第B章　プログラミング言語 Scala

Scala は 2003 年に発表された、比較的新しいマルチパラダイム（オブジェクト指向 + 関数型）のプログラミング言語である。Java 仮想機械 (JVM) 上に実装され、Java のライブラリとの連携が容易であり、Java の後継言語との評判も高い。

コンパイル時に型検査をする（静的な型付けをする）が、＿＿＿＿＿により多くの場合で型宣言を省略することができ、動的な型付けの言語に近い柔軟性を持つ。

ここでは、Scala の関数型言語的側面に焦点を絞って解説する。

Scala の情報のページ

`http://www.scala-lang.org/`　Scala のホームページ

B.1　Scala の実行

`scala` というコマンドで対話的な処理系が起動できる。（Java と同様に `main` メソッドを定義して、Scala コンパイラで class ファイルを生成して実行することもできるが、このドキュメントではこの実行方法の説明は割愛する。）

対話的な処理系では、式を入力すると、その式を評価した結果が出力される。

```
scala> 14
res0: Int = 14
scala> 2+6
res1: Int = 8
```

式の評価の他に、次のコマンドが利用可能である。

コマンド	省略形	意味
`:load` *file*	`:l`	*file* をロードする。
`:help`	`:h`	ヘルプを表示する。
`:replay`	`:r`	リセットし、それまでのコマンドを再実行する。
`:quit`	`:q`	scala を終了する。

通常は、ごく短い式を試す以外は、ファイルに関数などを定義して、`:load` コマンドで読み込む。Scala のソースファイルには、通常＿＿＿＿＿という拡張子をつける。

B.2　変数定義と関数定義

変数は、＿＿＿ というキーワードで定義する。

```
val count = 1
```

val で定義された変数は、代入不可である（つまり、値を変更することはできない）。Scala では他の形式を使って代入可能な変数を定義することもできるが、このドキュメントでは関数型言語的側面に注目するので、代入可能な変数の説明は割愛する。

関数の定義は ____というキーワードを使う。

```
def foo(x: Int): Int = {
   val y = x*x
   val z = y*y
   z*z
}
```

def のあとの foo が関数名、括弧の間の “x: Int” は x が仮引数名で、Int がその型である。引数が複数個ある場合は、この「変数名: 型」を , （コンマ）で区切って並べる。閉じ括弧の後の :と=の間の Int は戻り値型である。（ただし Scala では戻り値型の宣言は省略できる場合が多い。）

= の右辺にブレース（{, }）で囲んで関数本体を書く。ただし、関数本体が 1 つの式からなるときはブレースを省略して良い。例えば 2 倍する関数 twice は次のように定義できる。

```
def twice(x: Int) = 2*x
```

B.3　ケースクラスとパターンマッチ

ケースクラスは、関数型言語の_____の代替を目的とし、主に以下のような点で他と扱いが異なるクラスのことである。

- new キーワードを使わないで、コンストラクターを呼び出すことができるようになる。
- コンストラクターの引数と対応するフィールドが自動的に定義される。
- 後述のパターンマッチで使用することができる

例として、次のようなケースクラスを考える。

```
abstract class Color
case class NamedColor(name: String) extends Color
case class RGBColor(r: Int, g: Int, b: Int) extends Color
```

これは、色を名前 または RGB 値で表現するためのデータ型の定義である。例えば RGBColor クラスは, r, g, b という 3 つの Int 型のフィールドをもち、コンストラクターは引数として、この順にフィールドの初期値を受け取る。

NamedColor("Cyan") や、RGBColor(0x80, 0, 0x80) のような式で Color 型のオブジェクトを生成することができるようになる。Color というアブストラク

トクラスは、NamedColor と RGBColor という 2 つのケースクラスの共通のスーパークラスで、この両者のオブジェクトを参照する変数の型として必要になる。

ケースクラスは、＿＿＿＿＿＿＿＿＿＿ で場合分けの処理をすることができる。

```
def colorToStr(c: Color) : String = c match {
  case NamedColor(n)     => n
  case RGBColor(r, g, b) => "#%02x%02x%02x".format(r, g, b)
}
```

この例の場合、colorToStr 関数の引数が、NamedColor のインスタンスなら、その name フィールドがそのまま戻り値になる、RGBColor のインスタンスなら r, g, b フィールドから 16 進数による文字列が生成される。

一般にパターンマッチは以下のようなカタチを持つ。

```
式0 match {
  case パターン1 => 式1
  case パターン2 => 式2
  ...
}
```

式 $_0$ を計算して、パターン $_1$ にマッチするならば、パターン中の変数に対応するフィールドの値を代入して、式 $_1$ を計算する、そうでなければ、パターン $_2$ にマッチするならば、式 $_2$ を計算する、... というように、上から順に式がパターンとマッチするか試して、対応する => の右辺の式を計算する。

オブジェクト指向言語は、処理（メソッド）の種類は増えずにデータの種類（クラス）が増えていくことを想定している。そのため各メソッドの定義をクラスの中にまとめて書く。一方、関数型言語は、データの種類は増えずに処理（関数）が増えていくことを想定している。そのため関数の定義の中にデータの各種類に対する処理をまとめて書く。

B.4 リスト

リスト (List) は単純だが、有用性の高いデータ型である。一般に関数型言語で多用されるデータ型である。

リスト型の定義 リスト型は以下のような定義で説明することができる。ただし、この定義は説明のためのもので、実際の Scala ライブラリの中の定義とは、後で述べるように細かい点で違いがある。

```
abstract class List[T]
case class Nil[T]() extends List[T]
case class Cons[T](head: T, tail: List[T]) extends List[T]
```

List 型は Nil というフィールドのないコンストラクターと Cons というコンストラクターからなる。Cons は tail という自分自身と同じ型のフィールドを持っている。

ここで、[と] の中の T は＿＿＿＿＿＿＿＿＿＿である。Java では型パラメーターは <T>というように< と > を使うが、Scala は<と>は別の用途で使用するので、別の括弧 [と] を用いている。

注: 実際の Scala の標準ライブラリの List 型は、上の定義よりも少し凝った定義になっている。それにより、Nil のコンストラクターには () が不要、つまり Nil() と書く必要がなく、単に Nil と書けばよいし、Cons の代わりに中置記法の（右結合の）コンストラクター :: を使用する。つまり、Cons(x, xs) ではなく、x::xs と書く。

リストリテラル　Scala の標準ライブラリでは、リストを構築するために List という可変個引数の関数が用意されている。例えば、List(1) は 1 つの要素を持つリスト 1::Nil を表し、List(1, 2, 3) は 3 つの要素を持つリスト 1::2::3::Nil を表す。

リストに対する関数の例　リストに対する関数は、パターンマッチを使って定義することができる。

まず、例として整数のリストの要素の和を求める関数の書き方を紹介する。

```
def sum(xs: List[Int]) : Int = xs match {
  case Nil     => 0
  case x :: xs => x + sum (xs)
}
```

使用例:

```
scala> val xs = List(1, 2, 3)
xs: List[Int] = List(1, 2, 3)
scala> sum(xs)
res2: Int = 6
```

問 B.4.1 整数のリストのすべての要素の積を計算する関数

```
prod(xs: List[Int]) : Int
```

を定義せよ。

リストに対しては多くの標準ライブラリ関数が用意されているが、これらは、効率の点から List クラスのメソッドとして用意されている。つまり、

```
somefunction(list, y, z)
```

という書き方（上の sum のような使い方）をするのではなくて、Java のメソッドのように、

```
list.somemethod(y, z)
```

という使い方をする。統一感がないが、Java 仮想機械の上に実装されたハイブリッド言語としては致し方のない点かもしれない。

例えば、n 番め（C, Java の配列と同様、最初の要素は 0 番め）の要素を返す apply というメソッドは

```
scala> xs.apply(1)
res3: Int = 2
```

のように使う。

さらに、無引数のメソッドはフィールドのように

```
list.somemethod
```

という書き方で呼び出すことができる。例えば、リストの長さを求めるメソッド length は、

```
scala> xs.length
res4: Int = 4
```

のように使うことができる。このような無引数のメソッドとして、isEmpty（リストが空ならば真を返す）などが用意されている。

使用例:

```
scala> xs.head
res5: Int = 1
scala> xs.tail
res6: List[Int] = List(2, 3)
```

Scala では演算子もユーザーが定義することができる。++は標準ライブラリで用意されているリストの連接の演算子である。

使用例:

```
scala> val ys = List(8, 9)
ys: List[Int] = List(8, 9)
scala> xs ++ ys
res7: List[Int] = List(1, 2, 3, 8, 9)
scala> ys ++ ys
res8: List[Int] = List(8, 9, 8, 9)
scala> ys ++ xs
res9: List[Int] = List(8, 9, 1, 2, 3)
```

B.5 リストに対する高階関数と関数リテラル

高階関数は＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿関数（あるいはメソッド）である。以下の説明では、A => B は A を引数の型、B を戻り値の型とする関数の型を表す。

リストに対しては、多くの高階関数が標準ライブラリとして用意されている。例えば、map は、リストの要素に一斉に関数を適用し、その戻り値のリストを返すメソッドである。

このメソッドは次のように使用することができる。

```
scala> xs.map(twice)
res10: List[Int] = List(2, 4, 6)
```

ところで高階関数の引数として使う twice のように小さな関数にいちいち名前をつけるのは面倒なので、Scala では名前をつけずに関数を表現する記法が用意されている。これを＿＿＿＿＿＿＿＿あるいは＿＿＿＿＿＿という。（この名前は、かつて数学の一分野で、この目的のためにギリシャ文字の λ が使われたことに由来する。）

例えば、(x: Int) => 2*x という式で twice と同等の関数を表す。=> の左辺に括弧で囲んで仮引数のコンマ区切りの並びを、右辺に関数本体を書く。次は関数リテラルの使用例である。

```
scala> xs.map((x: Int) => 2*x)
res11: List[Int] = List(2, 4, 6)
```

filter は、リストの要素の中で、与えられた関数の値を真にする要素だけのリストを返すメソッドである。

使用例:

```
scala> xs.filter((x: Int) => x%2 == 1)
res12: List[Int] = List(1,3)
```

また、リストを生成するのに便利な高階関数も挙げておく。

```
def iterate[T](a: T, f: (T) => T, p: (T) => Boolean) : List[T] =
  if (p(a)) Nil else a :: iterate(f(a), f, p)
```

使用例:

```
scala> iterate(1, (x: Int) => x*2, (x: Int) => x>100)
res13: List[Int] = List(1, 2, 4, 8, 16, 32, 64)
```

iterate(a, f, p) は初期値 a, 漸化式 f からリスト（“数”列とは限らない）を生成する関数である。通常のリストでは無限列を扱うことはできないので、条件 p が成り立つところで打ち切るようにしている。

B.6　タプル

次に、リストとタプル（組）を利用する関数を紹介する。

タプル（組）は要素を “,” （コンマ）で区切って並べ、“(” と “)” で囲んで表す。組はリストの場合と異なり、要素の型が同一である必要はない。(1, 'a') という式の型は＿＿＿＿＿＿＿＿と表記される。また、(2, 'b', List(3)) という式の型は＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿と表記される。

組はパターン中に使用することもできる。

zip は 2 つのリストの同じ位置の要素を組にしたもののリストを返すメソッドである。長さが異なる場合は、短い方にあわせる。

使用例:

```
scala> List(1, 3, 5, 7, 11).zip(List(2, 4, 6, 10))
res14: List[(Int, Int)] = List((1,2), (3,4), (5,6), (7,10))
```

B.7 例: エラトステネスの篩（ふるい）

最後に、素数列を生成するプログラムを例として挙げる。ここまでの基本関数を利用すれば比較的簡潔に定義することができる。（ただし、この定義は効率面での改良の余地は多いにある。）

```
def from2(n: Int) = iterate(2, (x: Int) => x+1, (x: Int) => x>n)

def sieve(xs: List[Int]) : List[Int] = xs match {
  case Nil   => Nil
  case y::ys => ys.filter((z: Int) => z%y != 0)
}

def primes(n: Int) : List[Int] = {
  val seqs  = iterate(from2(n), sieve, (xs:List[Int]) => xs.isEmpty)
  seqs.map((xs:List[Int]) => xs.head)
}
```

注: _（アンダースコア）を使った Scala の略記法を使えば、この例はもう少し簡潔に書くことができるが、ここでは説明を割愛する。

使用例:

```
scala> primes(100)
res15: List[Int] = List(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
  37, 41, 43, 47, 53, 59, 61, 71, 73, 79, 83, 89, 97)
```

また、ここまでのプログラムに副作用としての代入（変数の破壊的な書換え）は一度も使われていないことに注意する。非破壊的なプログラムは、一般に並列実行と相性が良いとされている。

問 B.7.1 Stream という Scala の遅延評価リストのクラスの使い方を調べ、Stream を使って、素数の無限リストとしてエラトステネスの篩を実装せよ。

B.8 遊び

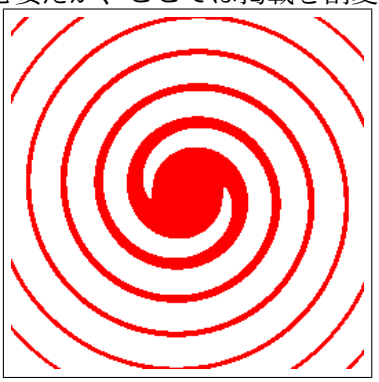
記号や数値だけでは地味なので、最後に遊び心の入ったプログラムとして、高階関数を画像の生成に応用した例[1]を紹介する。

基本的なアイデアは画像を、＿＿＿＿＿＿＿＿＿＿＿＿と見なすことである。すると、画像に対する操作は必然的に高階関数になる。

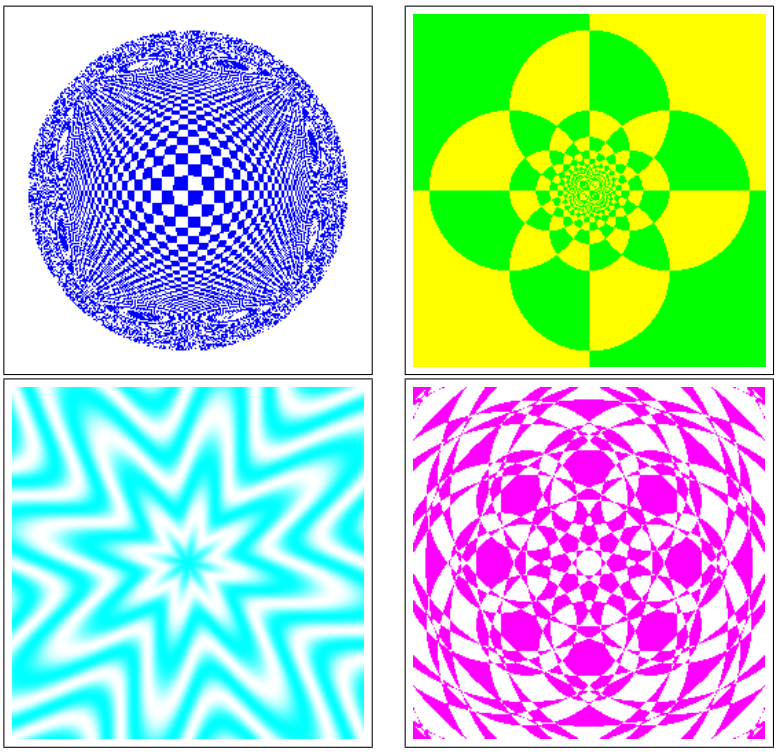
[1] この節のプログラム例は、Conal Elliott: “Functional Images”, In *The Fun of Programming*, pp.131–150,(2003) のプログラム（オリジナルは Haskell）を Scala に移植したものである。

```
type Point     = (Double,Double)
type Image[a]  = Point => a
type Region    = Image[Boolean]
type Transform = Point => Point
type Filter[a] = Image[a] => Image[a]

def vstrip: Region = p => {
   val (x,_) = p
   x.abs < 0.5
}

def dist0: Image[Double] = p => {
   val (x,y) = p
   sqrt(x*x + y*y)
}

def rotateP (t: Double) : Transform = p => {
    val (x, y) = p
    val c = cos(t)
    val s = sin(t)
    (x*c-y*s, y*c+x*s)
}

def swirlP (r: Double) : Transform = p =>
    rotateP (dist0(p) * 2 * Pi / r) (p)

def swirl[T](r: Double) : Filter[T] = im =>
    p => im (swirlP(-r)(p))
```

swirl(1)(vstrip) の結果を次に示す。（実際に画面に表示するためのコードも必要だが、ここでは掲載を割愛する。）



このような関数的手法で作成された画像の例を、（定義抜きで）最後に示す。これらの画像の具体的な定義は、上記の Elliott の論文に掲載されている。



問 B.8.1 アニメーションを、座標と時刻から色への関数と見なして、関数的に作成してみよ。