Functional Data Structures

What to Cover

- Defining functional data structures
- Pattern matching
- Data sharing
- Recursing over lists and generalizing to higher-order functions
- Algebraic Data Types

Functional Data Structures

A functional data structure is operated on using only pure functions.

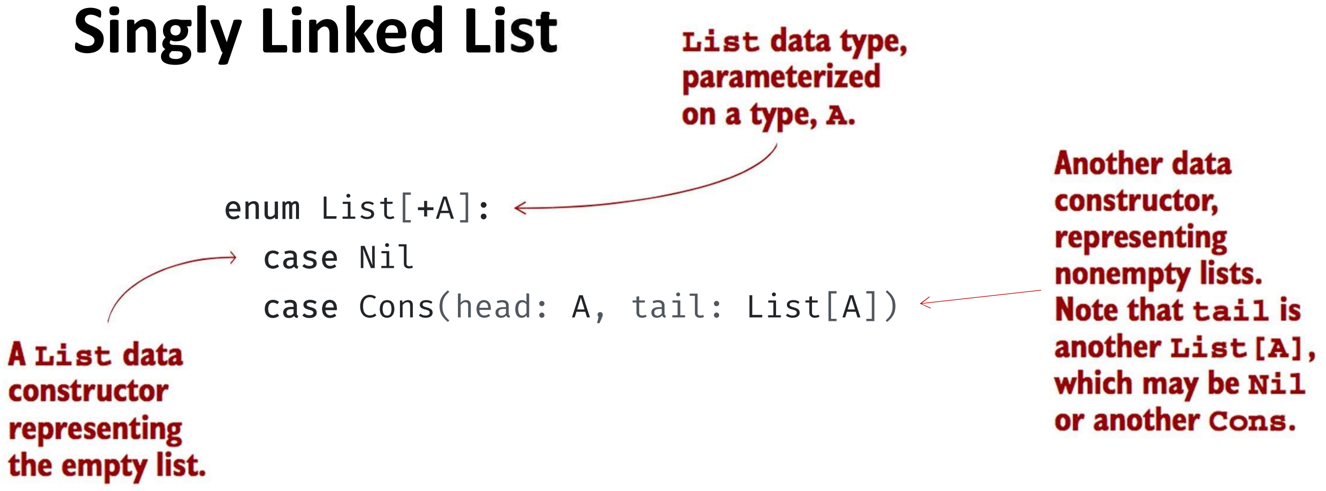
Therefore, functional data structures are by definition ***immutable****.*

Example

The empty list (**List()** or **Nil**) is as eternal and immutable as the integer values 3 or 4.

Just as 3 + 4 results in a new number 7 without modifying either 3 or 4, concatenating two lists together (**a ++ b** for two lists **a** and **b**) yields a new list and leaves the two inputs unmodified.

Singly Linked List



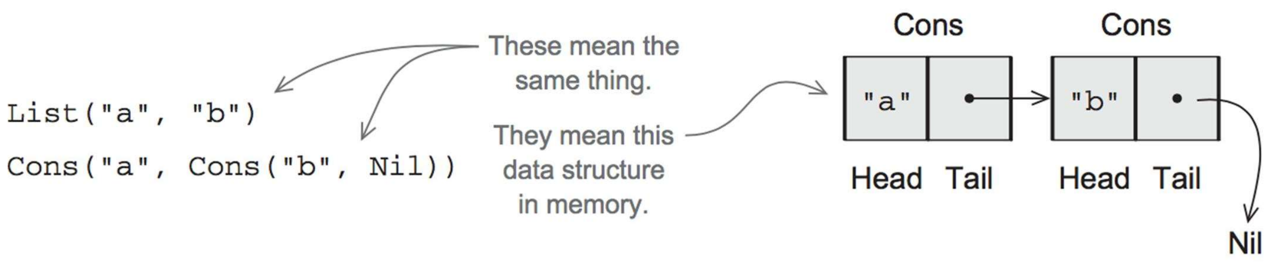
```
enum List[+A]:
  case Nil
  case Cons(head: A, tail: List[A])
```

List data type, parameterized on a type, A.
Another data constructor, representing nonempty lists. Note that tail is another List[A], which may be Nil or another Cons.
A List data constructor representing the empty list.

```
// Scala 2
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

Data Constructors

```
val ex1: List[Double] = Nil
val ex2: List[Int] = Cons(1, Nil)
val ex3: List[String] = Cons("a", Cons("b", Nil))
```



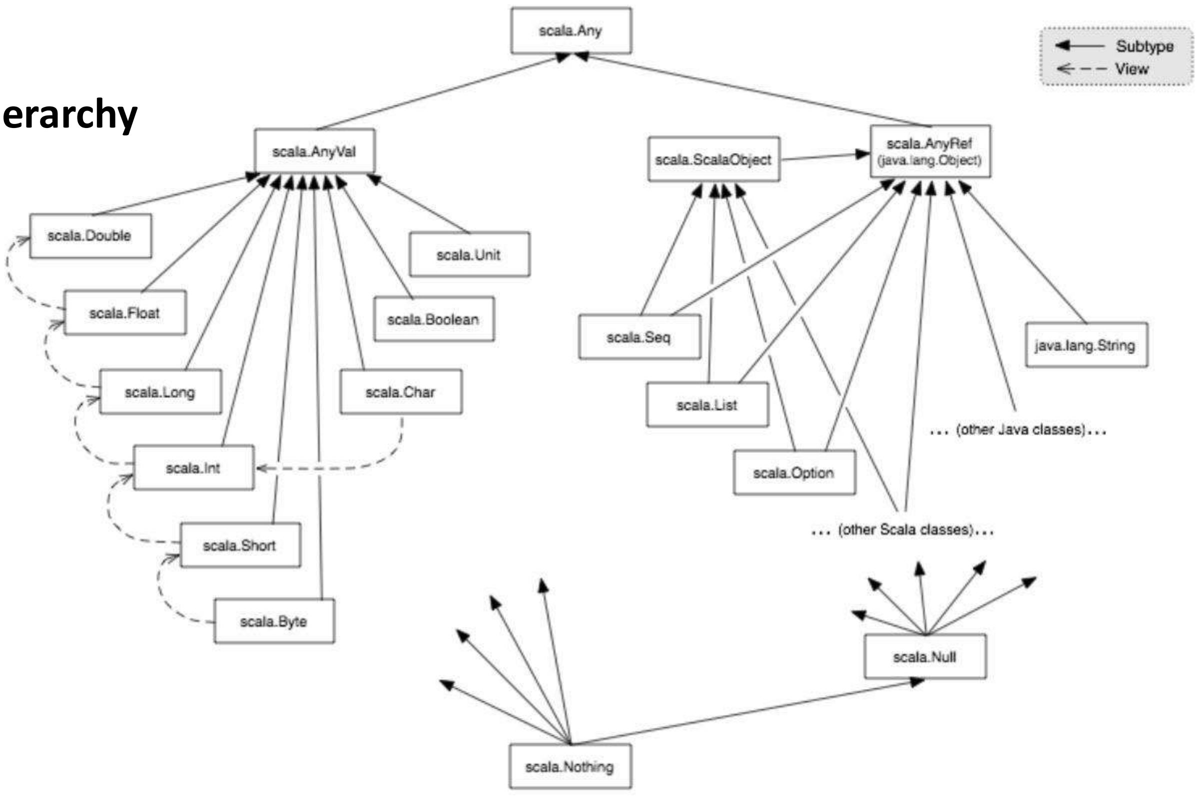
List("a", "b")
Cons("a", Cons("b", Nil))
These mean the same thing.
They mean this data structure in memory.
Cons
Cons
"a"
"b"
Head
Tail
Head
Tail
Nil

Variance Annotations

- In trait **List[+A]**, the **+** in front of the type parameter **A** is a **variance annotation** that indicates that **A** is a **covariant**.
- **Covariant**: For all types **X** and **Y**,

  if **X** is a subtype of **Y**, then **List[X]** is a subtype of **List[Y]**.
- If not annotated, parameter is **invariant**, meaning there is no subtyping relationship **List[X]** and **List[Y]**.
- **Nil** extends **List[Nothing]**. Since **Nothing** is a subtype of all types, in conjunction with the *variance annotation*, **Nil** can be considered a subtype of any **List[XXX]**.

Scala
Type Hierarchy



Companion Object

- We'll often declare a **companion object** in addition to our data type and its data constructors.
- The companion object is the one with *the same name as the data type* (in this case **List**) where we put various convenience functions for creating or working with values of the data type.

```
object List:
  def sum(xs: List[Int]): Int = ???
  def product(xs: List[Double]): Double = ???
  def apply[A](as: A*): List[A] = ???
  …
```

Pattern Matching

```
def sum(xs: List[Int]): Int = xs match
  case Nil => 0
  case Cons(x, xsl) => x + sum(xsl)
```

Pattern matching descends into the structure of the expression it examines and extract subexpressions of that structure.

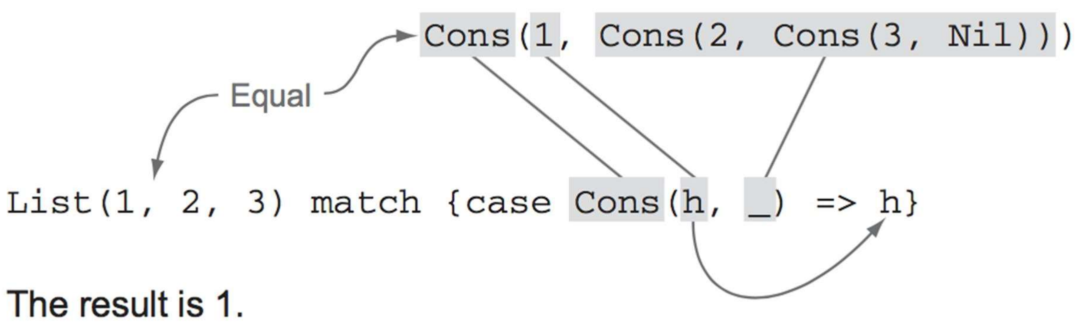
```
target match { pattern => result; … }
```

If the target *matches* the pattern in a case, the result of that case becomes the result of the entire match expression.

If multiple patterns match the target, Scala chooses the first matching case.

Pattern Matching Example

```
List(1,2,3) match { case Cons(h,_) => h }
```


Cons(1, Cons(2, Cons(3, Nil)))
Equal
List(1, 2, 3) match {case Cons(h, _) => h}
The result is 1.

Pattern Matching Example

- `List(1,2,3) match { case _ => 42 }` results in **42**.
  Variable pattern, _, matches any expression
- `List(1,2,3) match { case Cons(h,_) => h }` results in **1**.
  Data constructor pattern in conjunction with variables to *capture* or *bind* a subexpression of the target
- `List(1,2,3)match{case Cons(_,t) => t}` results in **List(2,3)**.
- `List(1,2,3)match{case Nil => 42}` results in a **MatchError** at runtime.

Exercise

What will be the result of the following match expression?

```
val x = List(1,2,3,4,5) match
  case Cons(x, Cons(2, Cons(4, _))) => x
  case Nil => 42
  case Cons(x, Cons(y, Cons(3, Cons(4, _)))) => x + y
  case Cons(h, t) => h + sum(t)
  case _ => 101
```

Variadic Functions

- The function **apply** in the companion object **List** is a **variadic function**, meaning it accepts zero or more arguments of type **A**:

```
def apply[A](as: A*): List[A] =
  if as.isEmpty then Nil
  else Cons(as.head, apply(as.tail*))
```

- Allows us to invoke it with syntax like **List(1,2,3,4)** or **List("hi","bye")**, with as many values as we want separated by commas (we sometimes call this the **list literal** or just **literal syntax**).

Variadic Functions (cont'd)

```
def apply[A](as: A*): List[A] =
  if as.isEmpty then Nil
  else Cons(as.head, apply(as.tail*))
```

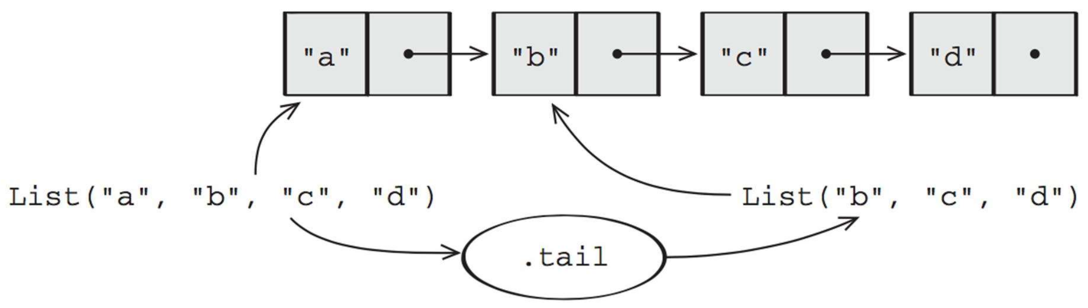
Variadic functions provides a little syntactic sugar for creating and passing a **Seq** of elements explicitly.

The argument will be bound to a **Seq[A]**.

Seq has functions like **head**, **tail**, **isEmpty**, etc.

The special ***** type annotation allows us to pass a **Seq** to a variadic method.

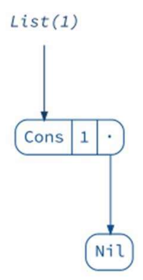
Data Sharing and Persistent Data Structures

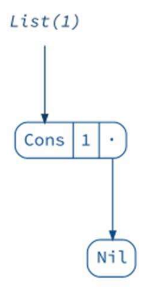


Both lists share the same data in memory. `.tail` does not modify the original list, it simply references the tail of the original list. Defensive copying is not needed, because the list is immutable.

Functional data structures are **persistent**, meaning that existing references are never changed by operations on the data structure.

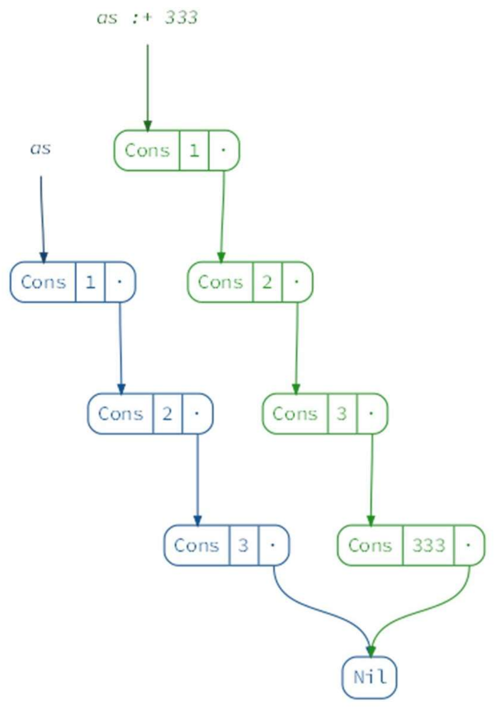
Prepending vs. Appending





Append

```
val as = List(1, 2, 3)
val bs = as :+ 333
```



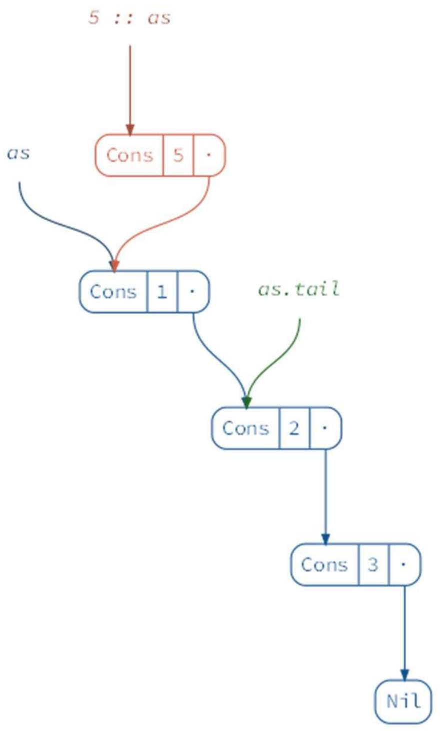
Exercises

[3-2] Implement the function **tail** for removing the first element of a **List**. Note that the function takes ***constant time***. What are different choices you could make in your implementation if the **List** is **Nil**?

[3-3] Using the same idea, implement the function **setHead** for replacing the first element of a **List** with a different value.

Data Sharing

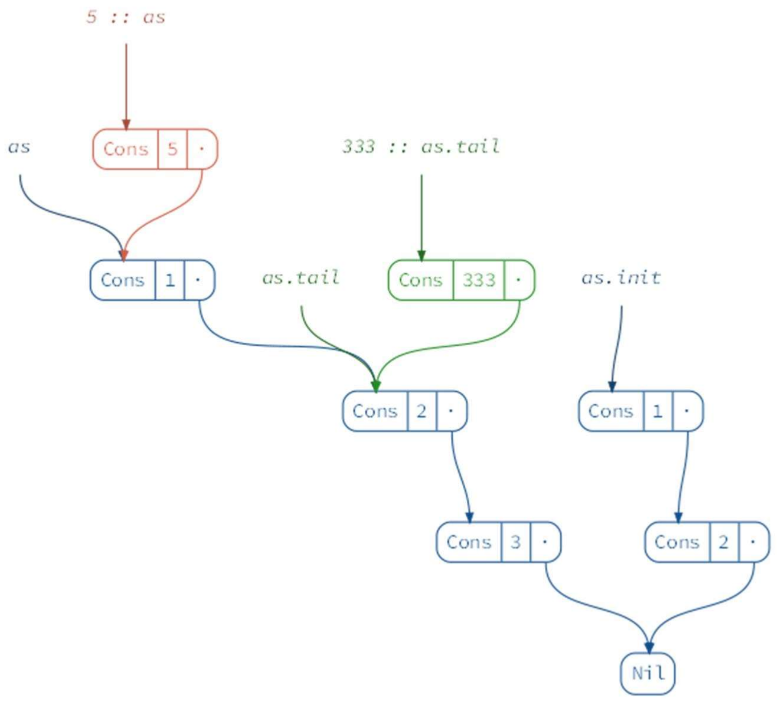
```
val as = List(1, 2, 3)
val tail = as.tail
val bs = 5 :: as
```



tail vs. init

```
val as = List(1, 2, 3)
val tail = as.tail
val bs = 5 :: as

val init_as = as.init
val cs = 333 :: as.tail
```



Efficiency of Data Sharing

append adds all the elements of one list to the end of another:

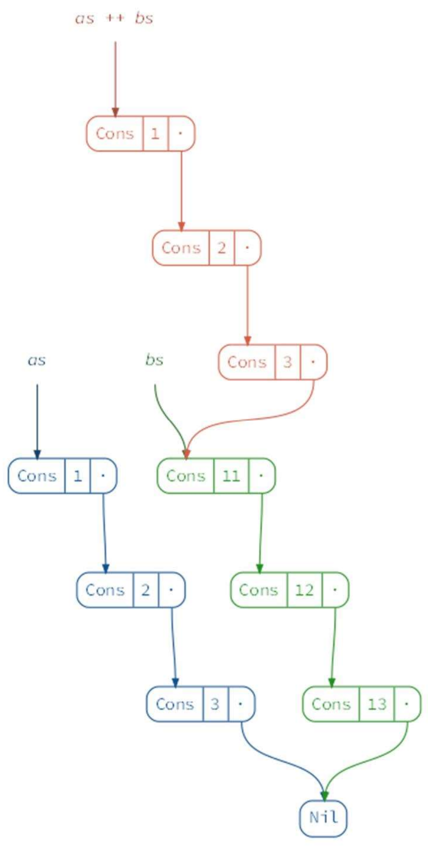
```
def append[A](a1: List[A], a2: List[A]): List[A] = a1 match
  case Nil => a2
  case Cons(h,t) => Cons(h, append(t, a2))
```

The run-time and memory usage are proportional to the length of **a1**.

What if we were to implement this same function for two arrays?

concatenation

```
val a = List(1, 2, 3)
val b = List(11, 12, 13)
val c = a ++ b
```



Exercise (Unhappy Scenario)

Implement a function, **init**, that returns a **List** consisting of all but the last element of a **List**.

```
init(List(1,2,3,4)) == List(1,2,3)
```

Why can't this function be implemented in constant time like **tail**?

```
def init[A](l: List[A]): List[A]
```

Efficiency of Data Sharing

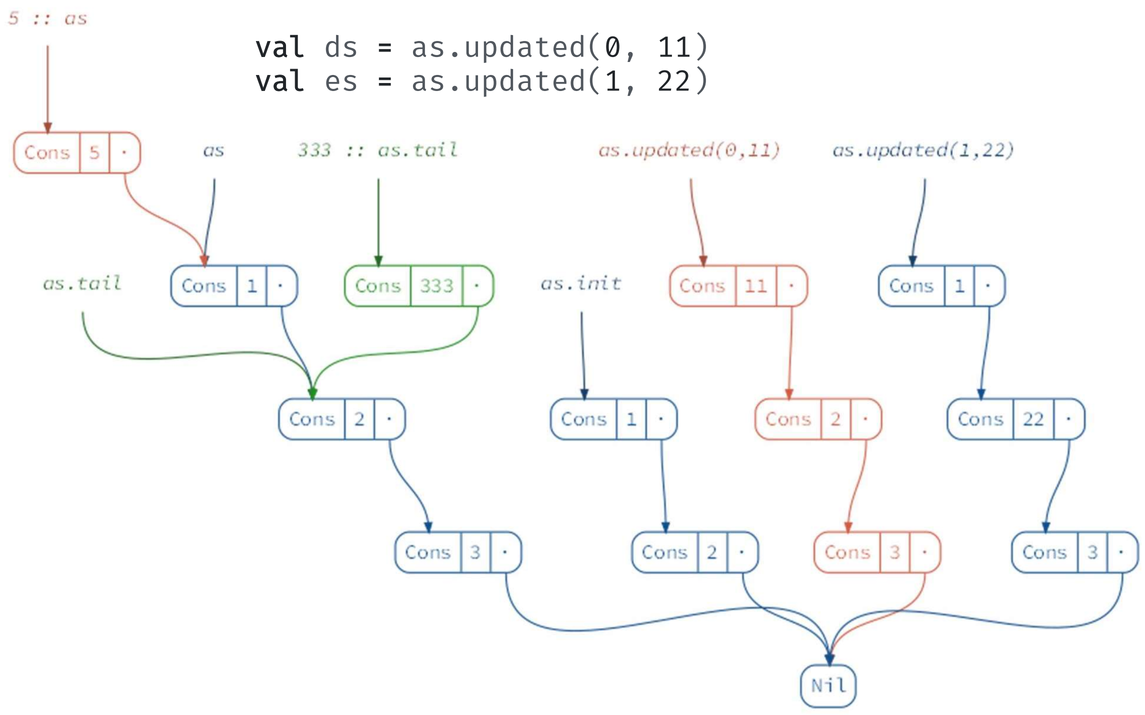
Because of the structure of a singly linked list, any time we want to replace the tail of a **Cons**, even if it's the last **Cons** in the list, we must copy all the previous **Cons** objects.

Writing purely functional data structures that support different operations efficiently is all about finding clever ways to exploit data sharing.

Vector in Scala standard library is a purely functional sequence implementation with constant-time random access, updates, **head**, **tail**, **init**, and constant-time additions to either the front or rear of the sequence.

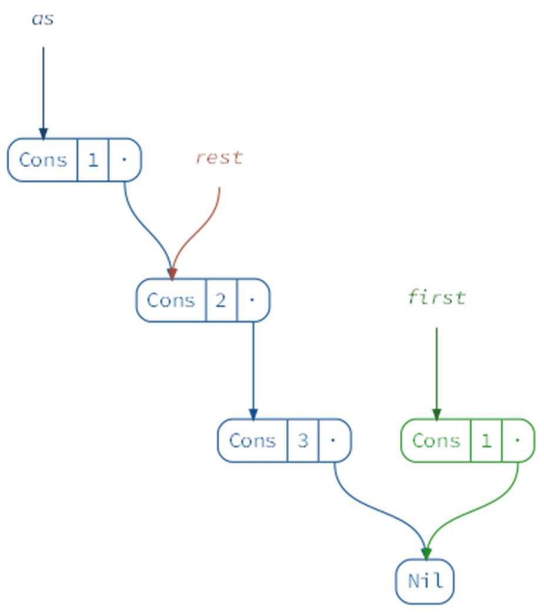
updated

```
val ds = as.updated(0, 11)
val es = as.updated(1, 22)
```



splitAt

```
val as = List(1, 2, 3)
val (first, rest) = as.splitAt(1)
```



Improving Type Inference for HOFs

```
def dropWhile[A](l: List[A], f: A => Boolean): List[A]

val xs: List[Int] = List(1,2,3,4,5)
val ex1 = dropWhile(xs, (x: Int) => x < 4)
```

Use curried form to maximize type inference.

```
def dropWhile[A](as: List[A])(f: A => Boolean): List[A]

val xs: List[Int] = List(1,2,3,4,5)
val ex1 = dropWhile(xs)(x => x < 4)
```

Currying and Type Inference

The main reason for grouping the arguments this way is to assist with type inference.

More generally, when a function definition contains multiple argument groups, **type information flows from left to right** across these argument groups.

Here, the first argument group fixes the type parameter **A** of **dropWhile** to **Int**, so the annotation on **x => x < 4** is not required.

Don’t Repeat Yourself (DRY)

```
def sum(ints: List[Int]): Int = ints match
  case Nil => 0
  case Cons(x,xs) => x + sum(xs) // +(x, sum(xs))
  // +(1, +(2, +(3, sum(Nil)))

def product(ds: List[Double]): Double = ds match
  case Nil => 1.0
  case Cons(x,xs) => x * product(xs) // *(x, product(xs))
  // *(1, *(2, *(3, product(Nil)))
```

Generalizing Away

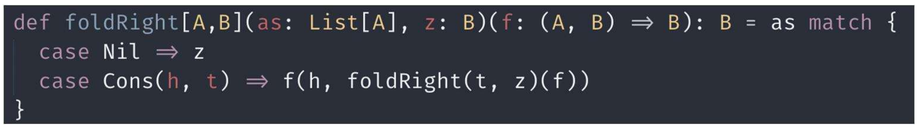
- Whenever you encounter duplication like this, you can **generalize it away** by pulling subexpressions out into function arguments.
- If a subexpression refers to any local variables (e.g., the + operation refers to the local variables **x** and **xs**), turn the subexpression into a function that accepts these variables as arguments.

```
def foldRight[A,B](as: List[A], z: B)(f: (A, B) => B): B =
  as match
    case Nil => z
    case Cons(h,t) => f(h, foldRight(t, z)(f))
```

foldRight

$$\begin{tikzcd}
 & op \arrow[dl] \arrow[dr] & & & & \\
x_1 \arrow[dr, bend right] & & op \arrow[dl] \arrow[dr] \arrow[ul, bend right] & & & \\
 & x_2 \arrow[dr, bend right] & & op \arrow[dl] \arrow[dr] \arrow[ul, bend right] & & \\
 & & x_3 \arrow[dr, bend right] & & op \arrow[dl] \arrow[dr] \arrow[ul, bend right] & \\
 & & & x_4 & & z \arrow[ul, bend right]
\end{tikzcd}$$

Execution Trace



```
def foldRight[A,B](as: List[A], z: B)(f: (A, B) ⇒ B): B = as match {
  case Nil ⇒ z
  case Cons(h, t) ⇒ f(h, foldRight(t, z)(f))
}
```

```
foldRight(Cons(1, Cons(2, Cons(3, Nil))), 0)((x,y) => x + y)
1 + foldRight(Cons(2, Cons(3, Nil)), 0)((x,y) => x + y)
1 + (2 + foldRight(Cons(3, Nil), 0)((x,y) => x + y))
1 + (2 + (3 + (foldRight(Nil, 0)((x,y) => x + y))))
1 + (2 + (3 + (0)))
6
```

Replace foldRight with its definition.

One way of describing what **foldRight** does is that it replaces the constructors of the list, **Nil** and **Cons**, with **z** and **f**, illustrated here:

```
Cons(1, Cons(2, Nil))
f   (1, f   (2, z  ))
```

Right Folds and Simple Uses

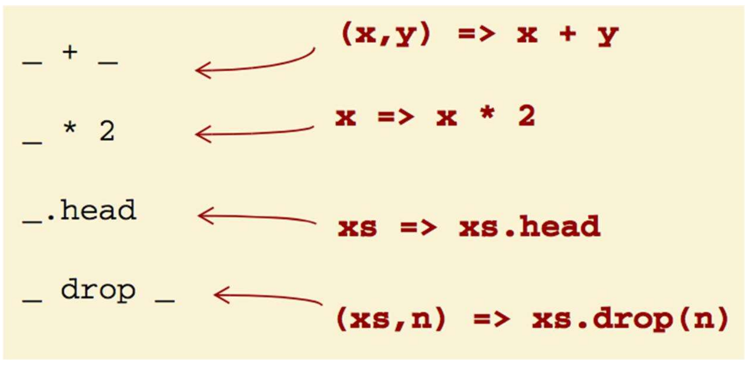
```
def sum2(ns: List[Int]) =
      foldRight(ns, 0)((x,y) => x + y)

def product2(ns: List[Double]) =
      foldRight(ns, 1.0)(_ * _)
```

_ * _ is more concise
notation for (x,y) => x * y

Underscore Notation for Anonymous Functions

- The anonymous function **(x,y) => x + y** can be written as **_ + _** in situations where the types of x and y could be inferred by Scala.
- This is a useful shorthand in cases where the function parameters are **mentioned just once** in the body of the function.



```
_ + _        <-   (x,y) => x + y
_ * 2        <-   x => x * 2
_.head       <-   xs => xs.head
_ drop _     <-   (xs,n) => xs.drop(n)
```

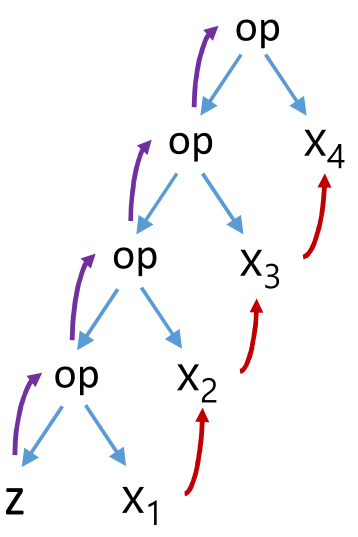
foldLeft

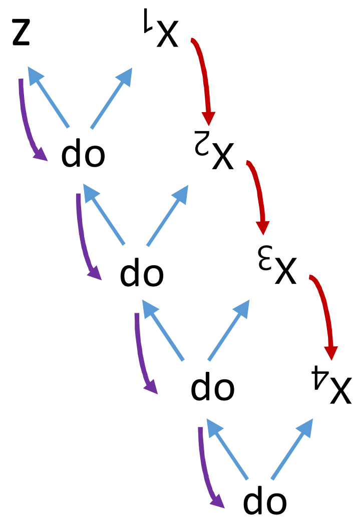
```
def foldLeft[A,B](as: List[A], z: B)(f: (B, A) => B): B

    foldLeft(List(1,2,3), 0)(_ + _)
  = foldLeft(List(2,3), 0 + 1)(_ + _)
  = foldLeft(List(3), (0 + 1) + 2)(_ + _)
  = foldLeft(Nil, ((0 + 1) + 2) + 3)(_ + _)
  = ((0 + 1) + 2) + 3
```

```
def foldLeft[A,B](as: List[A], z: B)(f: (B, A) ⇒ B): B = as match {
  case Nil ⇒ z
  case Cons(h, t) ⇒ foldLeft(t, f(z, h))(f)
}
```

foldLeft





Exercises

[Ex3-9] Compute the length of a list using **foldRight**.

def length[A](as: List[A]): Int

[Ex3-11] Write **sum**, **product**, and a function to compute the length of a list using **foldLeft**.

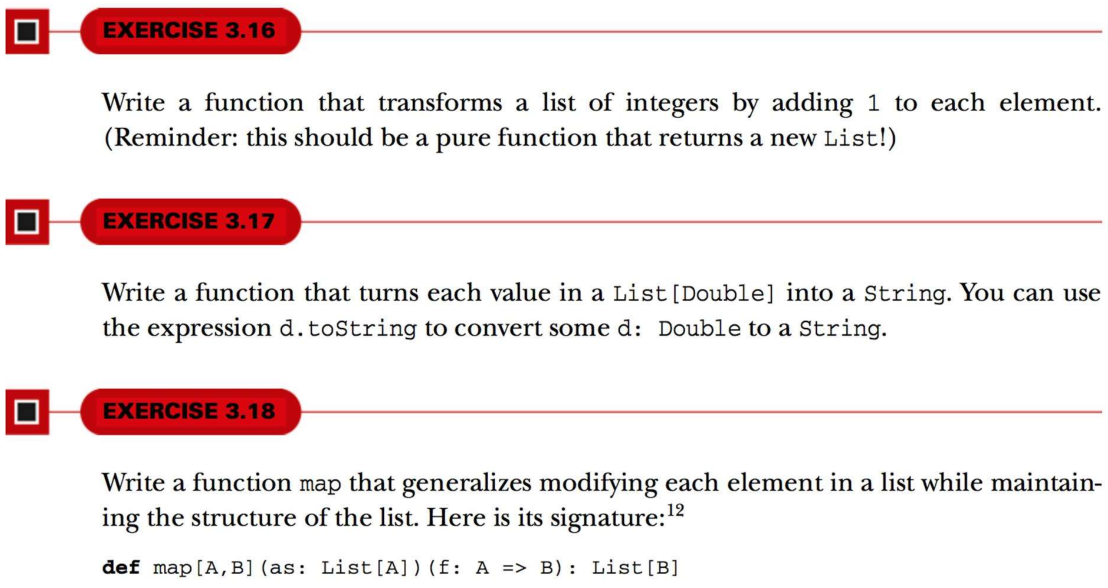
[Ex3-12] Write a function that returns the reverse of a list.

[Ex3-13] *Hard:* Can you write **foldLeft** in terms of **foldRight**? How about the other way around?

[Ex3-14] Implement **append** in terms of either **foldLeft** or **foldRight**.

[Ex3-15] *Hard:* Write a function that concatenates a list of lists into a single list. Its runtime should be linear in the total length of all lists.

More Exercises



EXERCISE 3.16

Write a function that transforms a list of integers by adding `1` to each element. (Reminder: this should be a pure function that returns a new `List`!)

EXERCISE 3.17

Write a function that turns each value in a `List[Double]` into a `String`. You can use the expression `d.toString` to convert some `d: Double` to a `String`.

EXERCISE 3.18

Write a function `map` that generalizes modifying each element in a list while maintaining the structure of the list. Here is its signature:[12]

```
def map[A,B](as: List[A])(f: A => B): List[B]
```

EXERCISE 3.19

Write a function `filter` that removes elements from a list unless they satisfy a given predicate. Use it to remove all odd numbers from a `List[Int]`.

```
def filter[A](as: List[A])(f: A => Boolean): List[A]
```

EXERCISE 3.20

Write a function `flatMap` that works like `map` except that the function given will return a list instead of a single result, and that list should be inserted into the final resulting list. Here is its signature:

```
def flatMap[A,B](as: List[A])(f: A => List[B]): List[B]
```

For instance, `flatMap(List(1,2,3))(i => List(i,i))` should result in `List(1,1,2,2,3,3)`.

EXERCISE 3.21

Use `flatMap` to implement `filter`.

EXERCISE 3.22

Write a function that accepts two lists and constructs a new list by adding corresponding elements. For example, `List(1,2,3)` and `List(4,5,6)` become `List(5,7,9)`.

EXERCISE 3.23

Generalize the function you just wrote so that it's not specific to integers or addition. Name your generalized function `zipWith`.

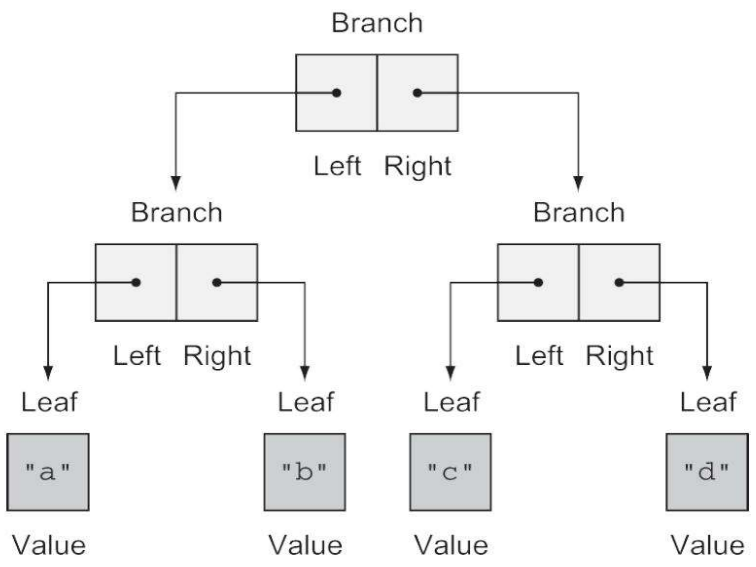
Trees

```
enum Tree[+A]:
  case Leaf(value: A)
  case Branch(left: Tree[A], right: Tree[A])

  def size: Int = this match
    case Leaf(_)       => 1
    case Branch(l, r) => 1 + l.size + r.size


 object Tree:
   extension (tree: Tree[Int])
     def firstPositive: Int = tree match
       case Leaf(i) => i
       case Branch(l, r) =>
         val lpos = l.firstPositive
         if lpos > 0 then lpos else r.firstPositive
```

Trees (cont'd)



```
Branch(Branch(Leaf("a"), Leaf("b")),
       Branch(Leaf("c"), Leaf("d")))
```