計算機科学概論演習

#04

青谷知幸

Tokyo Tech., Dept. of Math. & Comp. Sci.

April 27, 2016

目次

課題 3: 単語の出現回数

(file:count.scala)

lst をシンボルのリストとする．lst 中に現れるそれぞれのシンボルについて，それが何回現れたかを数え上げるプログラム counts を開発せよ．counts の入力の型は List[Symbol]，出力の型は List[SymbolCount] とし，SymbolCount は以下で定義されるものとする．

```
case class SymbolCount(sym:Symbol, num:Int)
```

例

- counts(List('a,'cat))＝
  List(SymbolCount('a,1),SymbolCount('cat,1))
- counts(List('slowly,'and,'slowly))＝
  List(SymbolCount('slowly,2),SymbolCount('and,1))

SymbolCount の並び順は気にしなくて良い

解き方: 方針

- `lst` の先頭から，シンボルの初出時に出現回数を数える
  $\Rightarrow$ 既出のシンボルを覚えておく
- `lst` の末尾から `SymbolCount` のリストを生成して (`clst` と呼ぶことにする)，`clst` の適切な `SymbolCount` について第 2 要素に 1 加算する $\Rightarrow$ 適切な `SymbolCount` を探す (`sort` と同様の手法)
- ...

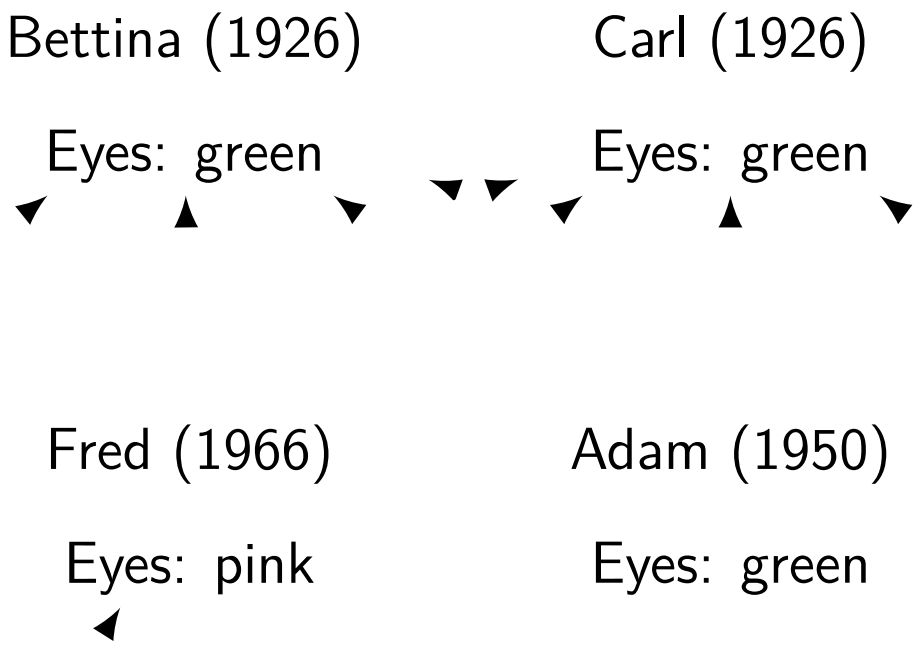
回答例

```
def counts: (List[Symbol]) => List[SymbolCount]
= (lst) =>
/**
 * 目的: lstに現れる其々のシンボルの出現回数を数え上げる
 * 例: counts(List('a,'cat)) =
 *       List(SymbolCount('cat,1), SymbolCount('a,1))
 *     counts(List('slowly,'and,'slowly)) =
 *       List(SymbolCount('slowly,2), SymbolCount('and,1))
 */
{ lst match{
    case Nil => Nil
    case s::rest => inc(s,counts(rest))
  }}
```

`inc(s,lst)` は `lst` に記録されたシンボル `s` の出現回数を 1 増やす補助函数

目次

家系図 (family tree)



Scala での記述: 最初の試み

```
case class FamilyTreeNode(
  father:FamilyTreeNode,
  mother:FamilyTreeNode,
  name:Symbol,
  date:Int,
  eyes:Symbol)
```

Scala での記述: 最初の試み

```
case class FamilyTreeNode(
  father:FamilyTreeNode,
  mother:FamilyTreeNode,
  name:Symbol,
  date:Int,
  eyes:Symbol)
```

Q. これで良い？

Scala での記述: 最初の試み

```
case class FamilyTreeNode(
  father:FamilyTreeNode,
  mother:FamilyTreeNode,
  name:Symbol,
  date:Int,
  eyes:Symbol)
```

Q. これで良い？
A. ダメ

Scala での記述: 最初の試み

```
case class FamilyTreeNode(
  father:FamilyTreeNode,
  mother:FamilyTreeNode,
  name:Symbol,
  date:Int,
  eyes:Symbol)
```

Q. これで良い？
A. ダメ:FamilyTreeNode 型の値が作れない

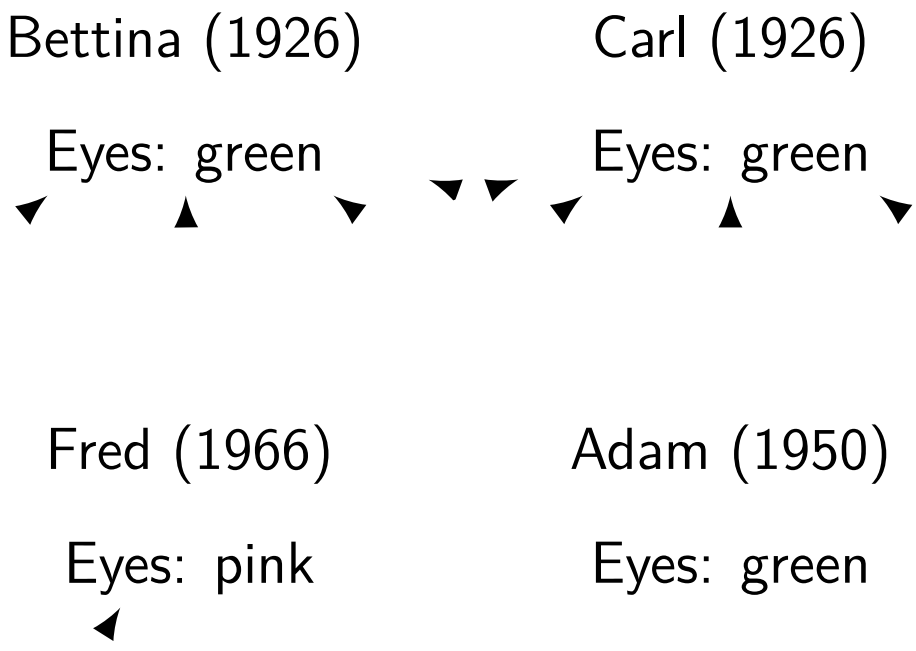
Scala での記述: 最初の試み

```
case class FamilyTreeNode(
  father:FamilyTreeNode,
  mother:FamilyTreeNode,
  name:Symbol,
  date:Int,
  eyes:Symbol)
```

Q. これで良い？
A. ダメ:FamilyTreeNode 型の値が作れない

```
FamilyTreeNode(
  FamilyTreeNode(FamilyTreeNode(...),FamilyTreeNode(...),...),
  FamilyTreeNode(FamilyTreeNode(...),FamilyTreeNode(...),...),
...)
```

家系図 (family tree)



Scala での記述: 完成

親のない内点がないと始祖 (Carl など) が作れないので，空の内点を用意してこれを表せるようにする．

```
abstract class FamilyTreeNode
case class Empty() extends FamilyTreeNode //空の内点
case class Child(
  father:FamilyTreeNode,
  mother:FamilyTreeNode,
  name:Symbol,
  date:Int,
  eyes:Symbol) extends FamilyTreeNode
```

始祖を表す内点は father と mother に Empty() を渡してつくる．

```
FamilyTreeNode(Empty(),Empty(),'Carl,1926,'green)
```

変数を効果的に使う

家系図のように複雑なデータを作る場合はそれぞれのノードに対応する変数を用意するとよい.

```
def Carl = Child(Empty(),Empty(),'Carl,1926,'green)
def Bettina = Child(Empty(),Empty(),'Bettina, 1926, 'green)
def Adam = Child(Carl,Bettina,'Adam,1950,'yellow)
def Dave = Child(Carl,Bettina,'Dave,1955,'black)
def Eva = Child(Carl,Bettina,'Eva,1965,'blue)
def Fred = Child(Empty(),Empty(),'Fred,1966,'pink)
def Gustav = Child(Fred,Eva,'Gustav,1988,'brown)
```

変数を効果的に使う

家系図のように複雑なデータを作る場合はそれぞれのノードに対応する変数を用意するとよい.

```
def Carl = Child(Empty(),Empty(),'Carl,1926,'green)
def Bettina = Child(Empty(),Empty(),'Bettina, 1926, 'green)
def Adam = Child(Carl,Bettina,'Adam,1950,'yellow)
def Dave = Child(Carl,Bettina,'Dave,1955,'black)
def Eva = Child(Carl,Bettina,'Eva,1965,'blue)
def Fred = Child(Empty(),Empty(),'Fred,1966,'pink)
def Gustav = Child(Fred,Eva,'Gustav,1988,'brown)
```

Q. 変数を使わないとどうなる？

変数を効果的に使う

家系図のように複雑なデータを作る場合はそれぞれのノードに対応する変数を用意するとよい.

```
def Carl = Child(Empty(),Empty(),'Carl,1926,'green)
def Bettina = Child(Empty(),Empty(),'Bettina, 1926, 'green)
def Adam = Child(Carl,Bettina,'Adam,1950,'yellow)
def Dave = Child(Carl,Bettina,'Dave,1955,'black)
def Eva = Child(Carl,Bettina,'Eva,1965,'blue)
def Fred = Child(Empty(),Empty(),'Fred,1966,'pink)
def Gustav = Child(Fred,Eva,'Gustav,1988,'brown)
```

Q. 変数を使わないとどうなる？
A. `Carl`, `Bettina` を何度も書く羽目に

家系図を処理する函数

家系図を処理する函数のテンプレートは以下のようになる

```
def funOnFTN: (FamilyTreeNode) => ...
= (tree) => {
  tree match{
    case Empty() => ...
    case Child(f,m,n,d,e) => {
      ...funOnFTN(f)...
      ...funOnFTN(m)...
    }
  }}
```

再帰的な構造が二箇所 (f と m) にあることに注意

例: hasBlueEyedAncestor

問題
瞳の色が青 (`'blue`) の祖先がいれば真，そうでなければ偽を返す函数 `hasBlueEyedAncestor` を開発せよ．

例: hasBlueEyedAncestor

問題
瞳の色が青 ('blue) の祖先がいれば真，そうでなければ偽を返す函数 hasBlueEyedAncestor を開発せよ．

```
def hasBlueEyedAncestor: (FamilyTreeNode) => Boolean =
(tree) =>
/*
 * 目的: treeにeyesが'blueであるノードがあれば真，なければ偽
 * 例: hasBlueEyedAncestor(Carl) == false
 *     hasBlueEyedAncestor(Eva) == true
 *     hasBlueEyedAncestor(Gustav) == true
 */
{...}
```

例: hasBlueEyedAncestor

```
def hasBlueEyedAncestor: (FamilyTreeNode) => Boolean =
(tree) =>
/*
 * 目的: treeにeyesが'blueであるノードがあれば真，なければ偽
 * 例: hasBlueEyedAncestor(Carl) == false
 *     hasBlueEyedAncestor(Eva) == true
 *     hasBlueEyedAncestor(Gustav) == true
 */
{ tree match{
  case Empty() => false
  case Child(f,m,n,d,e) =>
    e == 'blue || hasBlueEyedAncestor(f) ||
    hasBlueEyedAncestor(m)
}}
```

2 分木

木を表す一般的なデータ構造としては，2 分木が有名である．
2 分木は以下のように定義される:[1]

```
abstract class BinaryTree[+A]
case object Leaf extends BinaryTree[Nothing]
case class Branch[+A](
  data:A,
  left:BinaryTree[A],
  right:BinaryTree[A]) extends BinaryTree[A]
```

[1] `case object` は `case class` と殆ど同じものだが，引数がなく，値が `Leaf` ただ 1 つしかない．リストにおける `Nil` と同じである．他方，`case class Leaf() extends ...` は 0 個の引数が存在し，`Leaf()` とするたびに新しい値が作られる．

2 分木: 例

```
scala> Branch('h, Leaf, Branch('d,Leaf,Leaf))
res14: Branch[Symbol] = Branch('h,Leaf,Branch('d,Leaf,Leaf))

scala> Branch('h, Branch('i,Leaf,Leaf), Leaf)
res15: Branch[Symbol] = Branch('h,Branch('i,Leaf,Leaf),Leaf)
```

'h や'i, 'd の型が Symbol なので型変数 A が Symbol に置き換えられた.

家系図を 2 分木で表す

名前と誕生年と瞳の色からなるデータ構造を定義して，その値を要素とする 2 分木を作れば家系図が表現できる．

```
case class Person(name:Symbol, date:Int, eyes:Symbol)

def Carl = Branch(Person('Carl,1926,'green), Leaf,Leaf)
def Bettina = Branch(Person('Bettina, 1926, 'green),Leaf,Leaf)
def Adam = Branch(Person('Adam,1950,'yellow),Carl,Bettina)
def Dave = Branch(Person('Dave,1955,'black),Carl,Bettina)
def Eva = Branch(Person('Eva,1965,'blue),Carl,Bettina)
def Fred = Branch(Person('Fred,1966,'pink),Leaf,Leaf)
def Gustav = Branch(Person('Gustav,1988,'brown),Fred,Eva)
```

Carl, Bettina, Adam, Dave, Eva, Fred, Gustav の型はいずれも Branch[Person] である．

2 分木を処理する函数

2 分木を処理する函数のテンプレートは FamilyTreeNode を処理する函数と殆ど同じ

```
def funOnBT: (BinaryTree[...]) => ...
= (tree) => {
  tree match{
    case Leaf => ...
    case Branch(d,l,r) => {
      ...funOnBT(l)...
      ...funOnBT(r)...
    }
  }}
```

例: hasBlueEyedAncestor

問題
瞳の色が青 (`'blue`) の祖先がいれば真，そうでなければ偽を返す函数 `hasBlueEyedAncestor` を開発せよ．

例: hasBlueEyedAncestor

問題
瞳の色が青 ('blue) の祖先がいれば真，そうでなければ偽を返す函数 hasBlueEyedAncestor を開発せよ．

```
def hasBlueEyedAncestor: (BinaryTree[Person]) => Boolean =
(tree) =>
/**
 * 目的: treeにeyesが'blueであるノードがあれば真，なければ偽
 * 例: hasBlueEyedAncestor(Carl) == false
 *     hasBlueEyedAncestor(Eva) == true
 *     hasBlueEyedAncestor(Gustav) == true
 */
{...}
```

例: hasBlueEyedAncestor

```
def hasBlueEyedAncestor: (BinaryTree[Person]) => Boolean =
(tree) =>
/**
 * 目的: treeにeyesが'blueであるノードがあれば真，なければ偽
 * 例: hasBlueEyedAncestor(Carl) == false
 *     hasBlueEyedAncestor(Eva) == true
 *     hasBlueEyedAncestor(Gustav) == true
 * テンプレート: tree match{
 *   case Leaf => ...
 *   case Branch(Person(n,d,e),l,r) =>
 *          ...hasBlueEyedAncestor(l)..
 *          ...hasBlueEyedAncestor(r).. }
 */
{...}
```

例: hasBlueEyedAncestor

```
def hasBlueEyedAncestor: (BinaryTree[Person]) => Boolean =
(tree) =>
/*
 * 目的: treeにeyesが'blueであるノードがあれば真，なければ偽
 * 例: ...
 * テンプレート: tree match{
 *   case Leaf => ...
 *   case Branch(Person(n,d,e),l,r) =>
 *         ...hasBlueEyedAncestor(l)..
 *         ...hasBlueEyedAncestor(r).. }
 */
{ tree match{
  case Leaf => false
  case Branch(Person(n,d,e),l,r) =>
    e == 'blue || hasBlueEyedAncestor(l) ||
    hasBlueEyedAncestor(r)
}}
```

逆向き家系図

Gustav (1988)

Eyes: brown

Eva (1965)

Eyes: blue

Fred (1966)

Eyes: pink

Adam (1950)

Eyes: green

Dave (1955)

Eyes: black

Bettina (1926)

Eyes: green

Carl (1926)

Eyes: green

逆向き家系図のデータ構造

1 つの内点から伸びる矢印の本数は不定 $\Rightarrow$ リストを使う

逆向き家系図のデータ構造

1 つの内点から伸びる矢印の本数は不定 $\Rightarrow$ リストを使う

```
abstract class ParentTree
case object Empty extends ParentTree
case class MakeParent(
  children:List[ParentTree],
  name:Symbol,
  date:Int,
  eyes:Symbol
  ) extends ParentTree
```

ParentTree の函数の雛形

ParentTree を扱う函数の雛形 (テンプレート) は少し複雑になる.

```
def funOnParent: (ParentTree) => ... =
(tree) => {
  tree match{
    case Empty => ...
    case MakeParent(c,n,d,e) => ...
  }}
```

c は List[ParentTree] であることを思い出すと，リストの各要素について funOnParent を適用したくなる．だが，このままではリストなので適用できない.

ParentTree の函数の雛形

ParentTree を扱う函数の雛形 (テンプレート) は少し複雑になる.

```
def funOnParent: (ParentTree) => ... =
(tree) => {
  tree match{
    case Empty => ...
    case MakeParent(c,n,d,e) => ...
  }}
```

c は List[ParentTree] であることを思い出すと，リストの各要素について funOnParent を適用したくなる．だが，このままではリストなので適用できない.

解決策: List[ParentTree] を扱う函数を作りこの中で funOnParent を呼び出す

ParentTree の函数の雛形

```
def funOnParent: (ParentTree) => ... =
(tree) => {
  tree match{
    case Empty => ...
    case MakeParent(c,n,d,e) => ...funOnParentList(c)...
  }}

def funOnParentList: (List[ParentTree]) => ... =
(lst) => {
  lst match{
    case Nil => ...
    case e::rest =>
      ...funOnParent(e)...
      ...funOnParentList(lst)...
  }}
```

例: 青い瞳の子孫を探す

問題
瞳の色が青 (`'blue`) の子孫がいれば真，そうでなければ偽を返す函数 `hasBlueEyedDescendant` を開発せよ．

例: 青い瞳の子孫を探す

```
def hasBlueEyedDescendant: (ParentTree) => Boolean =
(tree) =>
/**
 * 目的: treeの中にeyesが'blueの内点があれば真，さもなくば偽
 * 例: ...
 * 雛形: tree match{ case Empty =>
 *                   case MakeParent(c,n,d,e) =>
 *                       ...hasBlueEyedChild(c)... }
 */
{ tree match{
    case Empty => false
    case MakeParent(children,n,d,e) =>
      e == 'blue || hasBlueEyedChild(children)
  }}
```

例: 青い瞳の子孫を探す

```
def hasBlueEyedChild: (List[ParentTree]) => Boolean =
(lst) =>
/**
 * 目的: lstの中にeyesが'blueの内点があれば真，さもなくば偽
 * 例: ...
 * 雛形: lst match{ case Nil => ...
 *                  case e::rest =>
 *                     ...hasBlueEyedDescendant(e)...
 *                     ...hasBlueEyedChild(rest)...   }
 */
{ lst match{
    case Nil => false
    case e::rest =>
      hasBlueEyedDescendant(e) || hasBlueEyedChild(rest)
  }}
```

目次

03 課題 3 のヒント

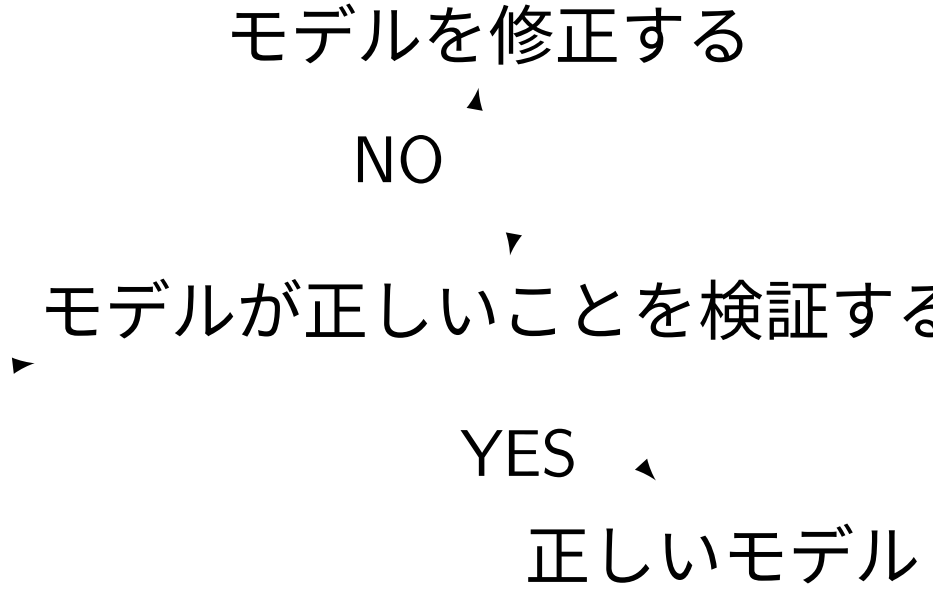
複雑な再帰的構造
- 家系図木
- 2 分木と 2 分探索木

反復的な改良によるプログラムの開発

2 つの複雑なデータの処理

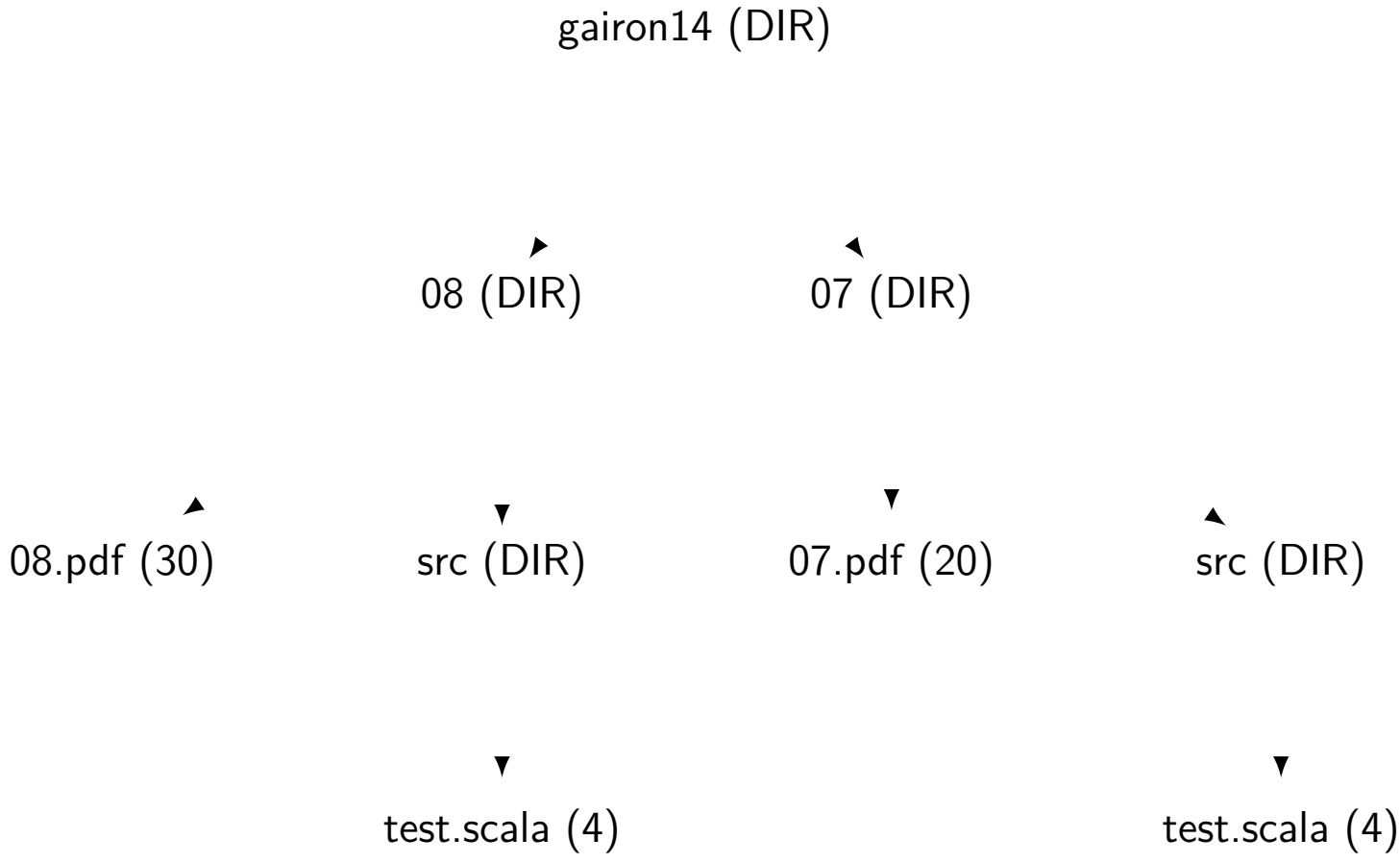
反復的な改良

研究者は現実世界の物事を数学で表現する．この数学的な表現をモデルという．以下のように反復的にモデルを改良して，正しいモデルを得る．



プログラムも反復的に開発していこう

例題: ファイルシステム



四角の内点はディレクトリ，楕円の内点はファイルを表し，ファイルは名前と共にファイルサイズが括弧内で書いてある．

モデルを作る: 最初の試み

モデル ver.1

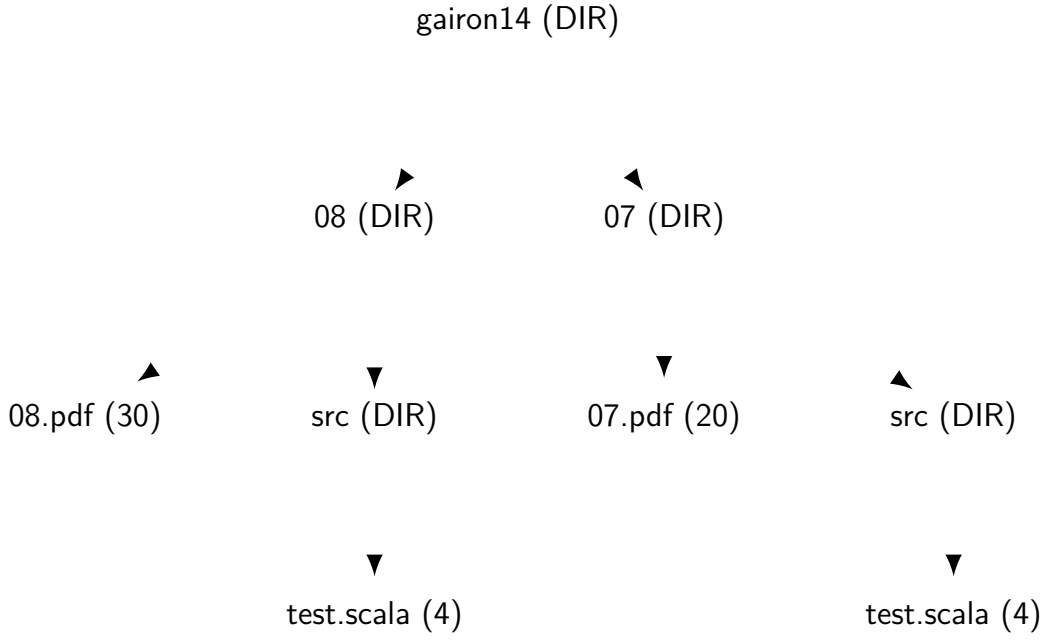
- ファイル $\triangleq$ `Symbol`
- ディレクトリ $\triangleq$ `List[DirFile]`
  ただし `DirFile` はファイルかディレクトリからなる

```
type File = Symbol

abstract class DirFile
type Directory = List[DirFile]
case class ContentFile(file:File) extends DirFile
case class ContentDir(dir: Directory) extends DirFile
```

`type A = B` は “`A` を `B` の別名とする” ことを意味する．`A` は今定義する型，`B` は定義済みの型であることに注意．逆には書けない．

サンプルコード




```
var pdf07 = '_07_pdf
var test07='test_scala
var src07=
  List(ContentFile(test07))
var d07=List(ContentFile(pdf07),
             ContentDir(src07))

var test08='test_scala
var src08=
  List(ContentFile(test08))
var pdf08='_08_pdf
var d08=List(ContentFile(pdf08),
             ContentDir(src08))

var gairon14 : Directory =
  List(ContentDir(d08),
       ContentDir(d07))
```

モデルの改良 1

ディレクトリは名前などの情報を持つ．これを表せるようにディレクトリのモデルを改良しよう．
モデル ver.2

- ディレクトリは名前と中身のリストからなる, e.g.,

```
case class Directory(name:Symbol, contents:List[DirContent])
```

- ディレクトリの中身 (`DirContent`) はファイルかディレクトリである (実質的には `DirFile` と同じ)

```
abstract class DirContent
case class ContentFile(file:File) extends DirContent
case class ContentDir(dir: Directory) extends DirContent
```

モデルの改良 2

モデル ver.2 ではファイルがただのシンボルで表されていた．これではファイルの大きさや中身を扱うことができない．ファイルの情報を表せるようにモデルを改良しよう．
モデル ver.3

- ファイルは名前と大きさを表す整数値と内容からなる,

  e.g.,

```
case class File(name: Symbol, size: Int, content: Symbol)
```

モデルの改良 3

モデル ver3 を更に改良して，ファイルとディレクトリを分けて管理するようにしよう．
モデル ver.4

- ディレクトリの中身はファイルのリストとディレクトリのリストである, e.g.,

```
case class Directory(name:Symbol, contents:DirContents)
case class DirContents(files: List[File],
                       dirs: List[Directory])
```

サンプルコード

```
val dir08 =
  Directory('_08,
    DirContents(List(File('_08_pdf,30,'a)),
                List(Directory('src,
                       DirContents(List(File('test_scala,4,'a)),
                                   Nil)))))
val dir07 =
  Directory('_07,
    DirContents(List(File('_07_pdf,20,'a)),
                List(Directory('src,
                       DirContents(List(File('test_scala,4,'a)),
                                   Nil)))))
val gairon14 = Directory('gairon14,
                     DirContents( Nil,
                                  List(dir07,dir08)))
```

モデルの比較

モデル ver.1

```
type File = Symbol
type Dir = List[DirFile]

abstract class DirFile
case class
  ContentFile(file:File)
  extends DirFile
case class
  ContentDir(dir: Dir)
  extends DirFile
```

モデル ver.4

```
case class
  File(name: Symbol,
       size: Int,
       content: Symbol)
case class
  Dir(name: Symbol,
      contents: DirContents)
case class
  DirContents(files: List[File],
              dirs: List[Dir])
```

目次

03 課題 3 のヒント

複雑な再帰的構造
- 家系図木
- 2 分木と 2 分探索木

反復的な改良によるプログラムの開発

2 つの複雑なデータの処理

2 本のリストを処理: その 1

問題
2 本の整数のリスト (`List[Int]`)alon1 と alon2 を受け取り，alon1 の Nil を alon2 に置き換えて新しいリストを作る函数 `replaceEoL` を開発せよ．

2 本のリストを処理: その 1

問題
2 本の整数のリスト (List[Int])alon1 と alon2 を受け取り，alon1 の Nil を alon2 に置き換えて新しいリストを作る函数 replaceEoL を開発せよ．

```
def replaceEoL:
(List[Int],List[Int]) => List[Int]
=
(alon1,alon2) =>
/**
 * 目的: alon1のNilをalon2に置き換えて新しいリストを作る
 * 例: ...
 */
{...}
```

2 本のリストを処理: その 1

問題
2 本の整数のリスト (List[Int])alon1 と alon2 を受け取り，alon1 の Nil を alon2 に置き換えて新しいリストを作る函数 replaceEoL を開発せよ．

```
def replaceEoL:
(List[Int],List[Int]) => List[Int]
=
(alon1,alon2) =>
/**
 * 目的: alon1のNilをalon2に置き換えて新しいリストを作る
 * 例: forall L:List[Int], replaceEoL(Nil,L) == L
 *     forall L:List[Int], replaceEoL(1::Nil,L) == 1::L
 */
{...}
```

replaceEoL: 例の分析

* 例: `forall L:List[Int], replaceEoL(Nil,L) == L`
* `forall L:List[Int], replaceEoL(1::Nil,L) == 1::L`

replaceEoL: 例の分析

```
* 例: forall L:List[Int], replaceEoL(Nil,L) == L
*      forall L:List[Int], replaceEoL(1::Nil,L) == 1::L
```

2 本目のリストは全く変更されていない
(そのまま結果の一部に使われているだけ)

replaceEoL: 例の分析

* 例: `forall L:List[Int], replaceEoL(Nil,L) == L`
* `forall L:List[Int], replaceEoL(1::Nil,L) == 1::L`

2 本目のリストは全く変更されていない
(そのまま結果の一部に使われているだけ)

⇓

`replaceEoL` の処理は 2 本目のリストの形には依存しない

replaceEoL: 例の分析

```
* 例: forall L:List[Int], replaceEoL(Nil,L) == L
*     forall L:List[Int], replaceEoL(1::Nil,L) == 1::L
```

2 本目のリストは全く変更されていない
(そのまま結果の一部に使われているだけ)

⇓

replaceEoL の処理は 2 本目のリストの形には依存しない

⇓

1 本目のリストを扱う函数として今まで通り設計すればよい

replaceEoL の完成

```
def replaceEoL:
(List[Int],List[Int]) => List[Int]
=
(alon1,alon2) =>
/**
 * 目的: alon1のNilをalon2に置き換えて新しいリストを作る
 * 例: forall L:List[Int], replaceEoL(Nil,L) == L
 *     forall L:List[Int], replaceEoL(1::Nil,L) == 1::L
 * テンプレート:
 * alon1 match{ case Nil => ...
 *              case e::rst => ...replaceEoL(rst,alon2)... }
 */
{alon1 match{
  case Nil => alon2
  case r::rst => r::replaceEoL(rst,alon2)
}}
```

2 本のリストを処理: その 2

問題
社員一人一人の労働時間と時給が別々のリストとして与えられる．この時，社員一人一人の給与を計算するプログラム `hoursToWages` を開発せよ．入力となる 2 本のリストの長さは等しい．

2 本のリストを処理: その 2

問題
社員一人一人の労働時間と時給が別々のリストとして与えられる．この時，社員一人一人の給与を計算するプログラム hoursToWages を開発せよ．入力となる 2 本のリストの長さは等しい．

```
def hoursToWages: (List[Int],List[Int]) => List[Int] =
(alon1, alon2) =>
/**
 * 目的: alon1とalon2のi番目同士を掛けあわせた値をi番目の要素
 *       とするリストを生成する
 * 例: ...
 */
{...}
```

2 本のリストを処理: その 2

問題
社員一人一人の労働時間と時給が別々のリストとして与えられる．この時，社員一人一人の給与を計算するプログラム hoursToWages を開発せよ．入力となる 2 本のリストの長さは等しい．

```
def hoursToWages: (List[Int],List[Int]) => List[Int] =
(alon1, alon2) =>
/**
 * 目的: alon1とalon2のi番目同士を掛けあわせた値をi番目の要素
 *       とするリストを生成する
 * 例: hoursToWages(Nil,Nil) == Nil
 *     hoursToWages(1::Nil,20::Nil) == 20::Nil
 */
{...}
```

hoursToWages: 例の分析

```
* 例: hoursToWages(Nil,Nil) == Nil
*     hoursToWages(1::Nil,20::Nil) == 20::Nil
```

hoursToWages: 例の分析

* 例: `hoursToWages(Nil,Nil) == Nil`
* `hoursToWages(1::Nil,20::Nil) == 20::Nil`

$$\texttt{alon1==Nil} \Longleftrightarrow \texttt{alon2==Nil} \text{ かつ}$$
$$\texttt{alon1==r1::rst1} \Longleftrightarrow \texttt{alon2==r2::rst2}$$

hoursToWages: 例の分析

```
 * 例: hoursToWages(Nil,Nil) == Nil
 *     hoursToWages(1::Nil,20::Nil) == 20::Nil
```

alon1==Nil $\Longleftrightarrow$ alon2==Nil かつ

alon1==r1::rst1 $\Longleftrightarrow$ alon2==r2::rst2

$\Downarrow$

```
(alon1,alon2) match{
  case (Nil,Nil) => ...
  case (r1::rst1, r2::rst2) => ...hoursToWages(rst1,rst2)...
}
```

hoursToWages の完成

```
def hoursToWages: (List[Int],List[Int]) => List[Int] =
(alon1, alon2) =>
/*
 * 目的: alon1とalon2のi番目同士を掛けあわせた値をi番目の要素
 *       とするリストを生成する
 * 例: hoursToWages(Nil,Nil) == Nil
 *     hoursToWages(1::Nil,20::Nil) == 20::Nil
 * テンプレート:
 * (alon1,alon2) match{
 *   case (Nil,Nil) =>
 *   case (r1::rst1, r2::rst2) => ...hoursToWages(rst1,rst2)...
 */
{(alon1,alon2)match{
  case (Nil,Nil) => Nil
  case (r1::rst1, r2::rst2) => r1*r2::hoursToWages(rst1,rst2)
}}
```

課題

以降の課題をとき，04.zip を OCW/OCW-i を使って提出せよ．
提出締め切りは6 月 4 日 15:00(JST)とする．

- 3 つ全て解けなくても提出する方が良い (課題ごとに採点するため)

課題 1: 瞳の色の収集 (file: color.scala)

`ParentTree` が与えられたとき，この中に現れる人物たちの瞳の色を全て収集する函数 `colors` を開発せよ．

課題 2: dna.scala

シンボルのリストが 2 本与えられる．1 本目のリストは DNA のパターンを表し，2 本目のリストは DNA を表しているものとしよう．パターンが DNA の接頭辞 (prefix) となっていれば真，さもなくば偽を返す函数 `DNAprefix` を開発せよ．

課題 2: dna.scala

シンボルのリストが 2 本与えられる．1 本目のリストは DNA のパターンを表し，2 本目のリストは DNA を表しているものとしよう．パターンが DNA の接頭辞 (prefix) となっていれば真，さもなくば偽を返す函数 `DNAprefix` を開発せよ．
例

- `DNAprefix(List('a,'t),List('a,'t,'c)) == true`
- `DNAprefix(List('a,'t),List('a)) == false`
- `DNAprefix(List('a,'t),List('a,'t)) == true`
- `DNAprefix(List('a,'c,'g,'t),List('a,'g,'c,'t,'d)) == false`
- `DNAprefix(List('a,'a,'c,'t),List('a,'a,'c,'t,'d)) == true`
- `DNAprefix(List('a,'a,'c,'t),List('a,'c,'t,'d)) == false`

課題 3: gift.scala

Louise，Jane，Laura，Dana はあなたの友達です．彼女たちはクリスマスでプレゼント交換をする予定で，あなたはその割り当て方を決めるプログラムを作って欲しいと依頼されました．自分自身のプレゼントが自分自身に渡されるようなことが無いようにして，プレゼントの可能な渡し方を全て列挙するプログラム `giftPick` を開発しましょう．

`giftPick` は人物名 (シンボル) のリスト `lst` を受け取り，リストのリスト `List(lst',lst'',...)` を返す．`lst'` や `lst''` の *i*番目のシンボルは `lst` の *i*番目の人がプレゼントを渡す相手の名前である．

gift.scala: 例

- `giftPick(List('Louise,'Jane))` ≡
  `List(List('Jane, 'Louise))`
- `giftPick(List('Louise,'Jane,'Laura))` ≡
  以下を要素とするリスト
  - `List('Laura, 'Louise, 'Jane)`
  - `List('Jane, 'Laura, 'Louise)`