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 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が、＿＿＿＿により多くの場合で型宣言を省略することができる。

ここでは、Scala の関数型言語的側面に絞って解説する。

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説明を割愛する。）

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`	`:h`	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説明を割愛する。

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。（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。

参考までに、Haskell での対応する代数的データ型の定義を示す。

```
data Color = NamedColor { name :: String }
           | RGBColor { r :: Int, g :: Int, b :: Int }
```

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、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 以下のような定義で説明することができる。（ただし、この定義は実際の Scala ライブラリの定義とは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 では型パラメータは <T>という記法を使うが、Scala は<と>は別の用途で使用するので、記法が異なっている。

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 () が必要ないし、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。

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。

リストに対する基本的な関数 以下で代表的なリストに対する関数を紹介する。これらは、すべてパターンマッチを使って定義されている。

注: これらと同等の関数は、実際には Scala 標準ライブラリの中で List クラスのメソッドとして用意されている。本当はこれらの定義済メソッドを使うほうが効率が良い。以下のコードは、説明用のサンプルとして提供する。

まずは、基本的な関数を紹介する。

```
def isEmpty[T](xs: List[T]) : Boolean = xs match {
  case Nil  => true
  case _::_ => false
}
```

次はリストの要素数を求める関数である。

```
def length[T](xs: List[T]) : Int = xs match {
  case Nil     => 0
  case _ :: xs => 1 + length (xs)
}
```

使用例:

```
scala> length(List(1, 2, 3))
res2: Int = 3
```

問 B.4.1 整数のリストのすべての要素を和を計算する関数

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

を定義せよ。

リストの連接をする関数と、リストのリストをフラットなリストにする関数である。

```
def append[T](xs: List[T], ys: List[T]) : List[T] = xs match {
  case Nil     => ys
  case z :: zs => z :: append(zs, ys)
}

def flatten[T](xss: List[List[T]]) : List[T] = xss match {
  case Nil => Nil
  case ys :: yss => append(ys, flatten(yss))
}
```

使用例:

```
scala> append(List(1, 2, 3), List(8, 9))
res3: List[Int] = List(1, 2, 3, 8, 9)
scala> flatten(List(List(1, 2, 3), List(5), List(8, 9)))
res4: List[Int] = List(1, 2, 3, 5, 8, 9)
```

B.5　高階関数と関数リテラル

高階関数は ________________________________ である。`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。

次に挙げるのは、リストの要素に一斉にある操作をする高階関数である。

```
def map[A,B](f : A => B, xs : List[A]) : List[B] = xs match {
  case Nil     => Nil
  case y :: ys => f(y) :: map(f, ys)
}

def filter[T](p : T => Boolean, xs : List[T]) : List[T] = xs match {
  case Nil     => Nil
  case y :: ys => if (p(y)) y :: filter(p, ys) else filter(p, ys)
}

def foldr[A,B](f : (A, B) => B, z : B, xs : List[A]) : B = xs match {
  case Nil     => z
  case y :: ys => f(y, foldr(f, z, ys))
}
```

これらの関数は次のように使用することができる。

```
scala> map(twice, List(1, 2, 3))
res3: List[Int] = List(2, 4, 6)
```

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> filter((x: Int) => i%2 == 0, List(1, 2, 3, 4))
res4: List[Int] = List(2, 4)
scala> foldr((x: Int, y: Int) => x*y, 1, List(1,2,3,4,5))
res5: Int = 120
```

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)
res8: List[Int] = List(1, 2, 4, 8, 16, 32, 64)
```

B.6 タプル

次は、リストとタプル（組）を利用する関数を挙げる。

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。

また、組はパターン中に使用することもできる。

```
  def zip[A,B] (xs : List[A], ys : List[B]) : List[(A, B)] = (xs, ys) match {
    case (Nil, _)           => Nil
    case (_, Nil)           => Nil
    case (x1::xs1, y1::ys1) => (x1, y1) :: zip(xs1, ys1)
  }

  def unzip[A,B] (xys : List[(A,B)]) : (List[A], List[B]) = xys match {
    case Nil            => (Nil, Nil)
    case (x, y) :: xys1 => {
      val (xs, ys) = unzip(xys1)
      (x::xs, y::ys)
    }
  }
```

使用例:

```
scala> zip(List(1, 3, 5, 7, 11), List(2, 4, 6, 10))
res4: List[(Int, Int)] = List((1,2), (3,4), (5,6), (7,10))
scala> unzip(List((1,2), (3,4)))
res5: (List[Int], List[Int]) = (List(1, 3),List(2, 4))
```

B.7　例: エラトステネスの篩

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

  def primes(n: Int) : List[Int] = {
    val from2 = iterate(2, (x: Int) => x+1, (x: Int) => x>n)
    val seqs  = iterate(from2, sieve, isEmpty)
    map((ys:List[Int]) => ys match { case x::xs => x }, seqs)
  }
```

使用例:

```
scala> primes(100)
res4: 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。