50.054 Introduction to Scala Part 2

ISTD, SUTD

Learning Outcomes

- Model problems and design solutions using Algebraic Datatype and Pattern Matching
- Compile and execute simple Scala programs

Recap

```scala
def sum(l:List[Int]):Int = l match {
        case Nil => 0
        case (hd::tl) => hd + sum(tl)
    }
```

resembles

$$sum(l) = \begin{cases} 0 & l \text{ is empty} \\ head(l) + sum(tail(l)) & \text{otherwise} \end{cases}$$

Recap

Defines a Scala `findMax()` function that implements the following specification

$$findMax(l) = \begin{cases} Int.MinValue & l \text{ is empty} \\ head(l) & head(l) > findMax(tail(l)) \\ findMax(tail(l)) & otherwise \end{cases}$$

Scala Generics

Consider

```scala
def reverse(l:List[Int]):List[Int] = l match {
    case Nil => Nil
    case (hd::tl) => reverse(tl) ++ List(hd)
}
```

this function should work for any element type, not just `Int`

```scala
def reverse[A](l:List[A]):List[A] = l match {
    case Nil => Nil
    case (hd::tl) => reverse(tl) ++ List(hd)
}
```

- `[A]` is a type argument, `A` is a generic type
- In `reverse(List(1,2,3))`, Scala infers that the `A` should be `Int`

Tail Recursion

```scala
def reverse[A](l:List[A]):List[A] = l match {
    case Nil => Nil
    case (hd::tl) => reverse(tl) ++ List(hd)
}
val l = (1 to 10000).toList
reverse(l) // stack overflow!
```

- Scala does not auto-rerwrite non-tail recursion to tail recursion.
    - In fact Haskell favors not to rewrite by default as it is using lazy-evaluation
- A manual rewrite can be done by creating a second argument as an accumulator
- For details on tail recursion rewriting, refer to
    - https://www.sciencedirect.com/science/article/pii/S1567832613000313
    - https://dl.acm.org/doi/abs/10.1145/3571233
    - https://en.wikipedia.org/wiki/Continuation-passing_style

Tail Recursion

```scala
import scala.annotation.tailrec
def reverse[A](l:List[A]):List[A] = {
    @tailrec
    def go(i:List[A], o:List[A]) : List[A] = i match {
        case Nil => o
        case (x::xs) => go(xs, x::o)
    }
    go(l,Nil)
}
```

- adding a `tailrec` annotation enforces the compiler to check whether the function is indeed tail recursive.

Map

- Many languages and frameworks offer pre-define generic traversal for list as map and reduce.
- Scala List comes with builtin support for these operations.

```
def addToEach(x:Int, l:List[Int]):List[Int] = l match {
    case Nil => Nil
    case (y::ys) => {
        val yx = y+x
        yx::addToEach(x,ys)
    }
}
```

can be rewritten using map

```
def addToEach(x:Int, l:List[Int]):List[Int] = l.map(y=>y+x)
```

- When a single argument method is called, Scala allows us to drop the dot.

```
def addToEach(x:Int, l:List[Int]):List[Int] = l map (y=>y+x)
```

Fold

Recall `sum`

```
def sum(l:List[Int]):Int = l match {
        case Nil => 0
        case (hd::tl) => hd + sum(tl)
    }
```

If we rewrite it into tailrec

```
def sum(l:List[Int]):Int = {
    def go(acc:Int, l:List[Int]):Int = l match {
        case Nil => acc
        case (hd::tl) => go(acc+hd, tl)
    }
    go(0,l)
}
```

FoldLeft

```scala
def sum(l:List[Int]):Int = {
    def go(acc:Int, l:List[Int]):Int = l match {
        case Nil => acc
        case (hd::tl) => go(acc+hd, tl)
    }
    go(0,l)
}
```

- We can rewrite it using `foldLeft`

```scala
def sum(l:List[Int]):Int = l.foldLeft(0)((acc,x)=> acc+x)
```

Note that `foldLeft` is implemented with tail recursion.

FoldRight

```
def sum(l:List[Int]):Int = {
    def go(acc:Int, l:List[Int]):Int = l match {
        case Nil => acc
        case (hd::tl) => go(acc+hd, tl)
    }
    go(0,l)
}
```

- Alernatively we can rewrite it using `foldRight`

```
def sum(l:List[Int]):Int = l.foldRight(0)((x,acc)=> x+acc)
```

Note that `foldRight` is implemented without tail recursion.

FoldLeft vs FoldRight

- Do `foldLeft` and `foldRight` always give the same result?

```
val l = List("a","better","world", "by", "design")
l.foldLeft("")((acc,x) => (acc+" "+x))
l.foldRight("")((x,acc) => (x+" "+acc))
```

we get

```
val res0: String = " a better world by design"
val res1: String = "a better world by design "
```

FoldLeft vs FoldRight

- `foldLeft` and `foldRight` are two methods defined in the List class's implementation
- For simplicity, we consider a monomorphic version without worrying about the generics.

```
enum List[String] {
    def foldLeft(acc:String)(agg:(String,String) => String):String = this match {
        case Nil => acc
        case (hd::tl) => {
            val acc_next = agg(acc,hd)
            tl.foldLeft(acc_next)(agg)
        }
    }
    def foldRight(acc:String)(agg:(String,String) => String):String = this match {
        case Nil => acc
        case (hd::tl) =>  agg(hd, tl.foldRight(acc)(agg))
    }
}
```

FoldLeft in slow motion

```
enum List[String] {
    def foldLeft(acc:String)(agg:(String,String) => String):String = this match {
        case Nil => acc
        case (hd::tl) => {
            val acc_next = agg(acc,hd)
            tl.foldLeft(acc_next)(agg)
        }
    }
}
val l = List("a","better","world", "by", "design")
val g = (acc,x) => (acc+" "+x)
l.foldLeft("")(g)
List("a","better","world", "by", "design").foldLeft("")(g) --->
List("better","world", "by", "design").foldLeft(g("","a"))(g) --->
List("better","world", "by", "design").foldLeft(" a")(g) --->
List("world", "by", "design").foldLeft(g(" a","better"))(g) --->
List("world", "by", "design").foldLeft(" a better")(g) --->
List("by", "design").foldLeft(g(" a better","world"))(g) --->
List("by", "design").foldLeft(" a better world")(g) --->
List("design").foldLeft(g(" a better world","by"))(g) --->
List("design").foldLeft(" a better world by")(g) --->
Nil.foldLeft(g(" a better world by","design"))(g) --->
Nil.foldLeft(" a better world by design")(g) --->
" a better world by design"
```

FoldLeft in slow motion (w/ laziness)

```
enum List[String] {
    def foldLeft(acc: => String)(agg:(String,String) => String):String = this match {
        case Nil => acc
        case (hd::tl) => {
            val acc_next = agg(acc,hd)
            tl.foldLeft(acc_next)(agg)
        }
    }
}

val l = List("a","better","world", "by", "design")
val g = (acc,x) => (acc+" "+x)
l.foldLeft("")(g)

List("a","better","world", "by", "design").foldLeft("")(g) --->
List("better","world", "by", "design").foldLeft(g("","a"))(g) --->
List("world", "by", "design").foldLeft(g(g("","a"),"better"))(g) --->
List("by", "design").foldLeft(g(g(g("","a"),"better"),"world"))(g) --->
List("design").foldLeft(g(g(g(g("","a"),"better"),"world"),"by"))(g) --->
Nil.foldLeft(g(g(g(g(g("","a"),"better"),"world"),"by"),"design"))(g) --->
g(g(g(g(g("","a"),"better"),"world"),"by"),"design") --->
" a better world by design"
```

FoldRight

```
enum List[String] {
    def foldRight(acc:String)(agg:(String, String) => String):String = this match {
        case Nil => acc
        case (hd::tl) =>  agg(hd, tl.foldRight(acc)(agg))
    }
}

val l = List("a","better","world", "by", "design")
val g = (acc,x) => (x+" "+acc)
l.foldRight("")(g)

List("a","better","world", "by", "design").foldRight("")(g) --->
g("a",List("better","world", "by", "design").foldRight("")(g)) --->
g("a",g("better", List("world", "by", "design").foldRight("")(g))) --->
g("a",g("better",g("world",List("by", "design").foldRight("")(g)))) --->
g("a",g("better",g("world",g("by", List("design").foldRight("")(g))))) --->
g("a",g("better",g("world",g("by", g("design", Nil.foldRight("")(g)))))) --->
g("a",g("better",g("world",g("by", g("design", ""))))) --->
"a better world by design "
```

When to use foldRight? FoldRight works well with infinite data if the agg() function's 2nd operand is lazy.

https://voidmainargs.blogspot.com/2011/08/folding-stream-with-scala.html

Filter

`filter` returns a new list with elements satisfying the boolean test functon.

```
val l = List(1,2,3,4)
def even(x:Int):Boolean = x%2==0
l.filter(even)
val res0: List[Int] = List(2, 4)
```

QuickSort with filter

```
def qsort(l:List[Int]):List[Int] = l match {
    case Nil => Nil
    case List(x) => List(x)
    case (p::rest) => {
        val ltp = rest.filter( x => x < p)
        val gep = rest.filter( x => !(x < p))
        qsort(ltp) ++ List(p) ++ qsort(gep)
    }
}
```

which resembles the math specification

$$qsort(l) = \begin{cases} l & |l| < 2 \\ qsort(\{x|x \in l \wedge x < head(l)\}) \uplus \{head(l)\} \uplus qsort(\{x|x \in l \wedge \neg(x < head(l))\}) & otherwise \end{cases}$$

where $\uplus$ unions two bags and maintains the order.

FlatMap

```
val l = (1 to 5).toList
l.map( i => if (i%2 ==0) { List(i) } else { Nil })
```

would yield

```
List(List(), List(2), List(), List(4), List())
```

What if we want to join the inner lists together?

```
l.flatMap( i => if (i%2 ==0) { List(i) } else { Nil })
```

would yield

```
List(2,4)
```

FlatMap and Map

With `flatMap` and `map`, we can define complex (multi-dimension) list transformation

```
def listProd[A,B](la:List[A], lb:List[B]):List[(A,B)] =
    la.flatMap( a => lb.map(b => (a,b)))

val l = (1 to 5).toList
val l2 = List('a', 'b', 'c')
listProd(l, l2)
```

which produces

```
List((1,a), (1,b), (1,c), (2,a), (2,b), (2,c), (3,a), (3,b), (3,c),
    (4,a), (4,b), (4,c), (5,a), (5,b), (5,c))
```

For comprehension

In Python, we note that

```
[ (x+1) for x in [1,2,3]]
```

is the same as

```
map(lambda x:x+1, [1,2,3])
```

In Scala, we have

```
for { x <- List(1,2,3) } yield (x+1)
```

is the same as

```
List(1,2,3).map(x=>x+1)
```

For comprehension

It turns out that

```
def listProd[A,B](la:List[A], lb:List[B]):List[(A,B)] =
    la.flatMap( a => lb.map(b => (a,b)))
```

can be rewritten as

```
def listProd[A,B](la:List[A], lb:List[B]):List[(A,B)] =
    for {
        a <- la
        b <- lb
    } yield (a,b)
```

In general,

```
for { x1 <- e1;  x2 <- e2; ...; xn <- en } yield e
```

is equivalent to

```
e1.flatMap( x1 => e2.flatMap(x2 =>  ... en.map( xn => e) ...))
```

Algebraic Data Type

Suppose we would like to implement a math expression library

$$
\begin{array}{rrcl}
(\texttt{MathExp}) & e & ::= & e + e \mid e - e \mid e * e \mid e/e \mid c \\
(\texttt{Constant}) & c & ::= & \ldots \mid -1 \mid 0 \mid 1 \mid \ldots
\end{array}
$$

We can model it using algebraic data type in Scala 3

```
enum MathExp {
    case Plus(e1:MathExp, e2:MathExp)
    case Minus(e1:MathExp, e2:MathExp)
    case Mult(e1:MathExp, e2:MathExp)
    case Div(e1:MathExp, e2:MathExp)
    case Const(v:Int)
}
```

- `MathExp` is the type, which have 5 alternatives.
- `Plus`, `Minus`, `Mult`, `Div` and `Const` are the value (pattern) constructors

Algebraic Data Type

```
enum MathExp {
    case Plus(e1:MathExp, e2:MathExp)
    case Minus(e1:MathExp, e2:MathExp)
    case Mult(e1:MathExp, e2:MathExp)
    case Div(e1:MathExp, e2:MathExp)
    case Const(v:Int)
}
```

We can represent the math expression `(1+2) * 3` as

```
import MathExp.*
val my_exp = Mult(Plus(Const(1), Const(2)), Const(3))
```

Algebraic Data Type

Given the specification of $eval()$

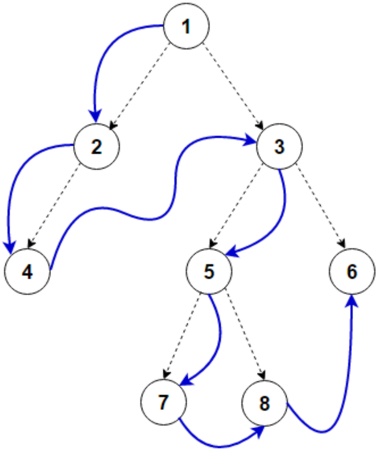
$$eval(e) = \begin{cases} eval(e_1) + eval(e_2) & if \;\; e = e_1 + e_2 \\ eval(e_1) - eval(e_2) & if \;\; e = e_1 - e_2 \\ eval(e_1) * eval(e_2) & if \;\; e = e_1 * e_2 \\ eval(e_1)/eval(e_2) & if \;\; e = e_1/e_2 \\ c & if \;\; e = c \end{cases}$$

We define

```
def eval(e:MathExp):Int = e match {
    case Plus(e1, e2)  => eval(e1) + eval(e2)
    case Minus(e1, e2) => eval(e1) - eval(e2)
    case Mult(e1, e2)  => eval(e1) * eval(e2)
    case Div(e1, e2)   => eval(e1) / eval(e2)
    case Const(i)      => i
}
eval(my_exp) // yield 9
```

Tree Algorithms

Recall that binary tree is a tree consisting of nodes with maximum 2 children per node. A Preorder traversal is described in the diagram below.



Preorder: 1, 2, 4, 3, 5, 7, 8, 6

Tree Algorithms

We can model a binary tree using an enum type

```
enum BT {
    case Empty
    case Node(v:Int, left:BT, right:BT)
}
import BT.*
val my_tree = Node(1, Node(2, Node(4, Empty, Empty), Empty),
                      Node(3, Node(5, Node(7, Empty, Empty),
                                      Node(8, Empty, Empty)),
                              Node(6, Empty, Empty)))

def preorder(t:BT):List[Int] = ???
```

Can you implement this algorthm?

Summary

In this lesson, we have discussed

- How to use List datatype to model and manipulate collections of multiple values.
- How to use Algebraic data type to define user customized data type to solve complex problems.