計算機科学概論演習

#02

青谷知幸

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

April 14, 2016

目次

目次

条件の導入

現実の問題には様々な条件指定が含まれる

例題
株式会社 XYZ は時給 1200 円を被雇用者に支払っている．被雇用者は週に 20 時間から 65 時間働く．労働時間が与えられた時，その労働時間が 20 時間から 65 時間の間に収まる場合に限って被雇用者の給与を計算するプログラム `wage` を作りなさい．

条件の導入

現実の問題には様々な条件指定が含まれる

例題
株式会社 XYZ は時給 1200 円を被雇用者に支払っている．被雇用者は週に 20 時間から 65 時間働く．労働時間が与えられた時，その労働時間が 20 時間から 65 時間の間に収まる場合に限って被雇用者の給与を計算するプログラム wage を作りなさい．

$$\mathtt{wage}(h) = 1200h \text{ただし} 20 \leq h \leq 65$$

条件の導入

現実の問題には様々な条件指定が含まれる

例題
株式会社 XYZ は時給 1200 円を被雇用者に支払っている．被雇用者は週に 20 時間から 65 時間働く．労働時間が与えられた時，その労働時間が 20 時間から 65 時間の間に収まる場合に限って被雇用者の給与を計算するプログラム wage を作りなさい．

$$\texttt{wage}(h) = 1200h \text{ただし} 20 \leq h \leq 65$$

どうやって条件 (ただし$20 \leq h \leq 65$) をプログラムで表すか？

整理: 条件記述に必要な道具

- 2 つの数値を比較する演算子: $<, \leq, =, \neq \geq, >$
- 2 つの比較の結果を結合する演算子: $\wedge, \vee$
- 真および偽を表す値/記号 (真偽値): `true`, `false`
- 真偽を反転させる演算子: $\neg$

例

- $3 < 4$ ($\equiv$ `true`)
- $\neg(3 < 4)$ ($\equiv \neg$`true` $\equiv$ `false`)
- $3 < 4 < 5$ ($\equiv$ `true`)
- $3 < 4 \wedge 5 > 6$ ($\equiv$ `true` $\wedge$ `false` $\equiv$ `false`)
- $3 < 4 \vee 5 > 6$ ($\equiv$ `true` $\vee$ `false` $\equiv$ `true`)

条件記述に必要な道具 in Scala

- 2 つの数値を比較する演算子: `<`, `<=`, `==`, `!=`, `>=`, `>`
- 2 つの比較の結果を結合する演算子: `&&`, `||`
- 真および偽を表す値 (真偽値): `true`, `false`
- `true` および `false` の型: `Boolean`
- 真偽を反転させる演算子: `!`

数式との対応

数式	Scala プログラム	実行結果
$3 = 3$	`3 == 4`	`false`
$3 < 4$	`3 < 4`	`true`
$\neg(3 < 4)$	`!(3 < 4)`	`false`
$3 < 4 < 5$	`3 < 4 && 4 < 5`	`true`
$3 < 4 \wedge 5 > 6$	`3 < 4 && 5 > 6`	`false`
$3 < 4 \vee 5 > 6$	`3 < 4 || 5 > 6`	`true`

練習問題

式	実行結果
`4 > 3 && 10 <= 100`	
`4 > 3 || 10 == 100`	
`! 2 == 3`	

練習問題

式	実行結果
`4 > 3 && 10 <= 100`	`true`
`4 > 3 || 10 == 100`	
`! 2 == 3`	

練習問題

式	実行結果
`4 > 3 && 10 <= 100`	`true`
`4 > 3 || 10 == 100`	`true`
`! 2 == 3`	

練習問題

式	実行結果
`4 > 3 && 10 <= 100`	`true`
`4 > 3 || 10 == 100`	`true`
`! 2 == 3`	*error*

練習問題

式	実行結果
`4 > 3 && 10 <= 100`	`true`
`4 > 3 || 10 == 100`	`true`
`! 2 == 3`	*error*
`!(2==3)`	`true`

練習問題

式	実行結果
`4 > 3 && 10 <= 100`	`true`
`4 > 3 || 10 == 100`	`true`
`! 2 == 3`	*error*
`!(2==3)`	`true`

!2==3 の実行

```
scala> ! 2==3
<console>:8: error: value unary_! is not a member of Int
              ! 2==3
              ^
```

(!2) == 3 と解釈されてしまうせいで，! を Int 型の値である 2 に適用しようとしている

条件を検査する函数 (述語)

```
def is5:
(Number) => Boolean
=
(n) =>
/**
 * 目的: nが5と等しいかどうかを判定
 * 例: is5(5) は true, is5(4) は false
 */
{n == 5}
```

Number は Int, Long, Float, Double の全てを部分集合として含む型

条件を検査する函数 (述語)

```
def isBetween5and6:
(Number) => Boolean
=
(n) =>
/**
 * 目的: nが範囲(5,6)に収まるかどうかを判定
 * 例:
 *  isBetween5and6(3.2) は false
 *  isBetween5and6(5.6) は true
 */
{5 < n && n < 6}
```

(5,6) は開区間「5 より大きく 6 より小さい」を表す (cf. [5,6] は閉区間「5 以上 6 以下」を表す)

条件を検査する函数 (述語)

```
def isBetween5and6orOver10:
(Number) => Boolean
=
(n) =>
/**
 * 目的: nが範囲(5,6)または[10,∞]に収まるかどうかを判定
 * 例:
 *  isBetween5and6Over10(10.1) は true
 *  isBetween5and6(5.6) は true
 *  isBetween5and6(6) は false
 */
{isBetween5and6(n) || 10<=n}
```

条件を検査する函数 (述語)

```
def equation1:
(Number) => Boolean
=
(n) =>
/**
 * 目的: nが$x^2 + 2x + 1 = 0$の解であるかどうかを判定
 * 例:
 *  equation1(-1) は true
 *  equation1(1) は false
 */
{n*n+2*n+1==0}
```

もうひとつの道具: 条件式

例題
株式会社 XYZ は時給 1200 円を被雇用者に支払っている．被雇用者は週に 20 時間から 65 時間働く．労働時間が与えられた時，その労働時間が 20 時間から 65 時間の間に収まる場合に限って被雇用者の給与を計算するプログラム wage を作りなさい.

$$\mathtt{wage}(h) = 1200h \text{ただし} 20 \leq h \leq 65$$

どうやって条件 (ただし$20 \leq h \leq 65$) をプログラムで表すか？
$\Rightarrow$ 条件式を使う

条件式の形

e_nを任意の式とする．条件式は以下のいずれかの形をとる:

- `if(`e_1`){`e_2`}`
- `if(`e_1`){`e_2`} else {`e_3`}`

ただし e_2および e_3の周りの `{}` は省略可

条件式の形

e_nを任意の式とする．条件式は以下のいずれかの形をとる:

- `if(`e_1`){`e_2`}`
  もし e_1を実行した結果が真ならば e_2を実行
- `if(`e_1`){`e_2`} else {`e_3`}`

ただし e_2および e_3の周りの `{}` は省略可

条件式の形

e_nを任意の式とする．条件式は以下のいずれかの形をとる:

- `if(`e_1`){`e_2`}`
  もし e_1を実行した結果が真ならば e_2を実行
- `if(`e_1`){`e_2`} else {`e_3`}`
  もし e_1を実行した結果が真ならば e_2を実行，そうでなければ e_3を実行

ただし e_2および e_3の周りの `{}` は省略可

条件式の形

e_nを任意の式とする．条件式は以下のいずれかの形をとる:

- `if(`e_1`){`e_2`}`
  もし e_1を実行した結果が真ならば e_2を実行
- `if(`e_1`){`e_2`} else {`e_3`}`
  もし e_1を実行した結果が真ならば e_2を実行，そうでなければ e_3を実行

ただし e_2および e_3の周りの `{}` は省略可

注意事項

`if(`e_1`){`e_2`} else {`e_3`}` を使う際，e_2の実行結果の型と e_3の実行結果の型は同じでなければならない

条件式の例

数学

$$\begin{cases} 5.0 & \text{if } n < 10 \\ 6.0 & \text{otherwise} \end{cases}$$

Scala

```
if(n<10){5.0}
else{6.0}
```

$$\begin{cases} 0.04 & \text{if } n \leq 1000 \\ 0.045 & \text{if } 1000 < n \leq 5000 \\ 0.055 & \text{if } 5000 < n \leq 10000 \\ 0.06 & \text{otherwise} \end{cases}$$

```
if(n<=1000){0.04}
else if(n<=5000){0.045}
else if(n<=10000){0.055}
else {0.06}
```

条件式の入れ子

条件式も式なので，`if(`e_1`){`e_2`}` や `if(`e_1`){`e_2`} else {`e_3`}` の式 e_1, e_2, e_3 としてつかうことができる

```
if(n<=1000){0.04}
else if(n<=5000){0.045}
else if(n<=10000){0.055}
else {0.06}
```

は以下の式から {} を省略したもの

```
if(n<=1000){0.04}
else{if(n<=5000){0.045}
else{if(n<=10000){0.055}
else {0.06}}}
```

条件式の入れ子

条件式も式なので，if(e_1){e_2} や if(e_1){e_2} else {e_3} の式 e_1, e_2, e_3 としてつかうことができる

```
if(n<=1000){0.04}
else if(n<=5000){0.045}
else if(n<=10000){0.055}
else {0.06}
```

は以下の式から {} を省略したもの

```
if(n<=1000){0.04}
else{if(n<=5000){0.045}
else{if(n<=10000){0.055}
else {0.06}}}
```

$$
\begin{aligned}
e &\triangleq \texttt{if(n<=1000)\{0.04\} else\{}e'\texttt{\}} \\
e' &\triangleq \texttt{if(n<=5000)\{0.045\} else\{}e''\texttt{\}} \\
e'' &\triangleq \texttt{if(n<=10000)\{0.055\} else\{0.06\}}
\end{aligned}
$$

練習問題

以下の条件式について，n が (a)500, (b)2800, (c)15000 のときの実行結果を答えよ

```
if(n <= 1000) 0.04
else if(n <= 5000) 0.045
else if(n <= 10000) 0.55
else 0.06
```

- (a):
- (b):
- (c):

練習問題

以下の条件式について，n が (a)500, (b)2800, (c)15000 のときの実行結果を答えよ

```
if(n <= 1000) 0.04
else if(n <= 5000) 0.045
else if(n <= 10000) 0.55
else 0.06
```

- (a): 0.04
- (b):
- (c):

練習問題

以下の条件式について，n が (a)500, (b)2800, (c)15000 のときの実行結果を答えよ

```
if(n <= 1000) 0.04
else if(n <= 5000) 0.045
else if(n <= 10000) 0.55
else 0.06
```

- (a): 0.04
- (b): 0.045
- (c):

練習問題

以下の条件式について，n が (a)500, (b)2800, (c)15000 のときの実行結果を答えよ

```
if(n <= 1000) 0.04
else if(n <= 5000) 0.045
else if(n <= 10000) 0.55
else 0.06
```

- (a): 0.04
- (b): 0.045
- (c): 0.06

条件の導入

現実の問題には様々な条件指定が含まれる

例題
株式会社 XYZ は時給 1200 円を被雇用者に支払っている．被雇用者は週に 20 時間から 65 時間働く．労働時間が与えられた時，その労働時間が 20 時間から 65 時間の間に収まる場合に限って被雇用者の給与を計算するプログラム `wage` を作りなさい．

条件の導入

現実の問題には様々な条件指定が含まれる

例題
株式会社 XYZ は時給 1200 円を被雇用者に支払っている．被雇用者は週に 20 時間から 65 時間働く．労働時間が与えられた時，その労働時間が 20 時間から 65 時間の間に収まる場合に限って被雇用者の給与を計算するプログラム wage を作りなさい．

$$\mathtt{wage}(h) = 1200h \text{ただし} 20 \leq h \leq 65$$

条件の導入

現実の問題には様々な条件指定が含まれる

例題
株式会社 XYZ は時給 1200 円を被雇用者に支払っている．被雇用者は週に 20 時間から 65 時間働く．労働時間が与えられた時，その労働時間が 20 時間から 65 時間の間に収まる場合に限って被雇用者の給与を計算するプログラム wage を作りなさい．

$$\mathtt{wage}(h) = 1200h \text{ただし} 20 \leq h \leq 65$$

どうやって条件 (ただし$20 \leq h \leq 65$) をプログラムで表すか？

wage in Scala

```
def wage:
(Double) => Double
=
(h) =>
/**
 * wage(h) = 1200h if 20 ≤ h ≤ 65
 */
{if(20 <= h && h <= 65){h * 1200}}
```

wage in Scala

```
def wage:
(Double) => Double
=
(h) =>
/**
 * wage(h) = 1200h if 20 <= h <= 65
 */
{if(20 <= h && h <= 65){h * 1200}}
```

しかしこれではダメ:

```
 found   : Unit
 required: Number
       {if(20 <= h && h <= 65){h * 1200}}
        ^
```

条件にかなわなかった時に何も値を出力しないのが問題！

wage in Scala

正しいプログラム:

```
def wage:
(Double) => Double
=
(h) =>
/**
 * wage(h) = 1200h if 20 ≤ h ≤ 65
 */
{if(20 <= h && h <= 65){h * 1200}else{0}}
```

else{0} として型が一致する値を用意しておく

目次

記号情報を扱う値たち

今日のプログラムには数値の計算だけでなく，記号情報 (名前や言葉，文章) を扱うことが求められる (例: 検索エンジン).
プログラミング言語はこのような記号情報を表現・処理するための方法を少なくとも一通り提供している.
Scala では以下の表現方法が使える.

- シンボル: `'abc`, `' 青谷`,
- 文字: `'a'`, `' 青'`, `'1'`
- 文字列: `"abc"`, `" 青谷"`, `"1234"`

シンボル，文字，文字列は全て値である．それぞれ値の型は以下の通り.

- シンボルの型: `Symbol`
- 文字の型: `Char`
- 文字列の型: `String`

シンボル

- シンボル: 単一引用符 ' を先頭とする文字の並びで表される値
  - ただし ' の直後の 1 文字目は数字以外でなければならない
- シンボルの型: `Symbol`

例

```
scala> 'abc
res0: Symbol = 'abc
```

```
scala> '123
<console>:1: error: unclosed character literal
       '123
       ^
```

```
scala> 'abc123
res1: Symbol = 'abc123
```

シンボルの上の計算

2 つのシンボルは等号 == と不等号 != によって比較可能

```
scala> 'a == 'b
res0: Boolean = false

scala> 'a != 'b
res1: Boolean = true

scala> 'a == 'a
res2: Boolean = true
```

例: 英和辞書

- englishToJapanese('Japan)='日本
- englishToJapanese('Germany)='ドイツ
- englishToJapanese('Italy)='イタリア

となる簡単な英和辞書函数 englishToJapanese

```
def englishToJapanese:
(Symbol) => Symbol
=
(s) =>
{ if(s == 'Japan) '日本
  else if(s == 'Germany) 'ドイツ
  else if(s == 'Italy) 'イタリア
  else '未登録です
}
```

文字列

文字列は二重引用符" で囲まれた文字の並び．シンボルとは異なり，" の直後に数字が来ても良いし，" の間に空白文字がきてもよい．

- "" (空の文字列)
- " 東工大"
- "1234" ('1234 はダメ)
- "Tokyo Institute of Technology"
  ('Tokyo Institute of Technology はダメ)

目次

例: 画素の位置を表すデータ

ディスプレイの画素 (pixel) の位置は x 座標と y 座標で表される．従って 2 つの整数 (通常は `Int` で十分) が与えられた時，画素が指定できる．

例: 画素の位置を表すデータ

ディスプレイの画素 (pixel) の位置は x 座標と y 座標で表される．従って 2 つの整数 (通常は `Int` で十分) が与えられた時，画素が指定できる．

データ構造の定義

位置を表す値のために，データ構造を定義する．2 つの整数からなる位置のデータ構造は以下のように定義する．

```
case class Position (x:Int, y:Int)
```

例: 画素の位置を表すデータ

ディスプレイの画素 (pixel) の位置は x 座標と y 座標で表される．従って 2 つの整数 (通常は Int で十分) が与えられた時，画素が指定できる．

データ構造の定義

位置を表す値のために，データ構造を定義する．2 つの整数からなる位置のデータ構造は以下のように定義する．

```
case class Position (x:Int, y:Int)
```

Position の値の生成

画素の位置 (x, y)を表す値は Position(x,y) によって作られる

```
scala> Position(3,2)
res0: Position = Position(3,2) //値の型は Position
scala> Position(3*3,2*2)       //Int 型の値を作り出す式も可
res1: Position = Position(9,4)
```

Position を使ったプログラム

Position p が与えられた時，p から $(0,0)$までの距離を求める函数 distanceToOrigin を定義せよ

```
def distanceToOrigin : (Position) => Double
= (p) =>
/**
 * 目的: pから(0,0)までの距離を計算
 * 例:
 * distanceToOrigin(Position(0,0))=0
 * distanceToOrigin(Position(0,1))=1
 **/
 { p match{
  case Position(x,y) => math.sqrt(x*x + y*y)
}}
```

Position 型の値の分解

p を Position 型の変数とし，Position の定義は以下とする:

```
case class Position (x:Int, y:Int)
```

以下のプログラムは p の実際の値が Position(a,b) という形に分解できるとき，変数 x の実際の値を変数 a の値，変数 y の実際の値を変数 b の値として math.sqrt(a*a + b*b) を計算する

```
p match {
  case Position(a,b) => math.sqrt(a*a + b*b)
}
```

分解できない時このプログラムはエラーとなるが，この例においては分解できないことがあり得ない．Position 型の値というのは Position(i,j)(i および j は任意の Int 型の値とする) という形でしか定義され得ない．

例: 名簿の要素

人の名前，電話番号，メールアドレスを管理する以下のような名簿を考える.

name	phone	mail
“青谷知幸”	“03-5734-3868”	“aotani@is”
“鹿島亮”	“03-5734-3502”	“kashima@is”

名簿の各要素 (Entry と呼ぶことにする) は

- 名前を表す文字列
- 電話番号を表す文字列
- メールアドレスを表す文字列

からなる．Entry を表すデータ構造は以下のように定義される

```
case class Entry(name:String, phone:String, mail:String)
```

Entry を使うプログラム例

- Entry の定義:

```
case class Entry(name:String, phone:String, mail:String)
```

- Entry 型の値から名前を取り出す函数 getName:

```
def getName :
(Entry) => String
=
(e) =>
/**
 * 例: getName(Entry("青谷知幸","3868","aotani")) は "青谷知幸"
 */
{
  e match { case Entry(n,p,m) => n }
}
```

Entry を使うプログラム例

- Entry の定義:

```
case class Entry(name:String, phone:String, mail:String)
```

- Entry 型の値からメールアドレスを取り出す函数 getMail:

```
def getMail :
(Entry) => String
=
(e) =>
/**
 * 例: getMail(Entry("青谷知幸","3868","aotani")) は "aotani"
 */
{
  e match { case Entry(n,p,m) => m }
}
```

Entry を使うプログラム例

- Entry の定義:

```
case class Entry(name:String, phone:String, mail:String)
```

- Entry 型の値における name の値が" 青谷知幸" のとき真, そうでないとき偽を返す函数 isAotani:

```
def isAotani :
(Entry) => Boolean
=
(e) =>
/**
 * 例: isAotani(Entry("青谷知幸","3868","aotani")) は true
 */
{
 e match { case Entry(n,p,m) => n == "青谷知幸" }
}
```

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

例題

午前 0:00 からの経過時間 (時，分，秒であらわされる) を受け取って経過秒数に変換する函数 `timeToSeconds` を定義せよ.

1. データの解析と設計

“午前 0:00 からの経過時間 (時，分，秒であらわされる)を受け取って経過秒数に変換する函数 timeToSeconds を定義せよ”

より，時，分，秒であらわされる経過時間を表現するためのデータ構造が必要であることがわかる．このデータ構造を Time と呼ぶことにして，Time を以下のように定義する．

```
case class Time(hours:Int, minutes:Int, seconds:Int)
```

経過秒数は Int で表すことにする．$24 * 3600 \leq 2^{31} - 1$が満たされるので，Int 型の値で表現できることに注意．

2. 函数の型と目的の記述

“午前 0:00 からの経過時間 (時，分，秒であらわされる) を受け取って経過秒数に変換する函数 `timeToSeconds` を定義せよ”

- 入力となる経過時間は `Time` 型の値
- 出力となる経過秒数は `Int` 型の値

```
def timeToSeconds:
(Time) => Int
=
(t) =>
/**
 * 目的: 経過時間を表すtを経過秒数に変換
 */
{...}
```

3. 函数の利用例の記述

“午前 0:00 からの経過時間 (時，分，秒であらわされる) を受け取って経過秒数に変換する函数 timeToSeconds を定義せよ”

```
def timeToSeconds:
(Time) => Int
=
(t) =>
/**
 * 目的: 経過時間を表すtを経過秒数に変換
 * 例: timeToSeconds(Time(0,0,1)) は 1
 *     timeToSeconds(Time(0,1,0)) は 60
 *     timeToSeconds(Time(1,0,0)) は 3600
 */
{...}
```

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

```
def timeToSeconds: (Time) => Int =
(t) =>
/**
 * 目的: 経過時間を表すtを経過秒数に変換
 * 例: timeToSeconds(Time(0,0,1)) は 1
 *     timeToSeconds(Time(0,1,0)) は 60
 *     timeToSeconds(Time(1,0,0)) は 3600
 * テンプレート: t match{ case Time(h,m,s) => ... }
 */
{...}
```

5. 函数の本体 (中身) の記述

```
def timeToSeconds: (Time) => Int =
(t) =>
/**
 * 目的: 経過時間を表すtを経過秒数に変換
 * 例: timeToSeconds(Time(0,0,1)) は 1
 *     timeToSeconds(Time(0,1,0)) は 60
 *     timeToSeconds(Time(1,0,0)) は 3600
 * テンプレート: t match{ case Time(h,m,s) => ... }
 */
{ t match{case Time(h,m,s) => h*3600 + m*60 + s}}
```

目次

例題: 図形の面積・長さ

図形の面積を求める函数 area と周囲の長さを求める函数 perimeter を考える．簡単のため，図形には円と正方形があることにしよう．area と perimeter を定義せよ．

1. データの解析と設計

“図形の面積を求める函数 area と周囲の長さを求める函数 perimeter を考える．簡単のため，図形には円と正方形があることにしよう．area と perimeter を定義せよ．”

円:

```
case class Circle(center:Position, radius:Int)
```

正方形:

```
case class Square(center:Position, length:Int)
```

2. 函数の型と目的の記述

“図形の面積を求める函数 area と周囲の長さを求める函数 perimeter を考える．簡単のため，図形には円と正方形があることにしよう．area と perimeter を定義せよ．”

```
def area :
(???) => Double
=
(s) =>
/**
 * 目的: 図形sの面積を計算する
 */
{...}
```

???には何を書けば良い？　Circle？　Square？

解決: 2 つの型を包含する型

“図形の面積を求める函数 area と周囲の長さを求める函数 perimeter を考える．簡単のため，図形には円と正方形があることにしよう．area と perimeter を定義せよ．”

図形を表す型 (=集合)Shape を定義して，その部分集合として Circle と Square を定義すれば良い．

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

- abstract class Shape は Shape 型を定義
  - ただし Shape(...) という値は作れない
- case class *X*(...) extends *Y*は型 (=集合)*X*と *Y*の間に包含関係 ($X \subseteq Y$) があることを宣言

包含関係の確認

```
scala> def id :
     | (Shape) => Shape
     | =
     | (x) =>
     | {x}                    //受け取った値をそのまま出力
id: Shape => Shape

scala> id(Circle(Position(3,2),3))
res0: Shape = Circle(Position(3,2),3.0)

scala> id(Square(Position(3,2),4))
res1: Shape = Square(Position(3,2),4.0)
```

包含関係の確認

```
scala> def id :
     | (Circle) => Shape        //引数が Shape ではなく Circle
     | =
     | (x) =>
     | {x}                      //受け取った値をそのまま出力
id: Circle => Shape

scala> id(Circle(Position(3,2),3))
res0: Shape = Circle(Position(3,2),3.0)

scala> id(Square(Position(3,2),4)) //Square 型の値を渡す
<console>:16: error: type mismatch;
 found   : Square
 required: Circle
              id(Square(Position(3,2),3))
                        ^
```

2. 函数の型と目的の記述 (改)

“図形の面積を求める函数 area と周囲の長さを求める函数 perimeter を考える．簡単のため，図形には円と正方形があることにしよう．area と perimeter を定義せよ．”

```
def area :
(Shape) => Double
=
(s) =>
/**
 * 目的: 図形sの面積を計算する
 */
{...}
```

3. 函数の利用例の記述

“図形の面積を求める函数 area と周囲の長さを求める函数 perimeter を考える．簡単のため，図形には円と正方形があることにしよう．area と perimeter を定義せよ．”

```
def area :
(Shape) => Double
=
(s) =>
/**
 * 目的: 図形sの面積を計算する
 * 例: area(Square(Position(10,10),4)) は 16
 *     area(Circle(Position(10,10),2)) は 4*PI
 */
{...}
```

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

```
def area : (Shape) => Double =
(s) =>
/**
 * 目的: 図形sの面積を計算する
 * 例: area(Square(Position(10,10),4)) は 16
 *     area(Circle(Position(10,10),2)) は 4*PI
 * テンプレート: s match{ case Square(p,e) => ...
 *                        case Circle(p,r) => ... }
 */                    Square(p,e) と Circle(p,r) で場合分け
{...}
```

5. 函数の本体 (中身) の記述

```
def area : (Shape) => Double =
(s) =>
/**
 * 目的: 図形sの面積を計算する
 * 例: area(Square(Position(10,10),4)) は 16
 *     area(Circle(Position(10,10),2)) は 4*PI
 * テンプレート: s match{ case Square(p,e) => ...
 *                        case Circle(p,r) => ... }
 */
{s match{
  case Square(p,e) => e*e       //補助函数 areaOfSquare へ
  case Circle(p,r) => r*r*PI    //補助函数 areaOfCircle へ
 }}
```

5. 函数の本体 (中身) の記述

```
def area : (Shape) => Double =
(s) =>
/**
 * 目的: 図形sの面積を計算する
 * 例: area(Square(Position(10,10),4)) は 16
 *     area(Circle(Position(10,10),2)) は 4*PI
 * テンプレート: s match{ case Square(p,e) => ...
 *                        case Circle(p,r) => ... }
 */
{s match{
  case Square(p,e) =>
    areaOfSquare(Square(p,e))   //補助函数を利用
  case Circle(p,r) =>
    areaOfCircle(Circle(p,r))   //補助函数を利用
 }}
```

5. 函数の本体 (中身) の記述

補助函数の定義 (目的，例，テンプレートは省略):

```
def areaOfCircle : (Circle) => Double
=
(c) =>
{ c match{
    case Circle(p,r) => r*r*PI
  }}

def areaOfSquare : (Square) => Double
=
(s) =>
{ s match{
   case Square(p,e) => e*e
  }}
```

5. 函数の本体 (中身) の記述

```
def area : (Shape) => Double =
(s) =>
/**
 * 目的: 図形sの面積を計算する
 * 例: area(Square(Position(10,10),4)) は 16
 *     area(Circle(Position(10,10),2)) は 4*PI
 * テンプレート: s match{ case Square(p,e) => ...
 *                        case Circle(p,r) => ... }
 */
{s match{
  case Square(p,e) =>
    areaOfSquare(Square(p,e))   //補助函数を利用
  case Circle(p,r) =>
    areaOfCircle(Circle(p,r))   //補助函数を利用
 }}
```

5. 函数の本体 (中身) の記述

```
def area : (Shape) => Double =
(s) =>
/**
 * 目的: 図形sの面積を計算する
 * 例: area(Square(Position(10,10),4)) は 16
 *     area(Circle(Position(10,10),2)) は 4*PI
 * テンプレート: s match{ case Square(p,e) => ...
 *                        case Circle(p,r) => ... }
 */
{s match{
  case x@Square(p,e) =>      //Square(p,e) を x とする
    areaOfSquare(x)          //補助函数を利用
  case y@Circle(p,r) =>      //Circle(p,r) を y とする
    areaOfCircle(y)          //補助函数を利用
 }}
```

目次

課題

以降のページにある課題をとき，02.zip を OCW/OCW-i を使って提出せよ．提出締め切りは4 月 28 日 15:00(JST)とする．

課題の取り組み方

- /home/aotani/Documents/classes/gairon16/02 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 02 ⏎
```

- 各プログラムはそれぞれ $HOME/Documents/gairon16/02/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 02.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 02 ⏎
./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”02” ディレクトリに 02.zip ができる．

課題 1: payback.scala

とある米国のクレジットカード会社は利用者が年間に使用した金額の一部を利用者に払い戻すことにした.

- 最初の 500 ドル以下には 0.25%
- 次の 1000 ドルまで (つまり年間使用額が 500 ドルより大きく 1500 ドル以下) には 0.50%
- 次の 1000 ドルまで (つまり年間使用額が 1500 ドルより大きく 2500 ドル以下) には 0.75%
- 2500 ドルより大きい部分は 1.0%

従って,

- 年間 400 ドル利用した顧客は 1 ドル $=0.25 \times 0.01 \times 400$
- 年間 1400 ドル利用した顧客は 5.75 ドル
  $=0.25 \times 0.01 \times 500 + 0.5 \times 0.01 \times 900$

利用者の年間使用額が与えられた時，その利用者が受け取る金額を計算する函数 `payBack` を開発せよ．入力はドルを `Double` 型で与えられるものとする.

課題 2: perimeter.scala

図形の周囲の長さを計算する函数 perimeter を定義せよ．ただし図形は円と正方形からなり，以下で定義されるものとする:

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

また Position の定義は以下とする:

```
case class Position (x:Int, y:Int)
```

例

- perimeter(Square(Position(10,10),4))≡16
- perimeter(Circle(Position(10,10),20))≡40*PI

課題 3: check_color.scala

色当てゲームは

- 出題者が 2 枚のタイルにそれぞれ色を割り当てる
- 挑戦者がそれぞれのタイルに何色が割り当てられたかを予想する

ゲームである．出題者の割り当てを $\langle t_1, t_2 \rangle$，挑戦者の予想を $\langle g_1, g_2 \rangle$と書くことにする．挑戦者が予想を答えると，出題者は以下のように返答する

- `'Perfect` if $t_1 = g_1$ かつ $t_2 = g_2$
- `'OneColorAtCorrectPosition` if $t_1 = g_1$ と $t_2 = g_2$ のどちらか一方のみが成立
- `'OneColorOccurs` if 上記以外で $g_1 = t_2$ と $g_2 = t_1$ の両方もしくは一方が成立
- `'NothingCorrect` otherwise

シンボルの組を 2 つ受け取って，シンボルを 1 つ返す函数 `checkColor` を定義せよ．

課題 3: check_color.scala

取り決め:

s1, s2 をシンボルとするとき，シンボルの組 ⟨s1, s2⟩は SymbolPair(s1,s2) で表されるものとする

例:

```
scala> checkColor(SymbolPair('red,'blue),SymbolPair('red,'blue))
res0: Symbol = 'Perfect

scala> checkColor(SymbolPair('red,'blue),SymbolPair('blue,'red))
res1: Symbol = 'OneColorOccurs

scala> checkColor(SymbolPair('red,'blue),SymbolPair('black,'yellow))
res2: Symbol = 'NothingCorrect

scala> checkColor(SymbolPair('red,'blue),SymbolPair('red,'red))
res3: Symbol = 'OneColorAtCorrectPosition
```