

```
iter 0 f = id
iter n f = f . (iter (n-1) f)
```

```
let f = iter 2 times2 in f 5
```

### Can be calculated as follows

```
let f = iter 2 times2 in f 5  = (iter 2 times2) 5
                               = (times2 . (iter 1 times2)) 5
                               = (times2 . (times2 . (iter 0 times2))) 5
                               = (times2 . (times2 . id )) 5
                               = times2 ((times2 . id ) 5)
                               = times2 (times2 ( id 5))
                               = times2 (times2 5)
                               = times2 10
                               = 20
```

# $\lambda$ -expressions

- Consider the following functions

```
times2 x = 2 * x
double xs = map times2 xs
```

- Haskell provides notation to write functions like `times2` in-line

```
? map (\x -> x * 2) [2,3,4]
[4,6,8] :: Int
```

```
? foldr (\x y -> x * y) 1 [1,2,3,4]
24 :: Int
```

N.B.: `\x y -> x * y` is a shorthand for `\x -> \y -> x * y`

- Church's  $\lambda$ -notation (character “ $\backslash$ ” used instead of “ $\lambda$ ”)

	usual	$\lambda$ -calculus
Declaration	$f(x) = x + 3$	$\lambda x. x + 3$
Application	$f(5)$	$(\lambda x. x + 3)(5)$
Reduction	$(x + 3)[x \leftarrow 5]$	$(x + 3)[x \leftarrow 5]$
Result	8	8

# Partial application

- Functions of multiple arguments . . .

```
multiply :: Int -> Int -> Int  
multiply a b = a * b
```

- . . . can be partially applied

```
? :type multiply 7  
Int -> Int
```

```
? :type map  
(a -> b) -> [a] -> [b]
```

```
? map (multiply 7) [1,2,3,4]  
[7, 14, 21, 28] :: [Int]
```

- Application and types

If  $f :: t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n \rightarrow t$     and     $e_1 :: t_1, \dots, e_k :: t_k$   
then  $f \ e_1 \dots e_k :: t_{k+1} \rightarrow \dots \rightarrow t_n \rightarrow t$

# How many arguments do functions have?

- Each function takes exactly one argument
  - ▶ `multiply :: Int -> Int -> Int` means  
`multiply :: Int -> (Int -> Int)`
  - ▶ Application `multiply 2 3` means `(multiply 2) 3`
  - ▶ Partial application is consistent with the view (= illusion) that functions take multiple arguments
- Operator sections: if  $\oplus$  is an infix binary operator

$$(a \oplus) \equiv \lambda x. a \oplus x$$

$$(\oplus a) \equiv \lambda x. x \oplus a$$

- Example

```
? map ((2*) . (3*)) [1,2,3]  
[6,12,18]
```

## Multiple arguments versus tupling

$f :: (\text{Int}, \text{Int}) \rightarrow \text{Int}$   
 $f (x, y) = x * y + 17$

$g :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$   
 $g x y = x * y + 17$

- Tuple arguments: no partial application
- But equivalent in the following sense:

$\text{curry} :: ((a, b) \rightarrow c) \rightarrow a \rightarrow b \rightarrow c$   
 $\text{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow (a, b) \rightarrow c$

$\text{curry } f = f' \text{ where } f' x1 x2 = f (x1, x2)$   
 $\text{uncurry } f' = f \text{ where } f (x1, x2) = f' x1 x2$

- “Curry” like “Haskell Brooks Curry”





# Examples

```
f :: (Int,Int) -> Int
f (x,y) = x * y + 17
```

```
g :: Int -> Int -> Int
g x y = x * y + 17
```

```
? f (3,4)
29 :: Int
```

```
? curry f 3 4
29 :: Int
```

```
? g (3,4)
... Error ...
```

```
? g 3 4
29 :: Int
```

```
? uncurry g (3,4)
29 :: Int
```