RECURSION AND DATA STRUCTURES

Benedict R. Gaster / @cuberoo_

UWE BRISTOL
University of the West of England

bettertogether

INTRODUCTION

As we have already seen many functions can be expressed in terms of loops

```
def sum ( is : List[Int] ) : Int = {
  var total = 0
  for (i <- is) {
    total = total + i
  }
  total
}
```

Sums the elements of a list: e.g. sum(List(1,2,3,4)) = 10

PRODUCT

```
def product ( is : List[Int] ) : Int = {
  var total = 1
  for (i <- is) {
    total = total * i
  }
  total
}
```

The product of a list: e.g. product(List(1,2,3,4)) = 24

RECURSION

In many programming languages functions can be defined in terms of themselves. Such functions are called recursive.

```
def sum ( is : List[Int] ) : Int =
  if (is.length == 0)
    0
  else
    is.head + sum(is.tail)
```

- sum maps the empty list to 0 (the unit of +).
- and the head of a non-empty list is added to the remainder of the list.

PATTERN MATCHING

Like Java, Scala allows "matching" (switch in Java) against integers with **match** expressions

```
def toYesOrNo(choice: Int): String = choice match {
  case 1 => "yes"
  case 0 => "no"
  case _ => "error"
}
```

If input is 1 return "yes", else if input is 0 return "no", otherwise return "error"

PATTERN MATCHING ON LISTS

Unlike Java, Scala allows pattern matching on (case) classes:

```
def isEmpty(ls: List[Int]): Boolean = ls match {
  case Nil    => true
  case x : xs => false
}
```

If input is Nil (empty list) return true, otherwise return false.

DEFINING RECURSIVE FUNCTIONS WITH PATTERN MATCHING

Combinding pattern matching and recursion is very powerful:

```
def sum ( is : List[Int] ) : Int = is match {
    case Nil             => 0
    case i :: remainder => i + sum(remainder)
  }
```

- sum pattern matches the empty list (Nil), returning 0.
- sum pattern matches against the non-empty list (i :: remainder), returning i plus the sum of remainder.

WE CAN DEFINE OTHER FUNCTIONS

```
def product ( is : List[Int] ) : Int = is match {
  case Nil             => 1
  case i :: remainder => i * product(remainder)
}
```

- product pattern matches the empty list (Nil), returning 1.
- product pattern matches against the non-empty list (i :: remainder), returning i times the product of remainder.

RELATIONSHIP BETWEEN SUM AND PRODUCT

Do you notice anything similar about sum and product?

RELATIONSHIP BETWEEN SUM AND PRODUCT

- sum pattern matches the empty list (Nil), returning 0.
- sum pattern matches against the non-empty list (i :: remainder), returning i plus the sum of remainder.

- product pattern matches the empty list (Nil), returning 1.
- product pattern matches against the non-empty list (>i :: remainder), returning i times the product of remainder.

GENERAL RECURSIVE PATTERN OVER LISTS

- Define a function R(f, unit, ls), where:
  - f is a function of type (Int,Int) => Int
  - unit is of type Int
  - ls is of type List[Int]
- Base case:
  - pattern matches Nil, then return a unit value.
- Recursive case:
  - pattern matches i :: remainder, then apply function f to i and the result of a recursive call to R.

FOLD

R is often called **fold** for lists of integers.

A DEFINITION OF FOLD FOR LISTS (OF INTS)

```
def fold ( f : (Int, Int) => Int, unit : Int, is : List[Int] ) : Int =
  is match {
    case Nil      => unit
     case i :: is => f(i, fold(f, unit, is))
   }
```

IMPLEMENT SUM AND PRODUCT WITH FOLD

```
val sum    = (xs : List[Int]) => fold((x,y) => x + y,  0, xs)
val prodct = (xs : List[Int]) => fold((x,y) => x * y,  1, xs)
```

WHY IS RECURSION USEFUL?

- Some functions, such as product, are simpler to define in terms of other functions.
- As we have already seen (e.g. fold) and will see more of, many functions are easier to define recursively interms of themselves.
- Many data-structures are defined recursively in terms of themselves, e.g. lists.

FIBONACCI NUMBERS

- In mathematics, the Fibonacci number are the numbers in the following integer sequence
  - $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$
- It can be defined by the following (recursive) mathematical definition:

$$F_n = F_{n-1} + F_{n-2}$$

with seed values

$$F_1 = 1 \; and \; F_2 = 1$$

FIBONACCI IN SCALA

```
def fib( n : Int) : Int = n match {
  case 1 | 2 => n
  case _ => fib( n-1 ) + fib( n-2 )
 }
```

FIBONACCI IN SCALA

Our Scala definition is very similar to the mathematical one!

```
def fib( n : Int) : Int = n match {
  case 1 | 2 => n
  case _ => fib( n-1 ) + fib( n-2 )
 }
```

$$F_n = F_{n-1} + F_{n-2}$$

with seed values

$$F_1 = 1 \; and \; F_2 = 1$$

DATA STRUCTURES

- A particular way of organizing data within a computer program
- We've already seen many examples there are many more:
  - Lists
  - Array
  - Set
  - Queues
  - Hash Tables
  - Trees

OBJECTS

All objects are data-structures

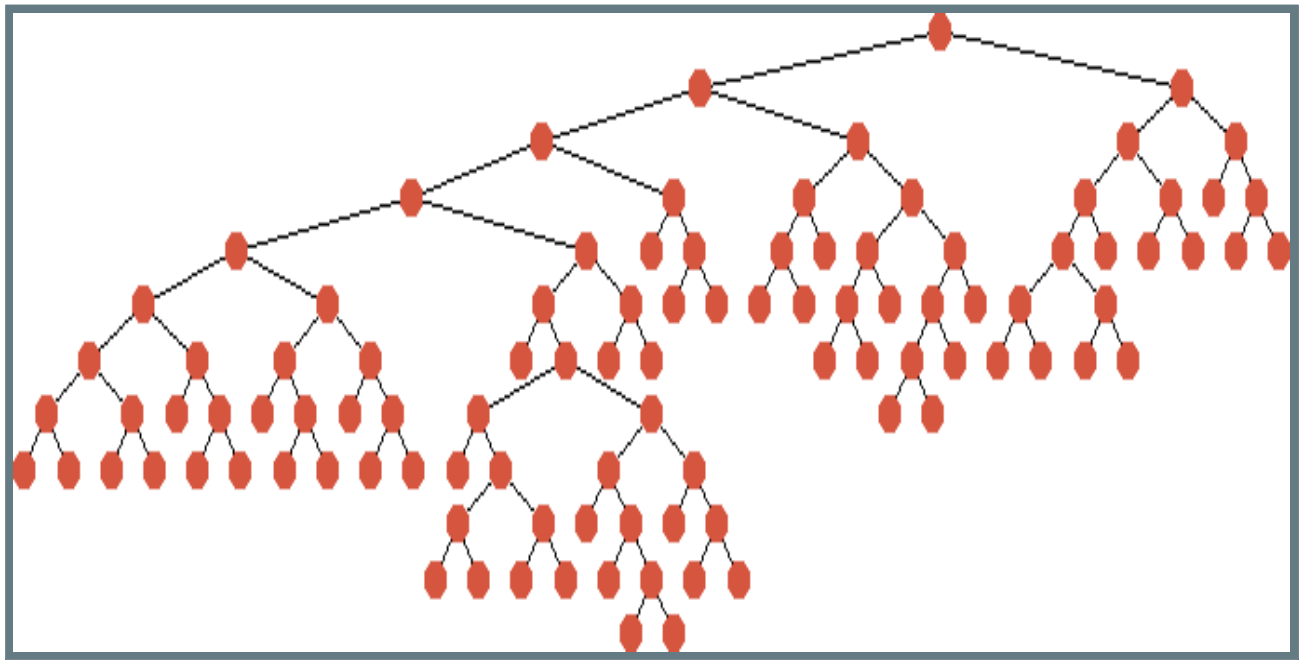
DATA STRUCTURES ARE OFTEN RECURSIVE

Many data-structures are naturally expressed recursively

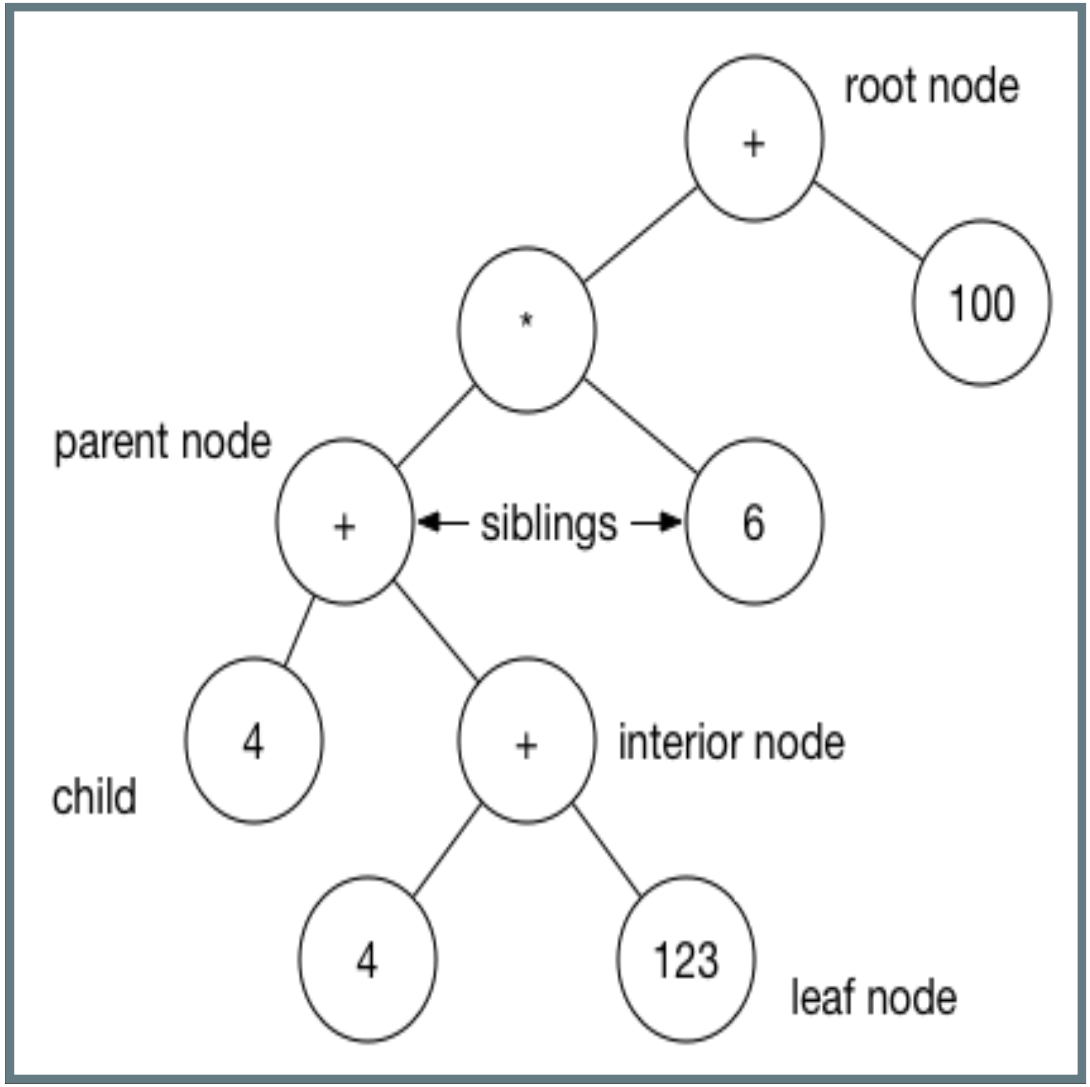
We will look at one such data-structure: Trees.

TREES

A widely used data-structure that simulates a hierarchical tree structure, with a root value and subtrees of children, represented as a set of linked nodes.



SOME TERMINOLOGY



WHY USE TREES

- Reflect structural relationships in the stored data.
- Natural approach to representing hierarchies.
- Provide efficient insertion and searching.

BINARY TREES

- A special and very common type of tree where each (parent) node has at most two childen.
- The children are refered to as the left and right child, respectively.

TRAVERSING BINARY TREES

- Depth-first
  - Decend all the way down one path of the tree, then down the next and so on.
- Breath-first
  - Process each level of the tree before decending to the next level.

TYPES OF TRAVERSAL

- There are three approaches to implementing depth-first traversal:
  - In-order; visit the left child, then the parent, and finally the right child.
  - Pre-order; visit the parent, then the left child, and finally the right child.
  - Post-order; visit the left child, then the right child, and finally the parent.
  -
- There is only a single approach to implementing breath-first traversal.

TREE BASE CLASS (SCALA TRAIT)

```
trait Tree {
   def toListInOrder    : List[Int]
   def toListPreOrder   : List[Int]
   def toListPostOrder  : List[Int]
   def toListLevelOrder : List[Int]
 }
```

TREE BASE CLASS - IN ORDER

```
trait Tree {
  def toListInOrder : List[Int] = this match {
    case Empty()     => Nil
    case Leaf(v)     => v :: Nil
    case Node(l,v,r) => l.toListInOrder ::: (v :: r.toListInOrder)
  }
  ...
}
```

TREE SUB CLASSES

```
case class Leaf(v : Int) extends Tree
case class Node(l : Tree, v : Int, r : Tree) extends Tree
case class Empty() extends Tree
```

A **case class** supports pattern matching

TREE BASE CLASS - PRE ORDER

```
trait Tree {
  def toListPreOrder : List[Int] = this match {
      case Empty()     => Nil
      case Leaf(v)     => v :: Nil
      case Node(l,v,r) => v :: l.toListPreOrder ::: r.toListPreOrder
   }
   ...
 }
```

TREE BASE CLASS - POST ORDER

```
trait Tree {
   // Left as an exercise for the reader
   ...
 }
```

BINARY SEARCH TREES

- Is a useful special case of binary trees.
- A binary Search Tree is a binary tree data structure where each node has a comparable value (key) and satisfies the restriction:
  - that the key in any node is larger than the keys in all nodes in that node's left sub-tree and smaller than the keys in all nodes in that node's right sub-tree.

BINARY SEARCH TREES - INSERTION

```
def insert( i : Int ) : Tree = this match {
   case Empty() => Leaf(i)
   case Leaf(v) => if (i < v)
                     Node(Leaf(i), v, Empty())
                   else if(i > v)
                     Node(Empty(), v, Leaf(i))
                   else
                     this
   case Node(l,v,r) => if (i < v)
                         // insert in left
                         Node(l.insert(i), v, r)
                       else if (i > v)
                         // insert in right
                         Node(l, v, r.insert(i))
                       else
                         // already in tree
                         this
}
```

BINARY SEARCH TREES - EXAMPLE

```
scala> val t = List(2,6,100,84,23,1,2).
                  foldRight(Empty() : Tree) ( (x, t) => t.insert(x) )
t: Tree = Node(Leaf(1),2,Node(Leaf(6),23,Node(Empty(),84,Leaf(100))))

scala> t.toListInOrder
res2: List[Int] = List(1, 2, 6, 23, 84, 100)
```

BINARY SEARCH TREES

- A tree built with **insert** is in sorted order when traversed **in order**.
- We can use this fact to implement a function to sort the elements of a list.

BINARY SEARCH TREES - LIST SORT

```
def sort ( ls : List[Int] ) : List[Int] =
  ls.foldRight(Empty() : Tree) ((x, t) => t.insert(x)).toListInOrder
```

CONCLUSION

- Pattern matching.
- Many functions can be defined in terms of themselves.
- Data-strucures are often defined by recursion.
- Binary trees and binary search trees are useful examples of recursive data-structures.