Lists

Jacek Wasilewski

Lists (1)

- Primary data structure
- For elements $x_1, x_2, \ldots, x_n$, list would be `List(`$x_1, x_2, \ldots, x_n$`).`
- Examples

```
val fruit = List("apples", "oranges", "pears”)
val nums = List(1, 2, 3, 4)
val diag3 = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty = List()
```

Lists (2)

- Properties
  - immutable
  - recursive structure
  - elements of the same type
- Many built-in functions

List Type

- List can have a specified type – using the generics notation.
- Example

```
val fruit: List[String] = List("apples", "oranges", "pears”)
val nums: List[Int] = List(1, 2, 3, 4)
val diag3: List[List[Int] =
    List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty: List[Int] = List()
```

List Constructors (1)

- Constructing lists
  `val nums: List[Int] = List(1, 2, 3, 4)`
- Structure
  - head
  - tail
- Head - single element
- Tail - everything else (as list)
- `List(1)` – element in the head, empty tail list

List Constructors (2)

- Empty list – **`Nil`**
  - same as **`List()`**
- Building list incrementally
  - by adding new element in front of it, to the head
  - using **`::`** operator – list extension
- Start with an empty list and extend it

List Constructors (3)

- Example

```
val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
val nums = 1 :: (2 :: (3 :: (4 :: Nil)))
val diag3 = (1 :: (0 :: (0 :: Nil))) ::
            (0 :: (1 :: (0 :: Nil))) ::
            (0 :: (0 :: (1 :: Nil))) :: Nil
val empty = Nil
```

- Need to end your list with `Nil`
- No need for parentheses

```
val nums = 1 :: 2 :: 3 :: 4 :: Nil
```

Basic Operations (1)

- Basic operations:
  - `head` – first element
  - `tail` – everything but the first element
  - `isEmpty` – true if empty

Basic Operations (2)

```
scala> fruit.isEmpty
res23: Boolean = false

scala> empty.isEmpty
res24: Boolean = true

scala> nums.head
res25: Int = 1

scala> nums.tail
res26: List[Int] = List(2, 3, 4)

scala> nums.tail.head
res27: Int = 2

scala> nums.tail.tail
res28: List[Int] = List(3, 4)
```

First Order Methods (1)

- `length` – length of the list
- `last` – last element of the list
- `init` – everything but the last

First Order Methods (2)

```
scala> val list: List[Int] = List(1, 2, 3, 4, 5)
list: List[Int] = List(1, 2, 3, 4, 5)

scala> list.length
res36: Int = 5

scala> list.last
res37: Int = 5

scala> list.init
res38: List[Int] = List(1, 2, 3, 4)

scala> list.init.tail
res39: List[Int] = List(2, 3, 4)
```

First Order Method (3)

- `take(n)` – first `n` elements of the list, or the whole list
- `drop(n)` – all elements except first `n`
- `splitAt(n)` – a tuple of two lists
  - first has `n` elements
  - second has the rest
  - first is equivalent to `take(n)`
  - second is equivalent to `drop(n)`

Tuple

- Tuple – generic structure for holding values
- Example

```
scala> val x = Tuple2[Int, String](1, "one")
x: (Int, String) = (1,one)
```

- Accessing values

```
scala> x._1
res0: Int = 1
scala> x._2
res1: String = one
```

- Built-in: `Tuple1` to `Tuple22`

First Order Method (4)

```
scala> val list: List[Int] = List(1, 2, 3, 4, 5)
list: List[Int] = List(1, 2, 3, 4, 5)

scala> list.take(3)
res40: List[Int] = List(1, 2, 3)

scala> list.drop(2)
res41: List[Int] = List(3, 4, 5)

scala> list.splitAt(3)
res43: (List[Int], List[Int]) = (List(1, 2, 3),List(4, 5))

scala> list.splitAt(3)._1
res44: List[Int] = List(1, 2, 3)
```

First Order Methods (5)

- `apply(n)` – element at position `n` (starting from 0)
- `(n)` – as above
- `zip(other)` – combines two lists into a list of pairs (tuples)
- `:::` - concatenates elements of two lists
- `reverse` – self-descriptive

First Order Methods (6)

```
scala> val list: List[Int] = List(1, 5, 2, 4, 3)
list: List[Int] = List(1, 5, 2, 4, 3)
scala> list.apply(0)
res46: Int = 1
scala> list.apply(1)
res47: Int = 5
scala> list.apply(3)
res48: Int = 4
scala> list(1)
res49: Int = 5
scala> list(3)
res50: Int = 4
```

First Order Methods (7)

```
scala> val listA = List("A","B","C","D")
listA: List[String] = List(A, B, C, D)

scala> val listB = List(1,2,3,4)
listB: List[Int] = List(1, 2, 3, 4)

scala> listA.zip(listB)
res51: List[(String, Int)] = List((A,1), (B,2), (C,3), (D,4))

scala> listB.zip(listA)
res52: List[(Int, String)] = List((1,A), (2,B), (3,C), (4,D))
```

First Order Methods (8)

```
scala> val list: List[Int] = List(1, 5, 2, 4, 3)
list: List[Int] = List(1, 5, 2, 4, 3)

scala> list.reverse
res53: List[Int] = List(3, 4, 2, 5, 1)
```

First Order Methods (9)

```
scala> val listA = List(5,4,3,2,1)
listA: List[Int] = List(5, 4, 3, 2, 1)

scala> val listB = List(6,7,8,9,10)
listB: List[Int] = List(6, 7, 8, 9, 10)


scala> listA :: listB
res55: List[Any] = List(List(5, 4, 3, 2, 1), 6, 7, 8, 9, 10)


scala> listA ::: listB
res54: List[Int] = List(5, 4, 3, 2, 1, 6, 7, 8, 9, 10)
```

Higher Order Methods

- Operations
  - transformations
  - filterings
  - computations

Mapping over Lists

- `map`
- What for: transform each element into a new one
- Example:

```
scala> val l = List(1,3,5,7)
l: List[Int] = List(1, 3, 5, 7)

scala> l.map(x => x * x)
res1: List[Int] = List(1, 9, 25, 49)

scala> l map {x => x * x}
res2: List[Int] = List(1, 9, 25, 49)
```

Iterating over Lists

- `foreach`
- What for: consume each element; list is not returned
- Example:

```
scala> val l = List(1,3,5,7)
l: List[Int] = List(1, 3, 5, 7)

scala> l.foreach(x => println(x))
1
3
5
7
```

Filtering Lists

- `filter`
- What for: filter with a condition
- Example:

```
scala> val l = List(1,3,5,7)
l: List[Int] = List(1, 3, 5, 7)

scala> l.filter(x => x < 4)
res5: List[Int] = List(1, 3)

scala> l filter {x => x < 4}
res6: List[Int] = List(1, 3)
```

Testing Predicates

- `forall` and `exists`
- What for: check if elements meet condition
- Example:

```
scala> val l = List(1,3,5,7)
l: List[Int] = List(1, 3, 5, 7)

scala> l forall {x => x < 4}
res7: Boolean = false

scala> l exists {x => x < 4}
res9: Boolean = true
```

Partitioning

- `partition`
- What for: split into two, based on the condition
- Example:

```
scala> val l = List(1,3,5,7)
l: List[Int] = List(1, 3, 5, 7)

scala> l.partition(x => x < 4)
res12: (List[Int], List[Int]) = (List(1, 3),List(5, 7))

scala> l.partition(x => x % 2 == 1)
res13: (List[Int], List[Int]) = (List(1, 3, 5, 7),List())
```

Reducing and Folding Lists (1)

- `reduceLeft` and `reduceRight`
- What for: combine elements into a value
- `reduceLeft` starts from the first element
- `reduceRight` starts from the last element
- Typically no difference

Reducing and Folding Lists (2)

- Example:

```
scala> val l = List(1,3,5,7)
l: List[Int] = List(1, 3, 5, 7)

scala> l.reduceLeft((x, y) => x + y)
res21: Int = 16

scala> l.reduceRight((x, y) => x + y)
res22: Int = 16

scala> (0 :: l).reduceLeft((x, y) => x + y)
res23: Int = 16
```

Reducing and Folding Lists (3)

- `foldLeft` and `foldRight`
- What for: combine elements into a value
- Reducing and folding - folding uses an accumulator, that needs to be defined, to store the result

Reducing and Folding Lists (4)

- Example:

```
scala> val l = List(1,3,5,7)
l: List[Int] = List(1, 3, 5, 7)

scala> (0 :: l).reduceLeft((x, y) => x + y)
res23: Int = 16

scala> l.foldLeft(0)((x, y) => x + y)
res29: Int = 16

scala> l.foldLeft(0)((x, y) => x * y)
res30: Int = 0

scala> l.foldLeft(1)((x, y) => x * y)
res31: Int = 105
```

Generators

- Lists of numbers can be generated
- `range`
- Example:

```
scala> List.range(1, 10)
res10: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Flattening Maps

- `flatMap`
- What for: map list of lists into a single list
- Example:

```
scala> val ll = List(List(1,2,3),List(4,5,6),List(7,8,9))
ll: List[List[Int]] = List(List(1, 2, 3), List(4, 5, 6), List(7, 8, 9))

scala> ll.flatMap(x => x)
res39: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Chaining

- Methods return new lists – e.g. filter, map
- No need to name the results - chain methods
- Example:

```
scala> List.range(1, 10)
     .filter(x => x % 2 == 0)
     .map(x => x * x)
     .foldLeft(0)((x, y) => x + y)
res43: Int = 120
```

References

- “Scala By Example”, Martin Odersky, EPFL
- “Functional Programming Principles in Scala”, Martin Odersky, Coursera