CSC 347 - Concepts of Programming Languages

Functional Programming with Lists

Instructor: Stefan Mitsch

Learning Objectives

❓ How do we support processing collections?

- Understand functional collections processing
- Understand list comprehensions

Exercise: Print Every Element of a List

Express in an imperative style

Java

```
void printList (List<Int> xs) =
  for (int i = 0; i < xs.length; i++)
    System.out.println(xs.get(i));
```

Scala

```
def printList (xs:List[Int]) : Unit =
  for i <- 0 until xs.length do
    println(xs(i))
```

- Syntax appears similar to imperative Java, but uses a *for-comprehension*
- `i until j` is the set of numbers `[i,j)` (exclusive of `j`)

List and Set Comprehensions

Set Comprehensions

$$\{(m,n) | m \in 0,\ldots,10 \land n \in 0,\ldots,10 \land m \leq n\}$$

List Comprehensions

- In many PLs
  - SETL (1960s)
  - Haskell

    ```
    s = [ (m,n) | m <- [0..10], n <- [0..10], m <= n ]
    ```

  - Scala

    ```
    val s = for m <- 0 to 10; n <- 0 to 10; if m <= n yield (m,n)
    ```

  - Python

    ```
    {(m, n) for m in range (0,11) for n in range(0,11) if m <= n}
    ```

  - JavaScript 1.7

Exercise: Print Every Element of a List

Express in a functional style with pattern matching

```
def printList (xs:List[Int]) : Unit =





val xs = List(11,21,31)
printList (xs)
```

Exercise: Print Every Element of a List

❓ Visit all elements of a list

```
def printList (xs:List[Int]) : Unit = xs match
  case Nil   => ()
  case y::ys =>

    printList (ys)


val xs = List(11,21,31)
printList (xs)
```

Exercise: Print Every Element of a List

Print an element when visiting

```scala
def printList (xs:List[Int]) : Unit = xs match
  case Nil   => ()
  case y::ys =>
    println (y)
    printList (ys)


val xs = List(11,21,31)
printList (xs)
```

Exercise: Print Every Element of a List

Format every element before printing

```scala
def printList (xs:List[Int]) : Unit = xs match
  case Nil   => ()
  case y::ys =>
    println ("0x%02x".format(y))
    printList (ys)


val xs = List(11,21,31)
printList (xs)
```

Exercise: Print Every Element of a List

Different type of list

```
def printList (xs:List[List[Int]]) : Unit = xs match
  case Nil   => ()
  case y::ys =>
    println (y.length)
    printList (ys)


val xss = List(List(11,21,31),List(),List(41,51))
printList (xss)
```

List Operation: Foreach

💡 Generalize the idea of processing every element

```
def foreach (xs:List[Int], f:Int=>Unit) : Unit = xs match
  case Nil   => ()
  case y::ys =>
    f (y)
    foreach (ys, f)


val xs = List(11,21,31)
foreach (xs, println)
```

List Operation: Foreach

Customize with changed function argument

```scala
def foreach (xs:List[Int], f:Int=>Unit) : Unit = xs match
  case Nil    => ()
  case y::ys =>
    f (y)
    foreach (ys, f)

def printHex (x:Int) = println("0x%02x".format(x))
val xs = List(11,21,31)
foreach (xs, printHex)
```

List Operation: Foreach

Generalize the type of list with parameters

```scala
def foreach [X] (xs:List[X], f:X=>Unit) : Unit = xs match
  case Nil   => ()
  case y::ys =>
    f (y)
    foreach (ys, f)

def printLength (xs:List[Int]) = println (xs.length)
val xss = List(List(11,21,31),List(),List(41,51))
foreach (xss, printLength)
```

List Operation: Foreach

Use a lambda expression (anonymous function)

```
def foreach [X] (xs:List[X], f:X=>Unit) : Unit = xs match
  case Nil   => ()
  case y::ys =>
    f (y)
    foreach (ys, f)


val xss = List(List(11,21,31),List(),List(41,51))
foreach (xss, (xs:List[Int]) => println (xs.length))
```

List Operation: Foreach

Using the builtin `List` class `foreach` method

- Lambda expression:

```
xs.foreach ((x:Int) => println (x))
```

- Named method/function:

```
def print (x:Int) = println (x)        val print = (x:Int) => println (x)
xs.foreach (print)                     xs.foreach (print)
```

- Types unnecessary if Scala can infer

```
xs.foreach (x => println ("0x%02x".format(x)))
```

Types and Function Parameters

```
def foreach [X] (xs:List[X], f:X=>Unit) : Unit = ...
```

- `X` is a *type parameter*
  - Type parameters in square brackets
  - Value parameters in round brackets
  - Types before values
- `f` is a parameter of *function type:* `(X=>Unit)`
  - takes an argument of type `X`
  - returns a result of type `Unit`

For Comprehensions

- For-comprehensions express collections processing

```
val xs = List(1,2,3)
```

List Method `foreach`

```
xs.foreach (x => println (x))
```

```
xs.foreach (println)
```

For Comprehension

```
for x <- xs do println (x)
```

Anonymous Functions

- Element processing code is often a small snippet
- Using lambda notation

```
val add = (x:Int, y:Int) => x+y
add(11,21)
```

- Use underscore when parameters used exactly once

```
val add = (_:Int) + (_:Int)
add(11,21)
```

- Types may be inferred in some contexts

```
var add : (Int,Int)=>Int = null
add = (x,y) => x+y
add = _ + _
add(11,21)
```

Observing a Recursive Function

Use a compound expression

```
def foreach [X] (xs:List[X], f:X=>Unit) : Unit =
  println("call foreach(%s)".format(xs))
  xs match
    case Nil   => ()
    case y::ys =>
      f (y)
      foreach (ys, f)
  println("rtrn foreach(%s)".format(xs))

val xs = List(21,31)
foreach (xs, (x:Int) => ())
```

```
call foreach(List(21, 31))
call foreach(List(31))
call foreach(List())
rtrn foreach(List())
rtrn foreach(List(31))
rtrn foreach(List(21, 31))
```

Reference and Structural Equality

- Return a reference to a list

```
def reference (xs:List[Int]) : List[Int] = xs



xs eq reference(xs) /* reference equality */
xs == reference(xs) /* value equality */
```

```
res1: Boolean = true
res2: Boolean = true
```

Reference and Structural Equality

- Return a copy of a list

```
def copy (xs:List[Int]) : List[Int] = xs match
  case Nil    => Nil
  case y::ys => y::copy(ys)

xs eq copy(xs) /* reference equality */
xs == copy(xs) /* value equality */
```

```
res1: Boolean = false
res2: Boolean = true
```

List Operation: Map

Return a transformed copy of a list

```scala
def map (xs:List[Int]) : List[String] = xs match
  case Nil    => Nil
  case y::ys => ("0x%02x".format (y)) :: map (ys)

val xs = List(11,21,31)
map(xs)
```

List Operation: Map

Change the type of list

```scala
def map (xs:List[List[Int]]) : List[Int] = xs match
  case Nil    => Nil
  case y::ys => (y.length) :: map (ys)

val xss = List(List(11,21,31),List(),List(41,51))
map(xss)
```

- every cons cell in input results in a cons cell in output

List Operation: Map

implement generic iterate, generalize element function

```
def map (xs:List[Int], f:Int=>String) : List[String] = xs match
  case Nil    => Nil
  case y::ys => f(y) :: map (ys, f)

val xs = List(11,21,31)
map(xs, "0x%02x".format (_))
```

List Operation: Map

abstracting types

```
def map [X,Y] (xs:List[X], f:X=>Y) : List[Y] = xs match
  case Nil   => Nil
  case y::ys => f(y) :: map (ys, f)

val xs = List(11,21,31)
map(xs, (y:Int) => "0x%02x".format (y))
```

List Operation: Map

`copy` is just a specialization of `map` with `f:X=>X = x=>x`

```
def map [X,Y] (xs:List[X], f:X=>Y) : List[Y] = xs match
  case Nil   => Nil
  case y::ys => f(y) :: map (ys, f)

val xs = List(11,21,31)
map(xs, x => x)
```

List Operation: Map

`foreach` is just a specialization of `map` with `f:X=>Unit`

```
def map [X,Y] (xs:List[X], f:X=>Y) : List[Y] = xs match
  case Nil    => Nil
  case y::ys => f(y) :: map (ys, f)

val xs = List(11,21,31)
map(xs, (x:Int) => println ("0x%02x".format(x)))
```

- Builtin `map`

  ```
  xs.map ("0x%02x".format (_))
  ```

Examples

- Map a `List[List[Int]]` to a `List[Int]`

```
List(List(11,21,31),List(),List(41,51)).map (_.length)
```

```
res1: List[Int] = List(3, 0, 2)
```

- Map a `List[String]` to a `List[Int]`

```
List("hi", "it's", "me").map (_.length)
```

```
res1: List[Int] = List(2, 4, 2)
```

- Map a `List[Int]` to a `List[Int]`

```
List(1, 2, 3, 4).map (_ + 1)
```

```
res1: List[Int] = List(2, 3, 4, 5)
```

For Comprehensions

Method `map`

```
xs.map (x => "0x%02x".format(x))
```

Method `foreach`

```
xs.foreach (x => println ("0x%02x".format(x)))
```

For Comprehension `yield`

```
for x <- xs yield "0x%02x".format(x)
```

For Comprehension `do`

```
for x <- xs do println ("0x%02x".format(x))
```

List Operation: Filter

Revisit the copy operation

```
def copy    [X] (xs:List[X])                    : List[X] = xs match
  case Nil              => Nil
  case y::ys            => y :: copy   (ys)


val zs = (0 to 7).toList
copy(zs)
```

List Operation: Filter

Copy only elements that satisfy a predicate `f`

```
def filter [X] (xs:List[X], f:X=>Boolean) : List[X] = xs match
  case Nil                => Nil
  case y::ys if f (y) => y :: filter (ys, f)
  case _::ys              =>      filter (ys, f)

val zs = (0 to 7).toList
filter(zs, ((_:Int) % 3 != 0))
```

For Comprehensions

Method `filter`

```
zs.filter (z => z % 3 != 0) // shorter: zs.filter(_ % 3 != 0)
```

Nested Methods

```
zs.filter (z => z % 3 != 0).map (z => "0x%02x".format(z))
```

For Comprehension

```
for  z <- zs; if z % 3 != 0  yield z
```

Combined For Comprehension

```
for z <- zs; if z % 3 != 0 yield "0x%02x".format(z)
```

List Operation: Flatten

Revisit the copy operation

```
def copy    [X] (xs:List[List[X]]) : List[List[X]] = xs match
  case Nil    => Nil
  case y::ys => y ::  copy    (ys)

val xss = List(List(11,21,31),List(),List(41,51))
copy(xss)
```

```
res1: List(List(11,21,31),List(),List(41,51))
```

List Operation: Flatten

💡 Create a copy with a flat structure: replace `::` with `:::`

```
def flatten [X] (xs:List[List[X]]) : List[X] = xs match
  case Nil   => Nil
  case y::ys => y ::: flatten (ys)

val xss = List(List(11,21,31),List(),List(41,51))
flatten(xss)
```

```
res1: List(11,21,31,41,51)
```

List Operation: FlatMap

Revisit method `map`

```
def map     [X,Y] (xs:List[X], f:X=>List[Y]) : List[List[Y]] = xs match
  case Nil    => Nil
  case y::ys => f(y) ::  map     (ys, f)

val as = List(3,0,2)
map    (as, (x:Int) => (1 to x).toList)
```

```
res1: List(List(1,2,3), List(), List(1,2))
```

List Operation: FlatMap

Create a transformed list with a flat structure: replace `::` with `:::`

```
def flatMap [X,Y] (xs:List[X], f:X=>List[Y]) : List[Y] = xs match
  case Nil   => Nil
  case y::ys => f(y) ::: flatMap (ys, f)

val as = List(3,0,2)
flatMap(as, (x:Int) => (1 to x).toList)
```

```
res1: List(1,2,3,1,2):::Nil
```

List Operation: FlatMap

Argument and return types of `map` and `flatMap`

- `map: (List[X],X=>List[Y]) => List[List[Y]]`
  - Each element of result list is a `List[Y]`
  - Length of result = length of `xs`
- `flatMap: (List[X],X=>List[Y]) => List[Y]`
  - Each element of result list is a `Y`
  - Length of result = sum of lengths of each `List[Y]`

For Comprehensions

Method `flatMap`

```
xss.flatMap (x=>x) // same as xss.flatten
```

For Comprehension

```
for xs <- xss; x <- xs yield x
```

For Comprehensions

- Multiple iterators

```
val xss = List(List(11,21,31),List(),List(41,51))
for xs <- xss; x <- xs yield (x, xs.length)
```

```
res1: List[(Int, Int)] = List((11,3), (21,3), (31,3), (41,2), (51,2))
```

- Cross product of independent iterators

```
val xs = List(11,21,31)
val ys = List("a","b")
for x <- xs; y <- ys yield (x, y)
```

```
res1: List[(Int, String)] = List((11,a), (11,b), (21,a), (21,b), (31,a), (31,b))
```

```
(for x <- (1 to 7);
     y <- (1 to 9) yield (x, y)).length
```

```
res1: Int = 63
```

For Comprehensions: Types

```scala
val xs = List(11,21,31)
val ys = List("a","b")
for x <- xs;
    y <- ys yield (x, y)
```

- `xs : List[Int]`
- `ys : List[String]`
- Scala infers types for iterator variables

```
x : Int
y : String
```

- `yield` provides the type for the result

```
(x, y) : (Int, String)
for ... yield (x, y) : List[(Int, String)]
```

Summary

- Element-wise list processing is a universal, higher-order function
- Element-processing function is passed to `foreach`, `map`, etc. as argument
- Scala for-comprehensions are a loop-style way of expressing list comprehensions