計算機科学概論演習

#03

青谷知幸

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

April 20, 2016

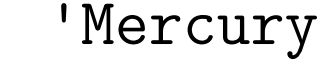
目次

リストを作る

- 空のリストを作る: `Nil`
- リスト `lst` の先頭に新しい要素 `e` を付け加える: `e::lst`
- 値 `a`, `b`, `c`, $\cdots$ からなるリストを作る: `List(a,b,c,`$\cdots$`)`

例:

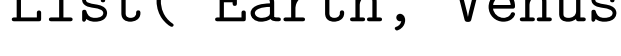
- `'Mercury::Nil` = `List('Mercury)`

- `'Venus::'Mercury::Nil` = `List('Venus,'Mercury)`

- `'Earth::'Venus::'Mercury::Nil` = `List('Earth,'Venus,'Mercury)`

リストの型

リストの型は `List[A]` で表される．`A` は要素の型である．

例:

```
scala> 'Mercury::Nil
res0: List[Symbol] = List('Mercury)

scala> 'Venus::'Mercury::Nil
res1: List[Symbol] = List('Venus, 'Mercury)

scala> 'Earth::'Venus::'Mercury::Nil
res2: List[Symbol] = List('Earth, 'Venus, 'Mercury)
```

リストの分解/値を取り出す

リスト lst を分解するには match 式を使う

```
lst match{
  case Nil => ... //lst が空の時
  case (hd::tl) => ... //先頭が hd，残りのリストが tl の時
}
```

hd::tl はリストの分解の仕方の定義である.

応用: 長さ 3 の整数のリストの要素の総和を求める

```
def addUp3: List[Int] => Int
= (lst) => lst match{
  case h1::h2::h3::rest => h1 + h2 + h3
}
```

rest は先頭の 3 要素を取り除いた残りのリストを表している. rest という変数を与える代わりに，以下のように Nil として具体的な値を書いても良い.

```
def addUp3Nil: List[Int] => Int
= (lst) => lst match{
  case h1::h2::h3::Nil => h1 + h2 + h3
}
```

addUp3 と addUp3Nil の違い

addUp3 には長さ 4 以上の整数のリストを渡せるが，addUp3Nil には長さ 4 以上の整数のリストを渡すとエラーになる.

```
scala> addUp3(List(1,2,3,4))
res0: Int = 6

scala> addUp3Nil(List(1,2,3,4))
scala.MatchError: List(1, 2, 3, 4) (of class ...)
...
```

addUp3 と addUp3Nil の違い

```
def addUp3: List[Int] => Int
= (lst) => lst match{
  case h1::h2::h3::rest => h1 + h2 + h3
}
```

addUp3(List(1,2,3))
≡ //lst を List(1,2,3) で置換して定義を展開
List(1,2,3) match {case h1::h2::h3::rest => h1 + h2 + h3}
≡ //List(1,2,3)=1::2::3::Nil, h1=1, h2=2, h3=3, rest=Nil
1 + 2 + 3
≡
6

addUp3 と addUp3Nil の違い

```
def addUp3Nil: List[Int] => Int
= (lst) => lst match{
  case h1::h2::h3::Nil => h1 + h2 + h3
}
```

```
addUp3Nil(List(1,2,3))
≡ //lst を List(1,2,3) で置換して定義を展開
List(1,2,3) match {case h1::h2::h3::Nil => h1 + h2 + h3}
≡ //List(1,2,3)=1::2::3::Nil, h1=1, h2=2, h3=3, Nil=Nil
1 + 2 + 3
≡
6
```

Nil=Nil は値の同値判定であることに注意

addUp3 と addUp3Nil の違い

```
def addUp3: List[Int] => Int
= (lst) => lst match{
  case h1::h2::h3::rest => h1 + h2 + h3
}
```

addUp3(List(1,2,3,4))
≡ //lst を List(1,2,3,4) で置換して定義を展開
List(1,2,3,4) match {case h1::h2::h3::rest => h1 + h2 + h3}
≡ //List(1,2,3,4)=1::2::3::4::Nil, h1=1, h2=2, h3=3, rest=4::Nil
1 + 2 + 3
≡
6

addUp3 と addUp3Nil の違い

```
def addUp3Nil: List[Int] => Int
= (lst) => lst match{
  case h1::h2::h3::Nil => h1 + h2 + h3
}
```

addUp3Nil(List(1,2,3,4))
≡ //lst を List(1,2,3,4) で置換して定義を展開
List(1,2,3,4) match {case h1::h2::h3::Nil => h1 + h2 + h3}
≡ //List(1,2,3,4)=1::2::3::4::Nil, h1=1, h2=2, h3=3, Nil=4::Nil
Error

Nil(=List()) は 4::Nil(=List(4)) と等しくないのでエラー

リストデータ構造の定義

Scala におけるリストの定義は大凡以下のようになっている

```
abstract class List
case class Nil extends List
case class ::(head:A, tl:List) extends List
```

ただし A はリストの要素の型.

リストデータ構造の定義

Scala におけるリストの定義は大凡以下のようになっている

```
abstract class List
case class Nil extends List
case class ::(head:A, tl:List) extends List
```

ただし A はリストの要素の型.

cf. 図形の定義

```
abstract class Shape
case class Circle(center:Position, radius:Int) extends Shape
case class Square(center:Position, length:Int) extends Shape
```

リストデータ構造の定義

Scala におけるリストの定義は大凡以下のようになっている

```
abstract class List
case class Nil extends List
case class ::(head:A, tl:List) extends List
```

ただし `A` はリストの要素の型.

`::` の第 2 引数が `List` 型の値であることに注意. 再帰的な構造になっている.

リストデータ構造の定義

Scala におけるリストの定義は大凡以下のようになっている

```
abstract class List
case class Nil extends List
case class ::(head:A, tl:List) extends List
```

ただし `A` はリストの要素の型.

`::` の第 2 引数が `List` 型の値であることに注意. 再帰的な構造になっている.

Q. ところで `head` の型が `A` になってるけどこれはいいの？

リストデータ構造の定義

Scala におけるリストの定義は大凡以下のようになっている

```
abstract class List
case class Nil extends List
case class ::(head:A, tl:List) extends List
```

ただし `A` はリストの要素の型.

`::` の第 2 引数が `List` 型の値であることに注意. 再帰的な構造になっている.

Q. ところで `head` の型が `A` になってるけどこれはいいの？

A. 厳密にはよくないです (これだと `A` という型がどこかに定義されている話になってしまう)

リストの (より正しい) 定義

```
abstract class List[A]
case class Nil extends List[Nothing]
case class ::[A](head:A, tl:List[A]) extends List[A]
```

- List は型 A を受け取って List[A] という型を返す函数
- 角括弧が 2 種類の意味で使われている
  - 今定義しようとしている型の直後: 型を表す変数 (型変数) の定義[1]
  - 既に定義されている型の直後: 型変数に型を渡す[2]

[1] cf. def f:(Int)=>Int=(x)=>{...} において x は Int の値を表す変数

[2] cf. def g:(Int)=>Int=(x)=>{f(x)} において f(x) は函数 f に変数 x の値を渡している

任意の長さのリストを扱う

例題
とある玩具店の目録がある．この目録を調べて，人形の在庫があるかどうかを確かめるプログラム `hasDoll` を開発せよ．目録は物品のシンボルで表された名前のリスト，人形は`'doll` で表されるものとする．

おさらい: データ構造と函数定義の仕方

データ構造を使うプログラムは以下のように作ると良い

1. データの解析と設計
   - 問題文中に登場する計算対象物 (`Entry` など) の計算に関係する特徴を決定
   - データ型を定義
2. 函数の型と目的の記述
3. 函数の利用例の記述
4. 函数のテンプレートの記述
   - 新たに定義したデータ型の引数について，その値の分解の仕方を記述 (確認の意味で)
   - もしその函数が条件分岐を含むなら，全ての適切な分岐を列挙しておく
5. 函数の本体 (中身) の記述

1. データの解析と設計

例題
とある玩具店の目録がある．この目録を調べて，人形の在庫があるかどうかを確かめるプログラム `hasDoll` を開発せよ．目録は物品のシンボルで表された名前のリスト，人形は`'doll` で表されるものとする．

- 目録: `List[Symbol]` で表す
- 人形: `'doll:Symbol` で表す

2. 函数の型と目的の記述

例題
とある玩具店の目録がある．この目録を調べて，人形の在庫があるかどうかを確かめるプログラム hasDoll を開発せよ．目録は物品のシンボルで表された名前のリスト，人形は'doll で表されるものとする．

```
def hasDoll:
(List[Symbol]) => Boolean
=
(inventory) =>
/**
 * 目的: 'dollがinventoryに含まれているかどうかを調べる
 */
{...}
```

3. 函数の利用例の記述

例題
とある玩具店の目録がある．この目録を調べて，人形の在庫があるかどうかを確かめるプログラム hasDoll を開発せよ．目録は物品のシンボルで表された名前のリスト，人形は'doll で表されるものとする．

```
def hasDoll:
(List[Symbol]) => Boolean
=
(inventory) =>
/**
 * 目的: 'dollがinventoryに含まれているかどうかを調べる
 * 例: hasDoll(List('doll, 'watergun, 'crown)) は 真
 *     hasDoll(List('car, 'watergun, 'crown)) は 偽
 */
{...}
```

4. 函数のテンプレート

```
def hasDoll: (List[Symbol]) => Boolean =
(inventory) =>
/**
 * 目的: 'dollがinventoryに含まれているかどうかを調べる
 * 例: hasDoll(List('doll, 'watergun, 'crown)) は 真
 *     hasDoll(List('car, 'watergun, 'crown)) は 偽
 * テンプレート:
 *   inventory match { case Nil => ...
 *                     case e::rest =>
 *                       if (e=='doll) ... else ...
 *   }
 */
{...}
```

5. 函数の本体

```
def hasDoll: (List[Symbol]) => Boolean =
(inventory) =>
/**
 * 目的: 'dollがinventoryに含まれているかどうかを調べる
 * 例: hasDoll(List('doll, 'watergun, 'crown)) は 真
 *     hasDoll(List('car, 'watergun, 'crown)) は 偽
 * テンプレート:
 *   inventory match { case Nil => ...
 *                     case e::rest =>
 *                       if (e=='doll) ... else ...
 *   }
 */
{inventory match{
  case Nil => false
  case e::lst => if (e=='doll) true else ???
}}
```

5. 函数の本体

```
def hasDoll: (List[Symbol]) => Boolean =
(inventory) =>
/**
 * 目的: 'dollがinventoryに含まれているかどうかを調べる
 * 例: hasDoll(List('doll, 'watergun, 'crown)) は 真
 *     hasDoll(List('car, 'watergun, 'crown)) は 偽
 * テンプレート:
 *   inventory match { case Nil => ...
 *                     case e::rest =>
 *                       if (e=='doll) ... else ...
 *   }
 */
{inventory match{
  case Nil => false
  case e::lst => if (e=='doll) true else hasDoll(lst)
}}
```

目次

再帰的データ構造とそれを扱う函数の設計の仕方

既に見たように，再帰的なデータ構造とそれを扱う函数は，普通のデータ構造およびそれを扱う函数と同じ方法で定義・設計すれば良い.

いつ再帰的なデータ構造を使うか？
データ解析と設計の段階 (Step 1) で判断する．問題文が任意の大きさ・長さのデータを扱うように書かれていた場合に再帰的なデータ構造を用いる．再帰的なデータ構造は大体以下の方法で定義できる

```
abstract class RecData
case class NotRecursiveOne(...) extends RecData //... の中には RecData が現れない
case class NotRecursiveTwo(...) extends RecData //... の中には RecData が現れない
```

テンプレートの改良

再帰的なデータ構造を扱う函数は往々にして再帰的に定義される．テンプレートを記述する際に，函数の再帰的な使用を明記してしまうとよい

例: 目録の大きさを求める函数 howLarge

```
def howLarge: (List[Symbol]) => Int =
(lst) =>
/**
 * テンプレート:
 * lst match {
 *   case Nil => ...
 *   case e::rest => ... howLarge(rest) ...
 * }
 */
{...}
```

例題: 総額を求める

問題文
オンラインショッピングサイトのためのプログラムを開発しているとする．顧客のカートに入っている商品のそれぞれの金額がわかっているとき，カート内の商品の金額の総和を求めるプログラム sum を開発せよ．

1. データ分析と定義

問題文
オンラインショッピングサイトのためのプログラムを開発しているとする．顧客のカートに入っている商品のそれぞれの金額がわかっているとき，カート内の商品の金額の総和を求めるプログラム `sum` を開発せよ．

- カートに入っている商品の個数は任意である．したがって再帰的なデータ構造を用いる必要がある．
- カートは商品のそれぞれの金額のリスト `List[Long]` で表されているものとする

2. 函数の型と目的の記述

問題文
オンラインショッピングサイトのためのプログラムを開発しているとする．顧客のカートに入っている商品のそれぞれの金額がわかっているとき，カート内の商品の金額の総和を求めるプログラム sum を開発せよ．

```
def sum: (List[Long]) => Long
=
(cart) =>
/**
 * 目的: cart内の値の総和を求める
 */
{...}
```

3. 函数の利用例の記述

問題文
オンラインショッピングサイトのためのプログラムを開発しているとする．顧客のカートに入っている商品のそれぞれの金額がわかっているとき，カート内の商品の金額の総和を求めるプログラム sum を開発せよ．

```
def sum: (List[Long]) => Long
=
(cart) =>
/**
 * 目的: cart内の値の総和を求める
 * 例: sum(List(1,2,3)) == 6
 *     sum(List(1,2,3,4)) == 10
 */
{...}
```

4. テンプレートの記述

```
def sum: (List[Long]) => Long
=
(cart) =>
/**
 * 目的: cart内の値の総和を求める
 * 例: sum(List(1,2,3)) == 6
 *     sum(List(1,2,3,4)) == 10
 * テンプレート:
 *   cart match{
 *     case Nil => ...
 *     case e::rest => ...sum(rest)...
 *   }
 */
{...}
```

5. 本体の記述

```
def sum: (List[Long]) => Long
=
(cart) =>
/**
 * 目的: cart内の値の総和を求める
 * 例: sum(List(1,2,3)) == 6
 *     sum(List(1,2,3,4)) == 10
 * テンプレート:
 *   cart match{
 *     case Nil => ...
 *     case e::rest => ...sum(rest)...
 *   }
 */
{ cart match{
  case Nil => 0
  case e::rest => e + sum(rest)
}}
```

目次

リストを作る函数

問題
社員全員について，それぞれの労働時間が与えられた時，それぞれの給与を計算するプログラム `hoursToWages` を開発せよ．
時給は 1200 円とする．

リストを作る函数

問題
社員全員について，それぞれの労働時間が与えられた時，それぞれの給与を計算するプログラム `hoursToWages` を開発せよ．
時給は 1200 円とする．

- 入力 $= 0$ 人以上の社員のそれぞれの労働時間: `List[Int]`
- 出力 $= 0$ 人以上の社員のそれぞれの給料: `List[Int]`

リストを作る函数

問題
社員全員について，それぞれの労働時間が与えられた時，それぞれの給与を計算するプログラム hoursToWages を開発せよ．
時給は 1200 円とする．

```
def hoursToWages: (List[Int]) => (List[Int]) =
(hours) =>
/**
 * 目的: 時間のリストから給料のリストを作る
 */
{...}
```

リストを作る函数

問題
社員全員について，それぞれの労働時間が与えられた時，それぞれの給与を計算するプログラム hoursToWages を開発せよ．
時給は 1200 円とする．

```
def hoursToWages: (List[Int]) => (List[Int]) =
(hours) =>
/**
 * 目的: 時間のリストから給料のリストを作る
 * 例: hoursToWages(Nil) == Nil
 *     hoursToWages(List(28)) == List(33600)
 *     hoursToWages(List(40,28)) == List(48000,33600)
 */
{...}
```

リストを作る函数

```
def hoursToWages: (List[Int]) => (List[Int]) =
(hours) =>
/**
 * 目的: 時間のリストから給料のリストを作る
 * 例: hoursToWages(Nil) == Nil
 *     hoursToWages(List(28)) == List(33600)
 *     hoursToWages(List(40,28)) == List(48000,33600)
 * テンプレート:
 *   hours match{
 *     case Nil => ...
 *     case h::rest => ... hoursToWages(rest) ...
 *   }
 */
{...}
```

リストを作る函数

```
def hoursToWages: (List[Int]) => (List[Int]) =
(hours) =>
/**
 * 目的: 時間のリストから給料のリストを作る
 * 例: hoursToWages(Nil) == Nil
 *     hoursToWages(List(28)) == List(33600)
 *     hoursToWages(List(40,28)) == List(48000,33600)
 * テンプレート:
 *   hours match{
 *     case Nil => ...
 *     case h::rest => ... hoursToWages(rest) ...
 *   }
 */
{ hours match{
    case Nil => Nil
    case h::rest => wage(h)::hoursToWages(rest)
}}
```

リストを作る函数

補助函数 wage の定義は以下の通り:

```
def wage: Int => Int =
(h) =>
/**
 * ...
 */
{1200 * h}
```

複雑なデータのリスト

現実世界のデータを扱うプログラムでは，往々にしてリストの要素が複雑なデータになる．

- 玩具店の目録:
  `List[`商品名と値段`]`
- オンラインショッピングサイトのカートの内容:
  `List[`商品名と値段と個数`]`
- 給与計算の入力:
  `List[`被雇用者のデータ（名前，年齢など）と労働時間`]`

例題: 総額計算

問題
カートの中にある商品の総額を計算するプログラム sumUp を開発せよ．カートには商品名とその単価，そして個数が複数並んでいるものとする．

例題: 総額計算

問題
カートの中にある商品の総額を計算するプログラム sumUp を開発せよ．カートには商品名とその単価，そして個数が複数並んでいるものとする．

- 入力 = 商品名と単価と個数の任意個の並び: List[Record]
  - 商品名と単価と個数の 3 つ組: `Record`
  - 商品名 (`name`): `Symbol`
  - 単価 (`price`): `Long`
  - 個数 (`num`): `Long`
- 出力 = 総額: Long

例題: 総額計算

問題
カートの中にある商品の総額を計算するプログラム sumUp を開発せよ．カートには商品名とその単価，そして個数が複数並んでいるものとする．

```
case class Record(name:Symbol, price:Long, num:Long)

def sumUp: (List[Record]) => Long =
(cart) =>
/**
 * ...
 */
{...}
```

例題: 総額計算

```
case class Record(name:Symbol, price:Long, num:Long)

def sumUp: (List[Record]) => Long =
(cart) =>
/**
 * テンプレート:
 *   cart match{
 *     case Nil => ...
 *     case r::rest => r match {
 *       case Record(a,p,n) => ... sumUp(rest) ...
 *     }}
 */
{...}
```

例題: 総額計算

```
def sumUp: (List[Record]) => Long =
(cart) =>
/**
 * テンプレート:
 *   cart match{
 *     case Nil => ...
 *     case r::rest => r match {
 *       case Record(a,p,n) => ... sumUp(rest) ...
 *     }}
 */
{cart match{
  case Nil => 0
  case r::rest => r match{
    case Record(a,p,n) => p*n+sumUp(rest)
  }
}}
```

目次

自然数を定義する

- 0は自然数である
- nが自然数のとき，nより 1 大きい数は自然数である

自然数を定義する

- 0は自然数である
- nが自然数のとき，nより 1 大きい数は自然数である

Scala で自然数を定義する

```
abstract class Nat
case class Zero() extends Nat
case class Succ(prev:Nat) extends Nat
```

- $0 \triangleq$ `Zero()`
- $2 \triangleq$ `Succ(Succ(Zero()))`
- $3 \triangleq$ `Succ(Succ(Succ(Zero())))`

自然数を定義する

- 0は自然数である
- nが自然数のとき，nより 1 大きい数は自然数である

Scala で自然数を定義する

```
abstract class Nat
case class Zero() extends Nat
case class Succ(prev:Nat) extends Nat

def pred: (Nat) => Nat = (n) =>
/**
 * 目的: n == Succ(n') となるn'を計算
 */
{ n match{ case Succ(p) => p }}
```

自然数を定義する

- 0は自然数である
- nが自然数のとき，nより 1 大きい数は自然数である

Scala で自然数を定義する

```
abstract class Nat
case class Zero() extends Nat
case class Succ(prev:Nat) extends Nat

def pred: (Nat) => Nat = (n) =>
/**
 * 目的: n == Succ(n') となるn'を計算
 */
{ n match{ case Succ(p) => p }}
```

- `pred(Succ(Succ(Zero()))) == Succ(Zero())`

任意の自然数を扱う

自然数 n が与えられた時，n 個の'hello をリストを生成する函数 hellos を開発せよ.

任意の自然数を扱う

自然数 n が与えられた時，n 個の'hello をリストを生成する函数 hellos を開発せよ.

```
def hellos : (Nat) => List[Symbol] = (n) =>
/**
 * 目的: n個の'helloからなるリストを作る
 * 例: hellos(Succ(Succ(Zero()))) == List('hello, 'hello)
 * テンプレート: n match{ case Zero() => ...
 *                        case Succ(p) => ...hellos(p)...}
 */
{...}
```

任意の自然数を扱う

自然数 n が与えられた時，n 個の'hello をリストを生成する函数 hellos を開発せよ.

```
def hellos : (Nat) => List[Symbol] = (n) =>
/**
 * 目的: n個の'helloからなるリストを作る
 * 例: hellos(Succ(Succ(Zero()))) == List('hello, 'hello)
 * テンプレート: n match{ case Zero() => ...
 *                        case Succ(p) => ...hellos(p)...}
 */
{ n match{
  case Zero() => Nil
  case Succ(p) => 'hello :: hellos(p)
}}
```

自然数の値を Int で表す

Nat ではなく Int で自然数を表したことにすることも可能.

```
def predInt : (Int) => Int = (n) => { n-1 }
def hellosInt : (Int) => List[Symbol] =
(n) =>
/**
 * 目的: n個の'helloからなるリストを作る
 * 例: hellosInt(2) == List('hello, 'hello)
 * テンプレート: n match{ case 0 => ...
 *                        case m if m > 0 =>
 *                            ...hellosInt(predInt(m))...}
 */
{n match {
  case 0 => Nil
  case m if m > 0 => 'hello::hellosInt(predInt(m))
}}
```

階乗計算

問題
0 以上の整数 n が与えられた時，n の階乗を計算するプログラムを開発せよ

階乗計算

問題
0 以上の整数 $\mathtt{n}$ が与えられた時，$\mathtt{n}$ の階乗を計算するプログラムを開発せよ
in Math.

$$!\mathtt{n} = \begin{cases} \mathtt{n} \times !(\mathtt{n}-1) & \text{if } \mathtt{n} \geq 1 \\ 1 & \text{if } \mathtt{n} = 0 \end{cases}$$

階乗計算

問題
0 以上の整数 n が与えられた時，n の階乗を計算するプログラムを開発せよ
in Scala

```
def fact: (Int) => Int = (n) =>
/**
 * 目的: 1×···×nを計算する
 * 例: fact(3) == 6, fact(10) == 3628800
 * テンプレート: n match{ case 0 => ...
 *                        case m if m >= 1 =>
 *                              ...fact(predInt(m))...}
 */
{ n match{
  case 0 => 1
  case m if m >= 1 => m*fact(predInt(m))
}}
```

目次

おさらい: 補助函数

問題文中または例の計算中に現れる量 (数値など) の間の依存関係ごとに補助函数を定義せよ

設計のヒント

1. 入力と出力の解析をする
2. テンプレートを作る
3. テンプレートを埋めて完全な関数定義を作る
   (テンプレート中に現れる部分式の結果をつなぎあわせて最終的な計算結果を作る)
   - 3.1 場合分けが必要なとき:
     `match` と/または `if-else` 式を使う
   - 3.2 特定の問題領域の知識が必要なとき:
     補助函数を作る・使う
   - 3.3 リストや自然数などの任意の長さのデータを扱うとき:
     補助函数を作る・使う
   - 3.4 函数が作りたいプログラムの一般化になるとき:
     その函数を呼び出す函数としてプログラムを定義する

補助函数の管理

補助函数を作る・使うことに決めたら，その函数の型・引数・目的を書いたものをウィッシュリストに入れて，プログラムの開発を管理するとよい.

ウィッシュリスト
プログラムを完成させるのに必要な (補助) 函数を列挙したメモのこと．この中にある函数を一つずつ，プログラム設計法に従って開発していくと良い.

補助函数の管理

補助函数を作る・使うことに決めたら，その函数の型・引数・目的を書いたものをウィッシュリストに入れて，プログラムの開発を管理するとよい.

ウィッシュリスト
プログラムを完成させるのに必要な (補助) 函数を列挙したメモのこと．この中にある函数を一つずつ，プログラム設計法に従って開発していくと良い.

(現実の) プログラム開発における注意点
函数をウィッシュリストに入れる前に，その函数が既に存在していないかどうかを可能な限り調べるべきである.

補助関数を作る・使う

例題
整数のリストが与えられた時，このリスト内の整数を降順に並べたリストを返す函数 `sort` を開発せよ.

補助関数を作る・使う

例題
整数のリストが与えられた時，このリスト内の整数を降順に並べたリストを返す函数 sort を開発せよ.

```
def sort: (List[Int])=>List[Int] = (lst) =>
/**
 * 目的: lstを降順にソートする
 * 例: sort(List(3,4,2,9)) == List(9,4,3,2)
 * テンプレート: lst match{ case Nil => ...
 *                          case e::rest => ...sort(rest)...}
 */
{...}
```

補助関数を作る・使う

例題
整数のリストが与えられた時，このリスト内の整数を降順に並べたリストを返す函数 sort を開発せよ.

```
def sort: (List[Int])=>List[Int] = (lst) =>
/**
 * 目的: lstを降順にソートする
 * 例: sort(List(3,4,2,9)) == List(9,4,3,2)
 * テンプレート: lst match{ case Nil => ...
 *                          case e::rest => ...sort(rest)...}
 */
{ lst match{
  case Nil => Nil
  case e::rest => e::sort(rest)
}}
```

補助関数を作る・使う

例題
整数のリストが与えられた時，このリスト内の整数を降順に並べたリストを返す函数 sort を開発せよ.

```
def sort: (List[Int])=>List[Int] = (lst) =>
/**
 * 目的: lstを降順にソートする
 * 例: sort(List(3,4,2,9)) == List(9,4,3,2)
 * テンプレート: lst match{ case Nil => ...
 *                          case e::rest => ...sort(rest)...}
 */
{ lst match{
  case Nil => Nil
  case e::rest => e::sort(rest) //ダメ
}}
```

e を sort(rest) の適切な位置に挿入する必要あり

補助関数を作る・使う

例題
整数のリストが与えられた時，このリスト内の整数を降順に並べたリストを返す函数 sort を開発せよ.

```
def sort: (List[Int])=>List[Int] = (lst) =>
/**
 * 目的: lstを降順にソートする
 * 例: sort(List(3,4,2,9)) == List(9,4,3,2)
 * テンプレート: lst match{ case Nil => ...
 *                          case e::rest => ...sort(rest)...}
 */
{ lst match{
  case Nil => Nil
  case e::rest => insert(e,sort(rest))
}}
```

補助関数を作る・使う

例題
整数のリストが与えられた時，このリスト内の整数を降順に並べたリストを返す函数 sort を開発せよ.

```
def insert:(Int,List[Int])=>List[Int] = (i,lst) =>
/**
 * 目的: iを降順ソート済みのリストlst中の要素eでi>=eを満たす
 *       最も左の要素の左に置く
 * 例: insert(3,List(6,4,2)) == List(6,4,3,2)
 * テンプレート: lst match{ case Nil => ...
 *                          case e::rest =>
 *                            if(i>=e) ...
 *                            else ...insert(i,rest)... }
 */
{...}
```

補助関数を作る・使う

```
def insert:(Int,List[Int])=>List[Int] = (i,lst) =>
/**
 * 目的: iを降順ソート済みのリストlst中の要素eでi>=eを満たす
 *       最も左の要素の左に置く
 * 例: insert(3,List(6,4,2)) == List(6,4,3,2)
 * テンプレート: lst match{ case Nil => ...
 *                          case e::rest =>
 *                            if(i>=e) ...
 *                            else ...insert(i,rest)... }
 */
{ lst match{
  case Nil => List(i)
  case e::rest =>
    if(i >= e) i::lst
    else e::(insert(i,rest))
}}
```

問題・函数の一般化

例題
多角形を描画するプログラム drawPolygon を開発せよ．ただし，頂点と頂点を結ぶ線を引く函数:

```
drawLine: (Position,Position) => Boolean
```

が使えるものとする.

問題・函数の一般化

例題
多角形を描画するプログラム drawPolygon を開発せよ．ただし，頂点と頂点を結ぶ線を引く函数:
`drawLine: (Position,Position) => Boolean`
が使えるものとする.

データの分析

- 入力: 多角形 $\triangleq$ `List[Position]` (頂点のリスト)
- 出力: `Boolean` (何でも良い)

問題・函数の一般化

例題
多角形を描画するプログラム drawPolygon を開発せよ．ただし，頂点と頂点を結ぶ線を引く函数:

```
drawLine: (Position,Position) => Boolean
```

が使えるものとする.

```
def drawPolygon: (List[Position]) => Boolean = (p) =>
/**
 * 目的: 多角形pを描画する
 * テンプレート: p match{ case Nil => ...
 *                        case e::Nil => ...
 *                        case b::e::rest =>
 *                            ...drawPolygon(e::rest)...}
 */
{...}
```

問題・函数の一般化

```
def drawPolygon: (List[Position]) => Boolean = (p) =>
/**
 * 目的: 多角形pを描画する
 * テンプレート: p match{ case Nil => ...
 *                        case e::Nil => ...
 *                        case b::e::rest =>
 *                           ...drawPolygon(e::rest)...}
 */
{ p match{
  case Nil => true
  case e::Nil => true
  case b::e::rest => drawLine(b,e) && drawPolygon(e::rest)
}}
```

Q: これでよい？

問題・函数の一般化

```
def drawPolygon: (List[Position]) => Boolean = (p) =>
/**
 * 目的: 多角形pを描画する
 * テンプレート: p match{ case Nil => ...
 *                        case e::Nil => ...
 *                        case b::e::rest =>
 *                           ...drawPolygon(e::rest)...}
 */
{ p match{
  case Nil => true
  case e::Nil => true
  case b::e::rest => drawLine(b,e) && drawPolygon(e::rest)
}}
```

Q: これでよい？

A: 辺が一本足りないのでダメ！

問題・函数の一般化

解決策
これまで定義した函数を `connectDots` とし，`drawPolygon` をそれを呼び出す函数として定義する

問題・函数の一般化

```
def connectDots: (List[Position]) => Boolean = (p) =>
/**
 * 目的: 多角形pを描画する
 * テンプレート: p match{ case Nil => ...
 *                        case e::Nil => ...
 *                        case b::e::rest =>
 *                           ...connectDots(e::rest)...}
 */
{ p match{
  case Nil => true
  case e::Nil => true
  case b::e::rest => drawLine(b,e) && connectDots(e::rest)
}}
```

問題・函数の一般化

```
def drawPolygon:(List[Position]) => Boolean = (p) =>
{p match{
  case Nil => true
  case e::rest => connectDots(p ++ List(e))
}}
```

目次

課題

以降の課題をとき，03.zip を OCW/OCW-i を使って提出せよ．
提出締め切りは4 月 28 日 0:00(JST)とする．

課題 1: add.scala

Nat 型の値を 2 つ受け取ってその和を計算する函数 add を開発せよ.
Nat の定義

```
abstract class Nat
case class Zero() extends Nat
case class Succ(prev:Nat) extends Nat
```

例

- add(Zero(),Zero()) == Zero()
- add(Succ(Zero()),Succ(Zero())) == Succ(Succ(Zero()))

課題 2: 店選びプログラム

(file:isok.scala)

予算の上限と店のコース料理/定食のメニューがこの順序で与えられた時，メニューに 1 つ以上のコース/定食があり，かつそれぞれのコース/定食が限られた予算で食べられれば `true`，そうでなければ `false` を返す函数 `isOK` を開発せよ．予算の上限は `Int` の値，コース料理/定食のメニューは以下で定義される `Plate` のリストで表されているものとする．

```
case class Plate(name:Symbol, price:Int)
```

ここで `name` は料理の名前，`price` は税込の値段である．

例

- `isOK(1000,List(Plate('A,800),Plate('B,1000))=true`
- `isOK(1000,List(Plate('A,800),Plate('B,1200))=false`
- `isOK(1000,Nil)=false`

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

課題の取り組み方

- /home/aotani/Documents/classes/gairon16/03 をコピーして始めること．これはターミナルで以下を実行すればよい．

  ```
  source /home/aotani/local/bin/gairon-copy.sh 03 ↩
  ```

- 各プログラムはそれぞれ $HOME/Documents/gairon16/03/src/main/resources/ 以下に用意されている対応ファイルに記入すること
- 提出ファイルは 03.zip である．これはターミナルで以下を実行することで自動生成される．

  ```
  source /home/aotani/local/bin/gairon-copy.sh 03 ↩
  ./makezip.sh ↩
  ```

  こうすることで，”書類 (Documents)” ディレクトリの下の”gairon16” ディレクトリの下の”03” ディレクトリに 03.zip ができる．