A Closer Look At Lists

Principles of Functional Programming

Lists Recap

Lists are the core data structure we will work with over the next weeks.

Type: `List[Fruit]`

Construction:

```
val fruits = List("Apple", "Orange", "Banana")
val nums = 1 :: 2 :: Nil
```

Decomposition:

```
fruits.head     // "Apple"
nums.tail       // 2 :: Nil
nums.isEmpty    // false

nums match
  case x :: y :: _ => x + y   // 3
```

List Methods (1)

Sublists and element access:

`xs.length`	The number of elements of `xs`.
`xs.last`	The list's last element, exception if `xs` is empty.
`xs.init`	A list consisting of all elements of `xs` except the last one, exception if `xs` is empty.
`xs.take(n)`	A list consisting of the first `n` elements of `xs`, or `xs` itself if it is shorter than `n`.
`xs.drop(n)`	The rest of the collection after taking `n` elements.
`xs(n)`	(or, written out, `xs.apply(n)`). The element of `xs` at index `n`.

List Methods (2)

Creating new lists:

`xs ++ ys`	The list consisting of all elements of `xs` followed by all elements of `ys`.
`xs.reverse`	The list containing the elements of `xs` in reversed order.
`xs.updated(n, x)`	The list containing the same elements as `xs`, except at index `n` where it contains `x`.

Finding elements:

`xs.indexOf(x)`	The index of the first element in `xs` equal to `x`, or `-1` if `x` does not appear in `xs`.
`xs.contains(x)`	same as `xs.indexOf(x) >= 0`

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match
  case List() => throw Error(”last of empty list”)
  case List(x) =>
  case y :: ys =>
```

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match
  case List() => throw Error(”last of empty list”)
  case List(x) => x
  case y :: ys =>
```

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match
  case List() => throw Error("last of empty list")
  case List(x) => x
  case y :: ys => last(ys)
```

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match
  case List() => throw Error("last of empty list")
  case List(x) => x
  case y :: ys => last(ys)
```

So, `last` takes steps proportional to the length of the list `xs`.

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match
  case List() => throw Error(”init of empty list”)
  case List(x) => ???
  case y :: ys => ???
```

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match
  case List() => throw Error(”init of empty list”)
  case List(x) =>
  case y :: ys =>
```

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match
  case List() => throw Error(”init of empty list”)
  case List(x) => List()
  case y :: ys =>
```

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match
  case List() => throw Error("init of empty list")
  case List(x) => List()
  case y :: ys => y :: init(ys)
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for ++:

```
extension [T](xs: List[T])
  def ++ (ys: List[T]): List[T] =
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for ++:

```
extension [T](xs: List[T])
  def ++ (ys: List[T]): List[T] = xs match
    case Nil =>
    case x :: xs1 =>
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for ++:

```
extension [T](xs: List[T])
  def ++ (ys: List[T]): List[T] = xs match
    case Nil => ys
    case x :: xs1 =>
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for ++:

```
extension [T](xs: List[T])
  def ++ (ys: List[T]): List[T] = xs match
    case Nil => ys
    case x :: xs1 => x :: (xs1 ++ ys)
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for ++:

```
extension [T](xs: List[T])
  def ++ (ys: List[T]): List[T] = xs match
    case Nil => ys
    case x :: xs1 => x :: (xs1 ++ ys)
```

What is the complexity of concat?

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for `++`:

```
extension [T](xs: List[T])
  def ++ (ys: List[T]): List[T] = xs match
    case Nil => ys
    case x :: xs1 => x :: (xs1 ++ ys)
```

What is the complexity of `concat`?

Answer: `O(xs.length)`

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing an extension method:

```
extension [T](xs: List[T])
  def reverse: List[T] = xs match
    case Nil =>
    case y :: ys =>
```

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing an extension method:

```
extension [T](xs: List[T])
  def reverse: List[T] = xs match
    case Nil => Nil
    case y :: ys =>
```

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing an extension method:

```
extension [T](xs: List[T])
  def reverse: List[T] = xs match
    case Nil => Nil
    case y :: ys => ys.reverse ++ List(y)
```

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing an extension method:

```
extension [T](xs: List[T])
  def reverse: List[T] = xs match
    case Nil => Nil
    case y :: ys => ys.reverse ++ List(y)
```

What is the complexity of `reverse`?

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing an extension method:

```
extension [T](xs: List[T])
  def reverse: List[T] = xs match
    case Nil => Nil
    case y :: ys => ys.reverse ++ List(y)
```

What is the complexity of `reverse`?

Answer: `O(xs.length * xs.length)`

Can we do better? (to be solved later).

Exercise

Remove the `n`'th element of a list `xs`. If `n` is out of bounds, return `xs` itself.

```
def removeAt[T](n: Int, xs: List[T]) = ???
```

Usage example:

```
removeAt(1, List('a', 'b', 'c', 'd'))
```

> *List(a, c, d)*

Exercise (Harder, Optional)

Flatten a list structure:

```
def flatten(xs: List[Any]): List[Any] = ???

flatten(List(List(1, 1), 2, List(3, List(5, 8))))
            >   res0: List[Any] = List(1, 1, 2, 3, 5, 8)
```