計算機科学概論演習

#05

青谷知幸

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

May 11, 2016

よいプログラマーの鉄則

とても良く似た函数を
いくつもかかない！

目次

値の抽象化

函数の抽象化

型の抽象化

局所的な函数の定義

函数を生成する (返す) 函数

課題

おさらい: hasDoll

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: hasDoll

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) =>
{inventory match{
  case Nil => false
  case e::lst => if (e=='doll) true else hasDoll(lst)
}}
```

おさらい: hasWatergun

例題

とある玩具店の目録がある．この目録を調べて，水鉄砲の在庫があるかどうかを確かめるプログラム hasWatergun 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，水鉄砲は'watergun で表されるものとする．

おさらい: hasWatergun

例題

とある玩具店の目録がある．この目録を調べて，水鉄砲の在庫があるかどうかを確かめるプログラム hasWatergun 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，水鉄砲は'watergun で表されるものとする．

```
def hasWatergun:(List[Symbol]) => Boolean =
(inventory) =>
{inventory match{
  case Nil => false
  case e::lst => if (e=='watergun) true else hasWatergun(lst)
}}
```

函数の類似

hasDoll と hasWatergun はそれ自身の名前の違いを無視すれば，'doll と'watergun の違いしかない

hasDoll

```
def hasDoll:
(List[Symbol]) => Boolean =
(inventory) =>
{inventory match{
  case Nil => false
  case e::lst =>
    if (e== 'doll ) true
    else hasDoll(lst)
}}
```

hasWatergun

```
def hasWatergun:
(List[Symbol]) => Boolean =
(inventory) =>
{inventory match{
  case Nil => false
  case e::lst =>
    if (e== 'watergun ) true
    else hasWatergun(lst)
}}
```

抽象化: 類似函数を 1 つに

hasDoll と hasWatergun は，シンボルのリストの他にもうひとつシンボルを受け取るようにした以下の函数で抽象化できる

```
def hasItem:
(List[Symbol], Symbol) => Boolean =
(inventory, item) =>
{inventory match{
  case Nil => false
  case e::lst => if (e==item) true else hasItem(lst,item)
}}
```

抽象化: 類似函数を 1 つに

hasDoll と hasWatergun は，シンボルのリストの他にもうひとつシンボルを受け取るようにした以下の函数で抽象化できる

```
def hasItem:
(List[Symbol], Symbol) => Boolean =
(inventory, item) =>
{inventory match{
  case Nil => false
  case e::lst => if (e==item) true else hasItem(lst,item)
}}
```

任意の items∈List[Symbol] について以下が成立:

- hasItem(items,'doll) ≡ hasDoll(items)
- hasItem(items,'watergun) ≡ hasWatergun(items)

抽象化: 類似函数を 1 つに

hasDoll と hasWatergun は，シンボルのリストの他にもうひとつシンボルを受け取るようにした以下の函数で抽象化できる

```
def hasItem:
(List[Symbol], Symbol) => Boolean =
(inventory, item) =>
{inventory match{
  case Nil => false
  case e::lst => if (e==item) true else hasItem(lst,item)
}}
```

任意の items∈List[Symbol] について以下が成立:

- hasItem(items,'doll) ≡ hasDoll(items)
- hasItem(items,'watergun) ≡ hasWatergun(items)

hasItem は人形や水鉄砲以外にも車のあるなしやプラモデルのあるなしを調べることができる

目次

値の抽象化

函数の抽象化

型の抽象化

局所的な函数の定義

函数を生成する (返す) 函数

課題

below と above

問題 (below)

lst を整数のリスト，n を整数とする．lst に含まれる整数のうち n 未満であるような数だけからなる新しいリストを生成する函数 below を開発せよ．

below と above

問題 (below)

lst を整数のリスト，n を整数とする．lst に含まれる整数のうち n 未満であるような数だけからなる新しいリストを生成する函数 below を開発せよ．

```
def lt: (Int, Int) => Boolean =
(l,r) => {l < r}

def below: (List[Int],Int) => List[Int] =
(lst,n) => lst match{
  case Nil => Nil
  case e::rst => if(lt(e,n)) e::below(rst,n) else below(rst,n)
}
```

below と above

問題 (above)

lst を整数のリスト，n を整数とする．lst に含まれる整数のうち n より大きい数ばかりからなる新しいリストを生成する函数 above を開発せよ．

below と above

問題 (above)

lst を整数のリスト，n を整数とする．lst に含まれる整数のうち n より大きい数ばかりからなる新しいリストを生成する函数 above を開発せよ．

```
def gt: (Int, Int) => Boolean =
(l,r) => {l > r}

def above: (List[Int],Int) => List[Int] =
(lst,n) => lst match{
  case Nil => Nil
  case e::rst => if(gt(e,n)) e::above(rst,n) else above(rst,n)
}
```

函数の類似

below と above の間には if 式の条件式の中で lt を使うか gt を使うかの違いしかない

below

```
def below:
(List[Int],Int) => List[Int]
=
(lst,n) => lst match{
  case Nil => Nil
  case e::rst =>
    if(lt(e,n))
      e::below(rst,n)
    else below(rst,n)
}
```

above

```
def above:
(List[Int],Int) => List[Int]
=
(lst,n) => lst match{
  case Nil => Nil
  case e::rst =>
    if(gt(e,n))
      e::above(rst,n)
    else above(rst,n)
}
```

below と above の抽象化

hasDoll と hasWatergun を抽象化して hasItem を得たように，
below と above を抽象化して filter1 を得よう

below と above の抽象化

hasDoll と hasWatergun を抽象化して hasItem を得たように，below と above を抽象化して filter1 を得よう

- hasItem $\triangleq$ hasDoll$\uparrow$hasWatergun
  - 'doll と'watergun を抽象化した変数 item を導入
  - 引数として hasItem がシンボルのリストと item$\in$Symbol を受け取るように定義

below と above の抽象化

hasDoll と hasWatergun を抽象化して hasItem を得たように，below と above を抽象化して filter1 を得よう

- hasItem $\triangleq$ hasDoll↑hasWatergun
  - 'doll と'watergun を抽象化した変数 item を導入
  - 引数として hasItem がシンボルのリストと item∈Symbol を受け取るように定義
- filter1 $\triangleq$ below↑above
  - lt と gt を抽象化した変数 op を導入
  - 引数として filter1 が整数のリストと整数と op∈???を受け取るように定義

below と above の抽象化

hasDoll と hasWatergun を抽象化して hasItem を得たように，below と above を抽象化して filter1 を得よう

- hasItem $\triangleq$ hasDoll↑hasWatergun
  - 'doll と'watergun を抽象化した変数 item を導入
  - 引数として hasItem がシンボルのリストと item∈Symbol を受け取るように定義
- filter1 $\triangleq$ below↑above
  - lt と gt を抽象化した変数 op を導入
  - 引数として filter1 が整数のリストと整数と op∈???を受け取るように定義

Q. ???には何 (どんな型) を書けば良い？

- item の型は Symbol $\Leftarrow$ 'doll と'watergun の型が Symbol
- op の型は??? $\Leftarrow$ lt と gt の型が???

below と above の抽象化

hasDoll と hasWatergun を抽象化して hasItem を得たように，below と above を抽象化して filter1 を得よう

- hasItem $\triangleq$ hasDoll↑hasWatergun
  - 'doll と'watergun を抽象化した変数 item を導入
  - 引数として hasItem がシンボルのリストと item∈Symbol を受け取るように定義
- filter1 $\triangleq$ below↑above
  - lt と gt を抽象化した変数 op を導入
  - 引数として filter1 が整数のリストと整数と op∈???を受け取るように定義

Q. ???には何 (どんな型) を書けば良い？

- item の型は Symbol $\Leftarrow$ 'doll と'watergun の型が Symbol
- op の型は??? $\Leftarrow$ lt と gt の型が(Int,Int)=>Boolean

below と above の抽象化

hasDoll と hasWatergun を抽象化して hasItem を得たように，below と above を抽象化して filter1 を得よう

- hasItem $\triangleq$ hasDoll↑hasWatergun
  - 'doll と'watergun を抽象化した変数 item を導入
  - 引数として hasItem がシンボルのリストと item∈Symbol を受け取るように定義
- filter1 $\triangleq$ below↑above
  - lt と gt を抽象化した変数 op を導入
  - 引数として filter1 が整数のリストと整数と op∈???を受け取るように定義

Q. ???には何 (どんな型) を書けば良い？

- item の型は Symbol $\Leftarrow$ 'doll と'watergun の型が Symbol
- op の型は(Int, Int)=>Boolean
  $\Leftarrow$ lt と gt の型が(Int,Int)=>Boolean

filter1 ≜ below↑above

below と above を抽象化した函数 filter1 は below と above の引数に加えて (Int, Int)=>Boolean 型の変数を受け取るようにすることで以下のように定義できる

```
def filter1:
( (Int,Int)=>Boolean, List[Int], Int) => List[Int]
=
( op, lst, n) => lst match{
  case Nil => Nil
  case e::rst => if(op(e,n)) e::filter1(op,rst,n)
                 else filter1(op,rst,n)
}
```

filter1 ≜ below↑above

below と above を抽象化した函数 filter1 は below と above の引数に加えて (Int, Int)=>Boolean 型の変数を受け取るようにすることで以下のように定義できる

```
def filter1:
( (Int,Int)=>Boolean, List[Int], Int) => List[Int]
=
( op, lst, n) => lst match{
  case Nil => Nil
  case e::rst => if(op(e,n)) e::filter1(op,rst,n)
                 else filter1(op,rst,n)
}
```

任意の nums∈List[Int] と n∈Int について以下が成立:

- filter1(lt,nums,n) ≡ below(nums,n)
- filter1(gt,nums,n) ≡ above(nums,n)

函数が引数に!

filter1(lt,nums,n) の振舞

1. nums=Nil, n=5 のとき
   filter1(lt,Nil,5)

filter1(lt,nums,n) の振舞

1. nums=Nil, n=5 のとき

   filter1(lt,Nil,5)

   = // def. of filter1: case Nil => Nil

filter1(lt,nums,n) の振舞

1. nums=Nil, n=5 のとき

   filter1(lt,Nil,5)

   = // def. of filter1: case Nil => Nil

   Nil

filter1(lt,nums,n) の振舞

1. nums=Nil, n=5 のとき
   filter1(lt,Nil,5)
   = // def. of filter1: case Nil => Nil
   Nil
2. nums=List(4), n=5 のとき
   filter(lt,List(4),5)

filter1(lt,nums,n) の振舞

1. nums=Nil, n=5 のとき

   filter1(lt,Nil,5)

   = // def. of filter1: case Nil => Nil

   Nil

2. nums=List(4), n=5 のとき

   filter(lt,List(4),5)

   = // def. of filter1: case e::rst => ...

filter1(lt,nums,n) の振舞

1. nums=Nil, n=5 のとき

   filter1(lt,Nil,5)

   = // def. of filter1: case Nil => Nil

   Nil

2. nums=List(4), n=5 のとき

   filter(lt,List(4),5)

   = // def. of filter1: case e::rst => ...

   if(lt(4,5)) 4::filter1(lt,Nil,5) else filter1(lt,Nil,5)

filter1(lt,nums,n) の振舞

1. nums=Nil, n=5 のとき

   filter1(lt,Nil,5)

   = // def. of filter1: case Nil => Nil

   Nil

2. nums=List(4), n=5 のとき

   filter(lt,List(4),5)

   = // def. of filter1: case e::rst => ...

   if(lt(4,5)) 4::filter1(lt,Nil,5) else filter1(lt,Nil,5)

   = // lt(4,5) $\triangleq$ 4<5 = true

filter1(lt,nums,n) の振舞

1. nums=Nil, n=5 のとき

   filter1(lt,Nil,5)

   $=$ // def. of filter1: case Nil => Nil

   Nil

2. nums=List(4), n=5 のとき

   filter(lt,List(4),5)

   $=$ // def. of filter1: case e::rst => ...

   if(lt(4,5)) 4::filter1(lt,Nil,5) else filter1(lt,Nil,5)

   $=$ // lt(4,5) $\triangleq$ 4<5 $=$ true

   4::filter1(lt,Nil,5)

filter1(lt,nums,n) の振舞

1. nums=Nil, n=5 のとき

   filter1(lt,Nil,5)

   = // def. of filter1: case Nil => Nil

   Nil

2. nums=List(4), n=5 のとき

   filter(lt,List(4),5)

   = // def. of filter1: case e::rst => ...

   if(lt(4,5)) 4::filter1(lt,Nil,5) else filter1(lt,Nil,5)

   = // lt(4,5) $\triangleq$ 4<5 = true

   4::filter1(lt,Nil,5)

   = // filter1(lt,Nil,5)=Nil by case 1.

filter1(lt,nums,n) の振舞

1. nums=Nil, n=5 のとき
   filter1(lt,Nil,5)
   = // def. of filter1: case Nil => Nil
   Nil
2. nums=List(4), n=5 のとき
   filter(lt,List(4),5)
   = // def. of filter1: case e::rst => ...
   if(lt(4,5)) 4::filter1(lt,Nil,5) else filter1(lt,Nil,5)
   = // lt(4,5) $\triangleq$ 4<5 = true
   4::filter1(lt,Nil,5)
   = // filter1(lt,Nil,5)=Nil by case 1.
   4::Nil

filter1(lt,nums,n) の振舞

1. nums=Nil, n=5 のとき
   filter1(lt,Nil,5)
   = // def. of filter1: case Nil => Nil
   Nil
2. nums=List(4), n=5 のとき
   filter(lt,List(4),5)
   = // def. of filter1: case e::rst => ...
   if(lt(4,5)) 4::filter1(lt,Nil,5) else filter1(lt,Nil,5)
   = // lt(4,5) $\triangleq$ 4<5 = true
   4::filter1(lt,Nil,5)
   = // filter1(lt,Nil,5)=Nil by case 1.
   4::Nil

いずれも below と同じ結果を得ている

filter1 の効能

filter1 によって実現できる函数は below と above だけではない

filter1 の効能

filter1 によって実現できる函数は below と above だけではない

- n と等しい整数だけのリスト: filter1(equal,nums,n) with def equal:(Int,Int)=>Boolean=(l,r)=>l==r

filter1 の効能

filter1 によって実現できる函数は below と above だけではない

- n と等しい整数だけのリスト: filter1(equal,nums,n)
  with def equal:(Int,Int)=>Boolean=(l,r)=>l==r
- n 以下の整数だけのリスト: filter1(leq,nums,n)
  with def leq:(Int,Int)=>Boolean=(l,r)=>l<=r

filter1 の効能

filter1 によって実現できる函数は below と above だけではない

- n と等しい整数だけのリスト: filter1(equal,nums,n)
  with def equal:(Int,Int)=>Boolean=(l,r)=>l==r
- n 以下の整数だけのリスト: filter1(leq,nums,n)
  with def leq:(Int,Int)=>Boolean=(l,r)=>l<=r
- n 以上の整数だけのリスト: filter1(geq,nums,n)
  with def geq:(Int,Int)=>Boolean=(l,r)=>l>=r

filter1 の効能

filter1 によって実現できる函数は below と above だけではない

- n と等しい整数だけのリスト: filter1(equal,nums,n)
  with def equal:(Int,Int)=>Boolean=(l,r)=>l==r
- n 以下の整数だけのリスト: filter1(leq,nums,n)
  with def leq:(Int,Int)=>Boolean=(l,r)=>l<=r
- n 以上の整数だけのリスト: filter1(geq,nums,n)
  with def geq:(Int,Int)=>Boolean=(l,r)=>l>=r
- 自乗した値が n 以上の整数だけのリスト:
  filter1(sqgeq,nums,n)
  with def sqgeq:(Int,Int)=>Boolean=(l,r)=>l*l>=r

抽象化の利点

- より広く使える函数が手に入る
- バグの修正が容易になる

```
def below: (List[Int],Int)=>List[Int] =
(lst,n) => {filter1(lt,lst,n)}
def above: (List[Int],Int)=>List[Int] =
(lst,n) => {filter1(gt,lst,n)}
```

と定義しておくことで，`below` や `above` に誤りが見つかった時，`filter1` を修正するだけで両方が修正できる

目次

値の抽象化

函数の抽象化

型の抽象化

局所的な函数の定義

函数を生成する (返す) 函数

課題

below と belowBudget

問題 (belowBudget)

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，予算の上限を越えないメニューを取り出す函数 blowBudget 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．

below と belowBudget

問題 (belowBudget)

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，予算の上限を越えないメニューを取り出す函数 blowBudget 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．

```
def belowBudget: (List[Plate],Int) => List[Plate]
= (menus,n) => menus match{
  case Nil => Nil
  case e::rst => if(leqP(e,n)) e::belowBudget(rst,n)
                 else belowBudget(rst,n)
}
```

函数の類似

below と belowBudget には，それ自身と引数の名前を無視すると，lt と leqP，Int と Plate の違いがある

below

```
def below:
(List[Int],Int)=>List[Int]
=
(lst,n) => lst match{
  case Nil => Nil
  case e::rst =>
    if(lt(e,n))
      e::below(rst,n)
    else below(rst,n)
}
```

belowBudget

```
def belowBudget:
(List[Plate],Int)=>List[Plate]
=
(menus,n) => menus match{
  case Nil => Nil
  case e::rst =>
    if(leqP(e,n))
      e::belowBudget(rst,n)
    else belowBudget(rst,n)
}
```

filterX≜below↑belowBudget

below と above を抽象化して filter1 を得たように，
below∈(List[Int],Int)=>List[Int] と
belowBudget∈(List[Plate],Int)=>List[Plate] を抽象化して
filterX を得よう

filterX≜below↑belowBudget

below と above を抽象化して filter1 を得たように，
below∈(List[Int],Int)=>List[Int] と
belowBudget∈(List[Plate],Int)=>List[Plate] を抽象化して
filterX を得よう

- filter1 ≜ below↑above
  - lt と gt を抽象化した変数 op を導入
  - 引数として filter1 が整数のリストと整数と
    op∈(Int,Int)=>Boolean を受け取るように定義

filterX≜below↑belowBudget

below と above を抽象化して filter1 を得たように，
below∈(List[Int],Int)=>List[Int] と
belowBudget∈(List[Plate],Int)=>List[Plate] を抽象化して
filterX を得よう

- filter1 ≜ below↑above
  - lt と gt を抽象化した変数 op を導入
  - 引数として filter1 が整数のリストと整数と
    op∈(Int,Int)=>Boolean を受け取るように定義
- filterX ≜ below↑belowBudget
  - leq と leqP を抽象化した変数 op を導入
  - Int と Plate を抽象化した変数 T を導入
  - 引数として filterX が T のリストと整数と
    op∈(T,Int)=>Boolean を受け取るように定義

filterX≜below↑belowBudget

below と above を抽象化して filter1 を得たように，
below∈(List[Int],Int)=>List[Int] と
belowBudget∈(List[Plate],Int)=>List[Plate] を抽象化して
filterX を得よう

- filter1 ≜ below↑above
  - lt と gt を抽象化した変数 op を導入
  - 引数として filter1 が整数のリストと整数と
    op∈(Int,Int)=>Boolean を受け取るように定義
- filterX ≜ below↑belowBudget
  - leq と leqP を抽象化した変数 op を導入
  - Int と Plate を抽象化した変数 T を導入
  - 引数として filterX が T のリストと整数と
    op∈(T,Int)=>Boolean を受け取るように定義

⇒filterX:((T,Int)=>Boolean,List[T],Int)=>???

filterX≜below↑belowBudget

below と above を抽象化して filter1 を得たように，
below∈(List[Int],Int)=>List[Int] と
belowBudget∈(List[Plate],Int)=>List[Plate] を抽象化して
filterX を得よう

- filter1 ≜ below↑above
  - lt と gt を抽象化した変数 op を導入
  - 引数として filter1 が整数のリストと整数と
    op∈(Int,Int)=>Boolean を受け取るように定義
- filterX ≜ below↑belowBudget
  - leq と leqP を抽象化した変数 op を導入
  - Int と Plate を抽象化した変数 T を導入
  - 引数として filterX が T のリストと整数と
    op∈(T,Int)=>Boolean を受け取るように定義

⇒filterX:((T,Int)=>Boolean,List[T],Int)=>List[T]

filterX≜below↑belowBudget

below と above を抽象化して filter1 を得たように，
below∈(List[Int],Int)=>List[Int] と
belowBudget∈(List[Plate],Int)=>List[Plate] を抽象化して
filterX を得よう

- filter1 ≜ below↑above
  - lt と gt を抽象化した変数 op を導入
  - 引数として filter1 が整数のリストと整数と
    op∈(Int,Int)=>Boolean を受け取るように定義
- filterX ≜ below↑belowBudget
  - leq と leqP を抽象化した変数 op を導入
  - Int と Plate を抽象化した変数 T を導入
  - 引数として filterX が T のリストと整数と
    op∈(T,Int)=>Boolean を受け取るように定義

⇒filterX[T]:((T,Int)=>Boolean,List[T],Int)=>List[T]

型を受け取る函数

- 普通の函数: 値を受け取って値を返す

```
def lt: (Int,Int) => Boolean = (l,r) => l < r
```

- 型を受け取る函数: 型と値を受け取って値を返す
  - 例: 恒等函数

```
def id[X]: (X)=>X = (x) => x
```

この函数は型 X と X の値を受け取って X の値を返す．id を使う時は型とその値を渡す．

```
scala> id[Int](3)
res0: Int = 3
```

```
scala> id[Symbol]('a)
res0: Symbol = 'a
```

ただし，引数から渡されるべき型が明らかな場合は型を明示的に渡さなくても良い．

```
scala> id(3) //3:Int
res0: Int = 3
```

```
scala> id('a) //'a:Symbol
res0: Symbol = 'a
```

filterX≜below↑belowBudget

below と belowBudget を抽象化した函数 filterX は below と belowBudget の引数に加えて型 T と (T, Int)=>Boolean 型の値 op を受け取るようにすることで以下のように定義できる:

```
def filterX[T]: ((T,Int)=>Boolean,List[T],Int) => List[T] =
(comp, lst, n) => lst match{
  case Nil => Nil
  case e::rst => if(comp(e,n)) e::filterX[T](comp,rst,n)
                 else filterX[T](comp,rst,n)
}
```

filterX≙below↑belowBudget

below と belowBudget を抽象化した函数 filterX は below と belowBudget の引数に加えて型 T と (T, Int)=>Boolean 型の値 op を受け取るようにすることで以下のように定義できる:

```
def filterX[T]: ((T,Int)=>Boolean,List[T],Int) => List[T] =
(comp, lst, n) => lst match{
  case Nil => Nil
  case e::rst => if(comp(e,n)) e::filterX[T](comp,rst,n)
                 else filterX[T](comp,rst,n)
}
```

型変数 T に適切な型を与えることで以下が成立:

- filterX[Int](lt,nums,n) ≡ below(nums,n)
- filterX[Plate](leqP,menus,n) ≡ belowBudget(menus,n)

複数の型を受け取る函数

*n*個の型を受け取る函数 f の定義

def f[T_1,$\cdots$,T_n]: 函数の型 = (引数) => {函数の本体}

例: 更に一般化した filter

```
def filter[A,B]: ((A,B)=>Boolean,List[A],B) => List[A] =
(comp, lst, n) => lst match{
  case Nil => Nil
  case e::rst => if(comp(e,n)) e::filter(comp,rst,n)
                 else filter(comp,rst,n)
}
```

型変数 A,B に適切な型を与えることで以下が成立:

- filter[Int,Int](lt,lst,n) ≡ filterX[Int](lt,nums,n)
- filter[Plate,Int](leqP,lst,n) ≡ filterX[Plate](leqP,menus,n)

目次

値の抽象化

函数の抽象化

型の抽象化

局所的な函数の定義

函数を生成する (返す) 函数

課題

函数の中で函数を定義

補助函数や変数を用意する時，1 つの函数の中だけでしか (名前の衝突を避けるために) 使えないようにしておきたくなることがある．このような時は補助函数を，それを使う函数の中で定義してしまうと良い．

例題

問題
楽器の演奏者の名簿 (lst∈List[Player]) と演奏者の名前 (n∈Symbol) が与えられる.

```
case class Player(name:Symbol, instrument:Symbol, year:Int)
```

一人の人が複数回名簿の中に現れることがある. lst に登録されている演奏者 n の演奏楽器のうち最新のものを出力するプログラム lastInstrument を開発せよ.

例題

問題

楽器の演奏者の名簿 (lst∈List[Player]) と演奏者の名前 (n∈Symbol) が与えられる．

```
case class Player(name:Symbol, instrument:Symbol, year:Int)
```

一人の人が複数回名簿の中に現れることがある．lst に登録されている演奏者 n の演奏楽器のうち最新のものを出力するプログラム lastInstrument を開発せよ．

解法の概要

1. lst 中の Player のうち name==n となるものだけのリストを作る

例題

問題

楽器の演奏者の名簿 (lst∈List[Player]) と演奏者の名前 (n∈Symbol) が与えられる.

```
case class Player(name:Symbol, instrument:Symbol, year:Int)
```

一人の人が複数回名簿の中に現れることがある. lst に登録されている演奏者 n の演奏楽器のうち最新のものを出力するプログラム lastInstrument を開発せよ.

解法の概要

1. lst 中の Player のうち name==n となるものだけのリストを作る
2. 1. で作ったリスト中の Player を year に関して降順に並び替えたリストを作る

例題

問題

楽器の演奏者の名簿 (lst∈List[Player]) と演奏者の名前 (n∈Symbol) が与えられる．

```
case class Player(name:Symbol, instrument:Symbol, year:Int)
```

一人の人が複数回名簿の中に現れることがある．lst に登録されている演奏者 n の演奏楽器のうち最新のものを出力するプログラム lastInstrument を開発せよ．

解法の概要

1. lst 中の Player のうち name==n となるものだけのリストを作る
2. 1. で作ったリスト中の Player を year に関して降順に並び替えたリストを作る
3. 2. で作ったリストの先頭要素を取り出して，instrument を取り出す

lastInstrument

解法の概要

1. lst 中の Player のうち name==n となるものだけのリストを作る
2. 1. で作ったリスト中の Player を year に関して降順に並び替えたリストを作る
3. 2. で作ったリストの先頭要素を取り出して，instrument を取り出す

```
def lastInstrument: (List[Player], Symbol) => Symbol =
(lst,n) => {
  head(sort(gtP,filter(eqP,lst))) match{
    case Player(name,instrument,year) => instrument
  }
}
```

lastInstrument

解法の概要

1. lst 中の Player のうち name==n となるものだけのリストを作る
2. 1. で作ったリスト中の Player を year に関して降順に並び替えたリストを作る
3. 2. で作ったリストの先頭要素を取り出して，instrument を取り出す

```
def filter[X]: ((X)=>Boolean, List[X]) => List[X] =
(p,lst) =>
/**
 * 目的: lstの要素eでp(e)が真になるものだけからなるリストを作る
 */
{lst match{
  case Nil => Nil
  case e::rst => if(p(e)) e::filter(p,rst) else filter(p,rst)
}}
```

lastInstrument

解法の概要

1. lst 中の Player のうち name==n となるものだけのリストを作る
2. 1. で作ったリスト中の Player を year に関して降順に並び替えたリストを作る
3. 2. で作ったリストの先頭要素を取り出して，instrument を取り出す

```
def head[X]: List[X] => X =
(lst) =>
/**
 * 目的: lstの先頭要素を取り出す
 */
{lst match{
  case e::rst => e
}}
```

lastInstrument

解法の概要

1. lst 中の Player のうち name==n となるものだけのリストを作る
2. 1. で作ったリスト中の Player を year に関して降順に並び替えたリストを作る
3. 2. で作ったリストの先頭要素を取り出して，instrument を取り出す

```
def lastInstrument: (List[Player], Symbol) => Symbol =
(lst,n) => {
  head(sort(gtP,filter(eqP,lst))) match{
    case Player(name,instrument,year) => instrument
  }
}
```

lastInstrument

解法の概要

1. lst 中の Player のうち name==n となるものだけのリストを作る
2. 1. で作ったリスト中の Player を year に関して降順に並び替えたリストを作る
3. 2. で作ったリストの先頭要素を取り出して，instrument を取り出す

```
def lastInstrument: (List[Player], Symbol) => Symbol =
(lst,n) => {
  head(sort( gtP ,filter( eqP ,lst))) match{
    case Player(name,instrument,year) => instrument
  }}
```

Q. eqP や gtP はどこに定義するか？

lastInstrument

解法の概要

1. lst 中の Player のうち name==n となるものだけのリストを作る
2. 1. で作ったリスト中の Player を year に関して降順に並び替えたリストを作る
3. 2. で作ったリストの先頭要素を取り出して，instrument を取り出す

```
def lastInstrument: (List[Player], Symbol) => Symbol =
(lst,n) => {
  head(sort( gtP ,filter( eqP ,lst))) match{
    case Player(name,instrument,year) => instrument
  }}
```

Q. eqP や gtP はどこに定義するか？A. lastInstrument 内部

lastInstrument

```
def lastInstrument: (List[Player], Symbol) => Symbol =
(lst,n) => {
  def eqP:(Player) => Boolean =
  (p) => p match{
    case Player(name,instr,year) => n == name
  }
  def gtP: (Player, Player) => Boolean =
  (p1,p2) => (p1,p2) match{
    case (Player(n1,i1,y1),Player(n2,i2,y2)) => y1 > y2
  }
  head(sort(gtP,filter(eqP,lst))) match{
    case Player(name,instrument,year) => instrument
  }}
```

eqP は外部に定義できないことに注意！

目次

値の抽象化

函数の抽象化

型の抽象化

局所的な函数の定義

函数を生成する (返す) 函数

課題

函数を返す函数: 簡単な例

```
def id: (List[Int]) => List[Int] = (x) => x

def tail: (List[Int]) => List[Int]
= (lst) => lst match{
  case Nil => Nil
  case e::rst => rst
}

def h: List[Int] => ???
= (lst) => lst match{
  case Nil => id
  case e::rst => tail
}
```

函数を返す函数: 簡単な例

```
def id: (List[Int]) => List[Int] = (x) => x

def tail: (List[Int]) => List[Int]
= (lst) => lst match{
  case Nil => Nil
  case e::rst => rst
}

def h: List[Int] => ???
= (lst) => lst match{
  case Nil => id
  case e::rst => tail
}
```

```
id(Nil) == Nil
id(List(1,2)) == List(1,2)
```

函数を返す函数: 簡単な例

```
def id: (List[Int]) => List[Int] = (x) => x

def tail: (List[Int]) => List[Int]
= (lst) => lst match{
  case Nil => Nil
  case e::rst => rst
}

def h: List[Int] => ???
= (lst) => lst match{
  case Nil => id
  case e::rst => tail
}
```

```
id(Nil) == Nil
id(List(1,2)) == List(1,2)
```

```
tail(Nil) == Nil
tail(List(1,2)) == List(2)
```

函数を返す函数: 簡単な例

```
def id: (List[Int]) => List[Int] = (x) => x

def tail: (List[Int]) => List[Int]
= (lst) => lst match{
  case Nil => Nil
  case e::rst => rst
}

def h: List[Int] => ???
= (lst) => lst match{
  case Nil => id
  case e::rst => tail
}
```

```
id(Nil) == Nil
id(List(1,2)) == List(1,2)
```

```
tail(Nil) == Nil
tail(List(1,2)) == List(2)
```

```
h(Nil) == id
h(List(1,2)) == tail
```

函数を返す函数: 簡単な例

```
def id: (List[Int]) => List[Int] = (x) => x

def tail: (List[Int]) => List[Int]
= (lst) => lst match{
  case Nil => Nil
  case e::rst => rst
}

def h: List[Int] => ???
= (lst) => lst match{
  case Nil => id
  case e::rst => tail
}
```

```
id(Nil) == Nil
id(List(1,2)) == List(1,2)
```

```
tail(Nil) == Nil
tail(List(1,2)) == List(2)
```

```
h(Nil) == id
h(List(1,2)) == tail
```

Q. ??? に来るべき型は？

函数を返す函数: 簡単な例

```
def id: (List[Int]) => List[Int] = (x) => x

def tail: (List[Int]) => List[Int]
= (lst) => lst match{
  case Nil => Nil
  case e::rst => rst
}

def h: List[Int] => ???
= (lst) => lst match{
  case Nil => id
  case e::rst => tail
}
```

```
id(Nil) == Nil
id(List(1,2)) == List(1,2)
```

```
tail(Nil) == Nil
tail(List(1,2)) == List(2)
```

```
h(Nil) == id
h(List(1,2)) == tail
```

Q. ??? に来るべき型は？　　A. `List[Int] => List[Int]`

函数を返す函数: 段階的な add

問題

x, y∈Int が与えられた時 x+y を計算する函数 add を定義せよ.

函数を返す函数: 段階的な add

問題

x, y∈Int が与えられた時 x+y を計算する函数 add を定義せよ.

素直な定義

```
def add:(Int,Int) => Int =
(x,y) => {x + y}
```

函数を返す函数: 段階的な add

問題

x, y∈Int が与えられた時 x+y を計算する函数 add を定義せよ.

素直な定義

```
def add:(Int,Int) => Int =
(x,y) => {x + y}
```

ちょっとひねくれた定義

```
def add: (Int)=>??? =
(x) => {
  def xaddr: (Int)=>Int = (y) => {x + y}
  xaddr
}
```

函数を返す函数: 段階的な add

問題

x, y∈Int が与えられた時 x+y を計算する函数 add を定義せよ.

素直な定義

```
def add:(Int,Int) => Int =
(x,y) => {x + y}
```

ちょっとひねくれた定義

```
def add: (Int)=>(Int) => Int =
(x) => {
  def xaddr: (Int)=>Int = (y) => {x + y}
  xaddr
}
```

add の振舞

```
def add: (Int)=>(Int) => Int =
(x) => {
  def xaddr: (Int)=>Int = (y) => {x + y}
  xaddr
}

 add(5)
```

add の振舞

```
def add: (Int)=>(Int) => Int =
(x) => {
  def xaddr: (Int)=>Int = (y) => {x + y}
  xaddr
}
```

add(5)
= //add の定義を展開
xaddr // def xaddr: (Int)=>Int = (y) => {5 + y}

add の振舞

```
def add: (Int)=>(Int) => Int =
(x) => {
  def xaddr: (Int)=>Int = (y) => {x + y}
  xaddr
}
```

add(5)
= //add の定義を展開
xaddr // def xaddr: (Int)=>Int = (y) => {5 + y}
= //xaddr の定義を展開
(y) => {5+y}

計算結果の函数が x=5 を記憶していることに注目

add の振舞

```
def add: (Int)=>(Int) => Int =
(x) => {
  def xaddr: (Int)=>Int = (y) => {x + y}
  xaddr
}
```

add(5)(6)
= //add の定義を展開
xaddr(6) // def xaddr: (Int)=>Int = (y) => {5 + y}
= //xaddr の定義を展開
5+6
=
11

函数を返す函数を使う利点

函数を返す函数を使うことでプログラムの再利用性が上がる.
map を以下の函数とする:

```
def map[A,B]: ((A)=>B, List[A])=>List[B] =
(f,lst) => lst match{
  case Nil => Nil
  case e::rst => f(e)::map(f,rst)
}
```

- List[Int] の各要素に 1 を加える:
  map(add(1),List(1,2,3))≡List(2,3,4)
- List[Int] の各要素に 3 を加える:
  map(add(3),List(1,2,3))≡List(4,5,6)

函数を返す函数を使う利点

函数を返す函数を使うことでプログラムの再利用性が上がる. map を以下の函数とする:

```
def map[A,B]: ((A)=>B, List[A])=>List[B] =
(f,lst) => lst match{
  case Nil => Nil
  case e::rst => f(e)::map(f,rst)
}
```

もし map も函数を返す函数として定義できたら (引数をバラバラに渡せるようになったら) 以下のようなプログラムが簡単にかけるようになる

- List[List[Int]] の各要素に 1 を加える:
  map(map(add(1)))(List(List(1,2,3),List(4,5,6)))≡
  List(List(2,3,4),List(5,6,7))

map の引数をバラす

バラし方は以下の通り:

1. map の第一引数だけを受け取る函数の名前と型を定義する (newmap とする). 引数の変数名は map と同じにする.
2. newmap の本体に第一引数を削った map の定義を埋め込み, map を返すようにする
3. map 内の再帰的な map の呼び出しを newmap の呼び出しに変える

map の引数をバラす

バラし方は以下の通り:

1. map の第一引数だけを受け取る函数の名前と型を定義する (newmap とする). 引数の変数名は map と同じにする.
2. newmap の本体に第一引数を削った map の定義を埋め込み, map を返すようにする
3. map 内の再帰的な map の呼び出しを newmap の呼び出しに変える

```
def map[A,B]: ((A)=>B, List[A])=>List[B] =
(f,lst) => lst match{
  case Nil => Nil
  case e::rst => f(e)::map(f,rst)
}
```

map の引数をバラす

バラし方は以下の通り:

1. map の第一引数だけを受け取る函数の名前と型を定義する (newmap とする). 引数の変数名は map と同じにする.
2. newmap の本体に第一引数を削った map の定義を埋め込み, map を返すようにする
3. map 内の再帰的な map の呼び出しを newmap の呼び出しに変える

```
def newmap[A,B]: (A=>B)=>(List[A])=>List[B] =
(f) => {
  ...
}
```

map の引数をバラす

バラし方は以下の通り:

1. map の第一引数だけを受け取る函数の名前と型を定義する (newmap とする). 引数の変数名は map と同じにする.
2. newmap の本体に第一引数を削った map の定義を埋め込み, map を返すようにする
3. map 内の再帰的な map の呼び出しを newmap の呼び出しに変える

```
def newmap[A,B]: (A=>B)=>(List[A])=>List[B] =
(f) => {
  def map: (List[A])=>List[B] = (lst)=>lst match{
    case Nil => Nil
    case e::rst => f(e)::map(f,rst)
  }
  map
}
```

map の引数をバラす

バラし方は以下の通り:

1. map の第一引数だけを受け取る函数の名前と型を定義する (newmap とする). 引数の変数名は map と同じにする.
2. newmap の本体に第一引数を削った map の定義を埋め込み, map を返すようにする
3. map 内の再帰的な map の呼び出しを newmap の呼び出しに変える

```
def newmap[A,B]: (A=>B)=>(List[A])=>List[B] =
(f) => {
  def map: (List[A])=>List[B] = (lst)=>lst match{
    case Nil => Nil
    case e::rst => f(e)::newmap(f)(rst)
  }
  map
}
```

add 中の xaddr は必要ない

add の定義

```
def add: (Int)=>(Int) => Int =
(x) => {
  def xaddr: (Int)=>Int = (y) => {x + y}
  xaddr
}
```

add は xaddr を返しているだけである．xaddr の定義を add の中で展開すると以下を得る:

```
def add:(Int)=>(Int)=>Int =
(x) => { (y) => {x+y} }
```

Scala では (変数 $_1$, $\cdots$, 変数 $_n$)=>{式} が無名の函数を定義する記法で，xaddr を使わずに書いても (展開しただけなので) 意味は同じである．

目次

値の抽象化

函数の抽象化

型の抽象化

局所的な函数の定義

函数を生成する (返す) 函数

課題

課題

以降の課題をとき，05.zip を OCW/OCW-i を使って提出せよ．
提出締め切りは5 月 19 日 15:00(JST)とする．

課題 1: flatten.scala

リストのリストを以下のようにリストに均す函数 `flatten` を開発せよ:

$$\texttt{flatten(List(List(}\overline{e_1}\texttt{),}\cdots\texttt{,List(}\overline{e_n}\texttt{)))} = \texttt{List(}\overline{e_1}\texttt{,}\cdots\texttt{,}\overline{e_n}\texttt{)}$$

ただし $\overline{e_1},\cdots,\overline{e_n}$の各要素の型は同じとする

例

- `flatten(Nil)` = `Nil`
- `flatten(List(List(1,2,3),List(4,5,6)))` = `List(1,2,3,4,5,6)`
- `flatten(List(List('a,'b),List('c,'d)))` = `List('a,'b,'c,'d)`

課題 2: filter.scala

X を任意の型とする．函数 p∈X=>Boolean とリスト lst∈List[X] が与えられた時，lst の要素 e のうちで p(e) が真になるものばかりを拾い集めて作ったリストを返す函数 filter を開発せよ．filter は p と lst を別々に受け取るものとする．

例

- `filter(isEven)(List(1,2,3)) == List(2)`
- `filter(isOdd)(List(1,2,3)) == List(1,3)`
- ```
  filter(isSingleton)
    (newmap(filter(isEven))(List(List(1,2,3),List(4,5,6))))
  == List(List(2))
  ```
  ```

課題 2: 補助函数たち

```
def isEven: Int => Boolean =
(n) =>{(n & 1) == 0}
```

```
def isOdd: Int => Boolean =
(n) =>{!isEven(n)}
```

```
def isSingleton[X]: (List[X]) => Boolean =
(lst) => lst match{
  case Nil => false
  case e::Nil => true
  case e::rst => false
}
```

課題 3: foldr.scala

X，Y を任意の型とする．

- (X)=>(Y)=>Y となる函数 f
- Y 型の値 n
- List[X] 型のリスト lst

がこの順に別々に与えられた時，lst の

- :: を f に
- Nil を n に

置き換える函数 foldr を開発せよ．

ヒント

:: が右結合であることに注意．つまり，

foldr(f)(n)(x::y::···::Nil) ≡ f(x,f(y,f(··· ,f(·,n)···)))

余談: foldr の面白さ

- f≜+(≡add)，n≜0 のとき，foldr(f)(n)(lst) は lst の総和を計算する
  - foldr(add)(n)(1::2::3::Nil)≡1+(2+(3+0))=6
- f≜*(≡mul)，n≜1 のとき，foldr(f)(n)(lst) は lst の要素を全て掛けあわせた値を計算する
  - foldr(mul)(n)(1::2::3::Nil)≡1*(2*(3*1))=6

  ただし mul の定義は以下:

```
def mul: (Int) => (Int) => Int =
(x) => {(y) => { x*y }}
```

- f≜&&(≡and)，n≜true のとき，foldr(f)(n)(lst) は lst の要素の総論理積を計算する
  - foldr(and)(true)

    (true::false::Nil)≡true&&(false&&true)=false