Scala By Example

DRAFT
July 13,2010

Martin Odersky
宮本隆志(和訳)
水尾学文(文責)

PROGRAMMING METHODS LABORATORY
EPFL
SWITZERLAND

目次

1 はじめに

Scala はオブジェクト指向と関数型のプログラミングをスムーズに統合します。一般的なプログラミングパターンを簡潔・エレガント・型安全に表現するようにデザインされています。Scala は革新的な構文をいくつか導入します。たとえば、

- 抽象型(abstract type)とミックスイン合成(mixin composition)は、 オブジェクトとモジュール・システムの概念を統合します。
- クラス階層を超えたパターン・マッチングは、 関数型とオブジェクト指向のデータ・アクセスを統合します。 その結果、XML 木の処理が大幅に単純化されます。
- 柔軟な構文と型システムにより、 先進的ライブラリと新しいドメイン特化言語の構築が可能です。

同時に、Scala は Java と互換性があります。Java のライブラリとフレームワークは、接着用コード(glue code)や宣言を追加せずに使用できます。

この文書は Scala を形式張らずに、多数の例を通して紹介します。

第 2 章と第 3 章では、Scala を興味深いものにしている特徴をいくつか明らかにします。 続く章では、Scala の構文をより詳しく紹介します。単純な式と関数から始め、オブジェクトとクラス、リストとストリーム、ミュータブル(変更可能)な状態、パターン・マッチングと話をすすめ、興味深いプログラミング技法を示す申し分のない例へと至ります。形式張らないこの説明は、より詳細かつ正確に Scala を定義する『Scala Language Reference Manual』で補完されます。

謝辞 著者らは Abelson と Sussman のすばらしい本『Structure and Interpretation of Computer Programs』[ASS96]に多くを負っています。その中の例と演習問題がここにあります。もちろん、あつかう言語は Scheme、Scala というように異なっていますが。また、例では、適切と思われる場所で Scala のオブジェクト指向的な構文を用いています。

2 最初の例

最初の例として、Scala でのクイックソートの実装を示します。

```
def sort(xs: Array[Int]) {
  def swap(i: Int, j: Int) {
    val t = xs(i); xs(i) = xs(j); xs(j) = t
  }
  def sort1(l: Int, r: Int) {
    val pivot = xs((l + r) / 2)
    var i = l; var j = r
    while (i <= j) {
      while (xs(i) < pivot) i += 1
      while (xs(j) > pivot) j -= 1
      if (i <= j) {
        swap(i, j)
        i += 1
        j -= 1
      }
    }
    if (l < j) sort1(l, j)
    if (j < r) sort1(i, r)
  }
  sort1(0, xs.length - 1)
}
```

この実装は Java や C で書くものと大変よく似ています。Scala では同じ演算子、似た制御構造を使います。ただし、少しばかり構文的な違いがあります。具体的には、

- 定義は予約語で始まります。関数定義は def で始まり、変数定義は var で始まり、値（すなわち読み出し専用の変数）の定義は val で始まります。
- シンボルの型宣言は、シンボルとコロンの後に書きます。 型宣言は省略できることもあります。コンパイラが文脈から推論するからです。
- 配列の型は T[] ではなく Array[T] と書き、配列の指定は a[i] ではなく a(i) と書きます。
- 関数は他の関数の内側に入れ子にできます。入れ子になった関数は、 それを包む関数のパラメータやローカル変数にアクセスできます。 次の例では、配列 xs は関数 swap と sort1 から見えるため、 それらにパラメータとして渡す必要はありません。

今までのところ、Scala は構文に少し変わった点があるものの、かなり普通の言語のように見えます。実際、プログラムを書くのは、普通の命令型(手続き型)でもオブジェクト指向でも可能です。これは重要なことです。なぜならこれは、Scala のコンポーネント(部品)を、Java、C#、Visual Basic などの主流言語で書かれたコンポーネントと組み合わせることを容易にする仕掛けの一つだからです。

しかし、プログラムをまったく違うスタイルでも書けます。クイックソートを再び、今度は関数型プログラミング風に書いてみます。

```
def sort(xs: Array[Int]): Array[Int] = {
  if (xs.length <= 1) xs
  else {
```

```
      val pivot = xs(xs.length / 2)
      Array.concat(
        sort(xs filter (pivot >)),
             xs filter (pivot ==),
        sort(xs filter (pivot <)))
    }
  }
```

関数型プログラミングはクイックソート・アルゴリズムの本質を簡潔にとらえています。

- 配列が空あるいは要素が１つなら、すでにソートされているので、 直ちにそれを返します。
- 配列が空でなければ、配列の真ん中の要素をピボットとして選びます。
- 配列を副配列に分割し、一つにはピボットより小さな要素を、 もう一つにはピボットより大きな要素を、 そして三つ目の副配列にはピボットと等しい要素を入れます。
- 初めの２つの副配列を sort 関数の再帰呼び出しでソートします(*1)。
- これら３つの副配列をひとつに結合して、結果を得ます。

命令型と関数型の実装のどちらも同じ漸近的計算量 --- 平均的な場合は O(N log(N))、最悪の場合は O(N2) になります。しかし命令型実装が引数の配列自体を変更するのに対して、関数型実装はソートされた新しい配列を返し、引数の配列を変更しません。ですから、関数型実装は命令型実装よりも一時的なメモリを多く必要とします。

この関数型実装によって Scala が配列への関数的操作に特化した言語の様に見えます。実際は違います。この例で用いた操作は、Scala の標準ライブラリの一部である**シーケンス**クラス Seq[T] のたんなるライブラリメソッドであり、そのライブラリ自身も Scala で実装されています。そして、配列は Seq のインスタンスなのでシーケンスクラスのメソッドはすべて使用できるのです。

とりわけ、**述語関数**(predicate function)を引数にとるメソッド filter があります。この述語関数は、配列の各要素に対して真偽値を返す必要があります。filter の結果は配列であり、元の配列の要素で述語関数が真を返すもの全てからなります。Array[T] 型オブジェクトの filter メソッドは、次のような書き方をします。

```
def filter(p: T => Boolean): Array[T]
```

ここで T => Boolean は関数の型を表わし、その関数は型 T の配列要素を引数とし Boolean 値を返します。filter のように、他の関数を引数としたり結果として返す関数は、**高階関数**と呼ばれます。

Scala は識別子と演算子の名前を区別しません。識別子は「文字で始まる、文字と数字からなる列」でも、「 '+'、'*'、':' のような特殊文字からなる列」でも構いません。Scala ではすべての識別子は中置演算子として使えます。二項演算 E op E' は常に、メソッド呼び出し E.op(E') として解釈されます。これは文字で始まる中置二項演算子にもあてはまります。したがって、式 xs filter (pivot >) は、メソッド呼び出し xs.filter(pivot >) と等価です。

---

*1 これは命令型アルゴリズムのすることと全く同じという訳ではありません。 後者では元の配列を、要素がピボットに対して小さい、あるいは、 大きいか等しい、に分けて２つの副配列へ分割します。

先のクイックソートプログラムで、filter は無名関数の引数に 3 回適用されています。最初の引数 pivot > は、引数 x をとり、値 pivot > x を返す関数を表しています。これは**部分適用された関数**の例です。この関数を、見えていない引数を明示して x => pivot > x と書いても同じです。この関数は無名、つまり名前を付けて定義されていません。引数 x の型が省略されているのは、Scala コンパイラが、関数の使用されている文脈から自動的に推論できるからです。まとめると xs.filter(pivot >) は、リスト xs の要素で pivot より小さいもの全てからなるリストを返します。

最初のクイックソートの命令型実装をよく見ると、二つ目の例で使われている多くの構文が、隠された形ではあるものの、出てきていることが分かります。

たとえば、+、-、< のような「普通の」二項演算子が特別扱かいされていないことが分かります。append などと同じく、その左側にあるオペランドのメソッドです。したがって、式 i + 1 は整数値 i のメソッド + の呼び出し i.+(i) とみなされます。もちろんコンパイラは、整数引数に対するメソッド + の呼び出しを特例とみなし、効率のよいインラインコードを生成できます（もし、ほどほどに賢いコンパイラなら、そうすることが期待されます)。

効率と、より良いエラー診断ができるように、while ループは Scala の組み込み構文となっています。しかし原理的には、たんなる事前定義された関数であってもよいのです。次は考えうる実装です。

```
def While (p: => Boolean) (s: => Unit) {
  if (p) { s ; While(p)(s) }
}
```

この While 関数は最初のパラメータとしてテスト関数をとり、そのテスト関数はパラメータをとらずブール値を返します。二番目のパラメータとしてコマンド関数をとり、そのコマンド関数もパラメータをとらず、Unit 型の結果値を返します。While 関数はテスト関数が真を返す限り、コマンド関数を呼び出します。

Scala の Unit 型は概ね Java の void に相当し、関数が特定の結果(戻り値)を返さない場合に使います。実際のところ、Scala は式指向の言語なので、すべての関数は何かしら結果を返します。もし明示的な戻り値が与えられなければ、値 () --- "unit" と発音します --- が肩代わりします。この値は Unit 型です。Unit を返す関数もまた、手続き(procedure)と呼ばれます。クイックソートの最初の実装中の swap 関数をさらに「**式指向**」な形にして、これを明示します。

```
def swap(i: Int, j: Int) {
  val t = xs(i); xs(i) = xs(j); xs(j) = t
  ()
}
```

この関数の結果(戻り値)は単に最後の式です。キーワード return は必要ありません。注意点として、明示的に値を返す関数は常に、「=」を本体あるいは定義式の前に必要とします。

3 アクターとメッセージによるプログラミング

この章では、ある応用分野の例を見てゆきます。この分野は Scala がとてもよく似合っています。電子オークションサービスを実装する仕事を考えてみて下さい。Erlang 風のアクタープロセスモデルを使ってオークション参加者の実装をしましょう。アクターとはメッセージを受信するオブジェクトです。各アクターはメッセージ受信用の「メールボックス」を持ち、それはキューとして表現されます。メールボックス中のメッセージを順番に処理することも、特定のパターンにマッチするメッセージを検索することもできます。

取引される商品ごとに一つの競売人アクターがあり、そのアクターが、商品についての情報公開、クライアントからの申し込み受付け、取引終了時の売り手および落札者への通知します。ここでは簡単な実装の概要を示します。

最初のステップとして、オークションでやり取りされるメッセージを定義します。クライアントからオークションサービスへのメッセージである AuctionMessage と、サービスからクライアントへの返答である AuctionReply の、２つの抽象基底クラス(abstrac base class)があります。両基底クラスには複数のケース(case)があり、図 3.1 で定義されています。

```
import scala.actors.Actor

abstract class AuctionMessage
case class Offer(bid: Int, client: Actor) extends AuctionMessage
case class Inquire(client: Actor)         extends AuctionMessage

abstract class AuctionReply
case class Status(asked: Int, expire: Date) extends AuctionReply
case object BestOffer                       extends AuctionReply
case class BeatenOffer(maxBid: Int)         extends AuctionReply
case class AuctionConcluded(seller: Actor, client: Actor)
                                            extends AuctionReply
case object AuctionFailed                   extends AuctionReply
case object AuctionOver                     extends AuctionReply
```

--- Listing 3.1: オークションサービスのメッセージクラス ---

各基底クラスごとに、クラス内の特定のメッセージ形式を定義する**ケースクラス**(case class)が複数あります。それらのメッセージが、小さな XML 文書に最終的にうまく対応できればよいのですが・・・。ここでは自動化ツールが存在すると仮定して、 XML 文書と、先のように定義された内部データ構造とを変換するとしましょう。

図 3.2 で示す Scala 実装、Auction クラスは、ある商品のオークションを調整する競売人アクターのためのものです。このクラスのオブジェクトは、次を指定して生成されます。

- オークション終了時に通知する必要のある売り手アクター
- 最低オークション価格
- オークションの終了予定時刻

アクターの振る舞いは act メソッドで定義されています。このメソッドでは 、TIMEOUT メッセージによってオークション終了が通知されるまで（receiveWithin を用いて）メッセージを選択し、それに対応することを繰り返します。最終的に停止する前まで、定数 timeToS

hutdown で定められた期間はアクティブであり続け、さらなる申し出に対してはオークションを締切った旨を返答します。

このプログラムで使われている構文について、さらにいくつか解説します。

- クラス Actor の receiveWithin メソッドはパラメータとして、 期間（ミリ秒単位）とメールボックスのメッセージを処理する関数をとります。 関数は一連のケースとして与えられ、各ケースはパターンと、 そのパターンに対応したメッセージを処理するアクションを指定します。 receiveWithin メソッドは、メールボックスからパターンにマッチする 最初のメッセージを選び、対応するアクションを適用します。
- receiveWithin の最後のケースは TIMEOUT パターンで守られています。 もし有効時間内にメッセージを受け取らなければ、receiveWithin メソッドの引数として渡された期間経過後に、このパターンが起動されます。 TIMEOUT は特別なメッセージで、Actor の実装そのものによって起動されます。
- 応答メッセージを送信する構文として、destnation ! SomeMessage（宛先 ! メッセージ）を用います。ここで ! は、 アクターとメッセージを引数とする二項演算子のように使われています。 これは Scala では、メソッド呼び出し destination.!(SomeMessage)、すなわち、 メッセージをパラメータとした destination アクターの ! メソッド呼び出しと同じです。

```
class Auction(seller: Actor, minBid: Int, closing: Date) extends Actor {
  val timeToShutdown = 36000000 // msec
  val bidIncrement = 10
  def act() {
    var maxBid = minBid - bidIncrement
    var maxBidder: Actor = null
    var running = true
    while (running) {
      receiveWithin ((closing.getTime() - new Date().getTime())) {
        case Offer(bid, client) =>
          if (bid >= maxBid + bidIncrement) {
            if (maxBid >= minBid) maxBidder ! BeatenOffer(bid)
            maxBid = bid; maxBidder = client; client ! BestOffer
          } else {
            client ! BeatenOffer(maxBid)
          }
        case Inquire(client) =>
          client ! Status(maxBid, closing)
        case TIMEOUT =>
          if (maxBid >= minBid) {
            val reply = AuctionConcluded(seller, maxBidder)
            maxBidder ! reply; seller ! reply
          } else {
            seller ! AuctionFailed
          }
          receiveWithin(timeToShutdown) {
            case Offer(_, client) => client ! AuctionOver
            case TIMEOUT => running = false
          }
      }
    }
  }
}
```

--- Listing 3.2: オークションサービスの実装 ---

ここまでの話は、Scala における分散プログラミングの雰囲気を示したものです。Scala にはアクタープロセス、メッセージ送受信、タイムアウトのあるプログラミングなどをサポートする豊富な言語構文があるように見えたかもしれません。実際は全く逆です。これまでに議論した言語構文はすべてクラス Actor のライブラリ内でメソッドとして提供されています。そのクラス自身が Scala で実装されており、基礎となるホスト言語（つまり Java や .NET）のスレッドモデルに基づいています。ここで使った Actor クラスの特徴すべての実装については、17.11 節で述べられています。

ライブラリベースのアプローチの利点は、相対的な、コア言語の簡潔さとライブラリ設計者への柔軟性にあります。コア言語では高レベルなプロセス間通信の詳細を規定する必要がないので、簡潔、一般性を保つことができます。メールボックス内メッセージのこのモデルは単なるライブラリモジュールなので、あるアプリケーションが異なるモデルを必要とする場合は、自由に変更できます。しかしながら、このアプローチはコア言語に対して、必要とされる言語の抽象化を使いやすい形で提供できるような十分な表現力を要求します。Scala はこのことを念頭においてデザインされました。Scala デザインの大きな目標の一つは、ライブラリモジュールによって実装される、ドメイン特化言語(DSL)ための使いやすいホスト言語たりうる、柔軟性にあります。たとえば先に示したアクター間通信の構文は、そのようなドメイン特化言語とみることができ、無意識に Scala コアを拡張しています。

4 式と簡単な関数

先の例は、Scala で何ができるかの印象を与えました。ここでは構文をより体系的に紹介します。最も簡単なレベルの、式と関数から始めます。

4.1 式と簡単な関数

Scala システムには、素敵な計算機と見ることのできるインタプリタが同梱されています。ユーザーは式をタイプすることで計算機と交信できます。計算機は評価した結果とその型を返します。たとえば

```
scala> 87 + 145
unnamed0: Int = 232

scala> 5 + 2 * 3
unnamed1: Int = 11

scala> "hello" + " world!"
unnamed2: java.lang.String = hello world!
```

部分式に名前を付け、以後は式の代わりにその名前を使うこともできます。

```
scala> def scale = 5
scale: Int

scala> 7 * scale
unnamed3: Int = 35

scala> def pi = 3.141592653589793
pi: Double

scala> def radius = 10
radius: Int

scala> 2 * pi * radius
unnamed4: Double = 62.83185307179586
```

定義は予約語 def で始まります。= 記号に続く式を表す名前を導入します。インタプリタは導入された名前と型を返します。

def x = e のような定義の実行は、式 e を評価しません。その代わり、x が使われる時はいつでも e は評価されます。一方で、Scala には値の定義 val x = e もあり、定義評価の一環として右辺 e を評価します。その後 x が使われるときは、前に計算された e の値で直ちに置き換えられので、式を再評価する必要はありません。

式はどのように評価されるのでしょう？ 演算子とオペランドから構成される式は次の簡単化のステップを繰り返し適用して評価されます。

・最も左の操作を選ぶ

- そのオペランドを評価する
- オペランド値に演算子を適用する

def で定義された名前は、名前を（未評価の）定義の右辺で置き換えることで評価されます。val で定義された名前は、名前を定義の右辺の値で置き換えることで評価されます。評価プロセスは値を得たら終了します。値は文字列や数や配列やリストといったデータアイテムです。

Example 4.1.1 数式の評価です。

```
   (2 * pi) * radius
→ (2 * 3.141592653589793) * radius
→ 6.283185307179586 * radius
→ 6.283185307179586 * 10
→ 62.83185307179586
```

式を値へと段階的に簡単にするプロセスを**簡約**と呼びます。

4.2 パラメータ

def を使って、パラメータを持った関数を定義できます。たとえば

```
scala> def square(x: Double) = x * x
square: (Double)Double

scala> square(2)
unnamed0: Double = 4.0

scala> square(5 + 3)
unnamed1: Double = 64.0

scala> square(square(4))
unnamed2: Double = 256.0

scala> def sumOfSquares(x: Double, y: Double) = square(x) + square(y)
sumOfSquares: (Double,Double)Double

scala> sumOfSquares(3, 2 + 2)
unnamed3: Double = 25.0
```

関数のパラメータは関数名の後に置かれ、常に括弧で囲まれます。各パラメータは型を伴い、型はパラメータ名とコロンに続いて示されます。現時点では、倍精度数の scala.Double 型のような基本的な数値型だけを必要とします。Scala では、いくつかの標準的な型について**型エイリアス**が定義されていて、数値型は Java と同じように書けます。たとえば double は scala.Double の型エイリアスであり、int は scala.Int の型エイリアスです。

パラメータをもつ関数は、式の演算子と同じように評価されます。はじめに関数の引数が（左から右の順序で）評価されます。つぎに関数適用が関数の右辺で置き換えられ、同時に関数のすべての形式上のパラメータが対応する実際の引数で置き換えられます。

Example 4.2.1

```
   sumOfSquares(3, 2+2)
→ sumOfSquares(3, 4)
→ square(3) + square(4)
→ 3 * 3 + square(4)
→ 9 + square(4)
→ 9 + 4 * 4
→ 9 + 16
→ 25
```

この例は、インタプリタが関数適用の前に、関数の引数を値に簡約することを示しています。この代わりに、簡約されていない引数に関数適用することも選択できます。それは次の簡約列をもたらします。

```
   sumOfSquares(3, 2+2)
→ square(3) + square(2+2)
→ 3 * 3 + square(2+2)
→ 9 + square(2+2)
→ 9 + (2+2) * (2+2)
→ 9 + 4 * (2+2)
→ 9 + 4 * 4
→ 9 + 16
→ 25
```

二つ目の評価順序は**名前渡し**(call-by-name)として、はじめの例は**値渡し**(call-by-value)として知られています。純粋な関数だけを使用する式や、したがって、置き換えモデルで簡約可能な式では、どちらの枠組みでも同じ値となります。

値渡しには、引数評価を繰り返さないという利点があります。名前渡しには、パラメータが関数で全く使用されない時に引数の評価を避けるという利点があります。ふつう、値渡しは名前渡しより効率的ですが、値渡し評価は、名前渡し評価なら終了する箇所でループするかもしれません。次を考えてみましょう。

```
scala> def loop: Int = loop
loop: Int

scala> def first(x: Int, y: Int) = x
first: (Int,Int)Int
```

すると、first(1, loop) は名前渡しでは 1 に簡約されますが、値渡しでは繰り返し自分自身へと簡約され、したがって評価は終了しません。

```
   first(1, loop)
→ first(1, loop)
→ first(1, loop)
→ ...
```

Scala はデフォルトでは値渡しを使いますが、パラメータ型の前に => が置かれた場合は名前渡し評価へと切り替えます。

Example 4.2.2

```
scala> def constOne(x: Int, y: => Int) = 1
constOne: (Int,=> Int)Int
```

```
scala> constOne(1, loop)
unnamed0: Int = 1

scala> constOne(loop, 2) // 無限ループとなる
^C                   // Ctrl-C で実行を停止
```

4.3 条件式

Scala の if-else は２つの選択肢から１つを選びます。構文は Java の if-else と似ています。しかし Java の if-else が文の選択にしか使えないのに対し、Scala は同じ構文を、２つの式の選択にも使えます。ですから Scala の if-else は Java の条件演算子 ... ? ... : ... の代わりに使えます。

Example 4.3.1

```
scala> def abs(x: Double) = if (x >= 0) x else -x
abs: (Double)Double
```

Scala のブール式は Java のものと似ています。定数 true と false、比較演算子、ブール値の否定 !、ブール値の演算子 && と || から構成されています。

4.4 例 : ニュートン法による平方根計算

ここまでに紹介した構文を、もう少し興味深いプログラムを組み立てて、示しましょう。課題は x の平方根を計算する関数

```
def sqrt(x: Double): Double = ...
```

を書くことです。

平方根を計算する一般的な方法は、近似を繰り返すニュートン法です。まず初期推定値 y（たとえば y=1）から始めます。次いで現在の推定値 y を、y と x/y の平均値を取って繰り返し改良します。次の例では √2 を近似する、推定値 y、商 x/y、その平均が３列に示されています。

```
1          2/1 = 2              1.5
1.5        2/1.5 = 1.3333       1.4167
1.4167     2/1.4167 = 1.4118    1.4142
1.4142     ...                  ...
y          x / y                ( y + x / y )/2
```

Scala では、このアルゴリズムを小さな関数群によって実装でき、各関数がアルゴリズムの各要素を表すようにできます。

はじめに、推定値から結果を得ることを繰り返す関数を定義します。

```
def sqrtIter(guess: Double, x: Double): Double =
  if (isGoodEnough(guess, x)) guess
  else sqrtIter(improve(guess, x), x)
```

sqrtIter は自分自身を再帰的に呼び出します。命令型プログラムのループは常に、関数型プログラムでは再帰でモデル化できます。

sqrtIter の定義には、パラメータ部に続いて戻り値型があることに注意して下さい。このような戻り値型は再帰関数では必須です。非再帰関数では戻り値型はオプションであり、もしそれがなければ、型チェッカーが関数の右辺から計算します。しかし非再帰関数であっても、よりよい文書化のために戻り値型を書いておくことは、しばしばよい考えです。

二つ目のステップとして、sqrtIter から呼ばれる２つの関数を定義します。推定値を改良する関数 improve と、終了テスト isGoodEnough です。定義は次ようになります。

```
def improve(guess: Double, x: Double) =
  (guess + x / guess) / 2

def isGoodEnough(guess: Double, x: Double) =
  abs(square(guess) - x) < 0.001
```

最後に、sqrt 関数自身を sqrtIter の適用として定義します。

```
def sqrt(x: Double) = sqrtIter(1.0, x)
```

演習 4.4.1 isGoodEnough の判定は小さな数に対してはあまり正確ではなく、大きな数に対しては終了しないかもしれません（何故でしょう？）。これらの問題のない別の isGoodEnough を設計しなさい。

演習 4.4.2 式 sqrt(4) の実行をトレースしなさい。

4.5 ネストした関数

関数型プログラミングスタイルでは、小さなヘルパー関数をたくさん作ることを推奨しています。先の例では、sqrt の実装はヘルパー関数 sqrtIter、improve、isGoodEnough を使っています。これらの関数の名前は sqrt の実装にだけ関係します。ふつう我々は、sqrt のユーザーがこれらの関数に直接アクセスすることを望みません。

そうすることを（そして名前空間の汚染を避けるのを)、ヘルパー関数を呼び出す関数自身の中へ入れることで強制できます。

```
def sqrt(x: Double) = {
  def sqrtIter(guess: Double, x: Double): Double =
    if (isGoodEnough(guess, x)) guess
    else sqrtIter(improve(guess, x), x)
  def improve(guess: Double, x: Double) =
    (guess + x / guess) / 2
  def isGoodEnough(guess: Double, x: Double) =
    abs(square(guess) - x) < 0.001
  sqrtIter(1.0, x)
```

```
}
```

このプログラムにおいて、中括弧 {...} は**ブロック**を囲みます。Scala のブロックはそれ自身が式です。各ブロックはその値を定義する結果式で終わります。結果式の前には補助的な定義があってもよく、それらはそのブロック内からしか見えません。

ブロック内の各定義は、後にセミコロンが続かねばならず、それによって後に続く定義や結果式と分離されます。しかし、次の条件の何れかが真でなければ、各行の最後にセミコロンが暗黙のうちに挿入されます。

1. 問題となる行がピリオドのような単語や、 式の最後として正しくない中置演算子で終わる場合。
2. あるいは次の行が、式のはじまりとならないような単語で始まる場合。
3. あるいは括弧（...）または、角括弧 [...] の内側にいる場合（これらの中に複数の文を入れることはできません)。

したがって次は正しいです。

```
def f(x: Int) = x + 1;
f(1) + f(2)

def g1(x: Int) = x + 1
g(1) + g(2)

def g2(x: Int) = {x + 1}; /* ';' は必須 */ g2(1) + g2(2)
def h1(x) =
  x +
  y
  h1(1) * h1(2)

def h2(x: Int) = (
  x    // 括弧は必須。
  + y  // 括弧がないと 'x' の後にセミコロンが挿入される。
)
h2(1) / h2(2)
```

Scala は通常のブロック構造のスコープ規則を用いています。外側のブロックで定義された名前は、そこで再定義されない限り、内側のブロックからも見えます。この規則によって sqrt の例を簡略化できます。ネストさせた関数の追加パラメータとして x を渡す必要はありません。なぜなら外側の関数 sqrt のパラメータは常に見えるからです。次が簡略化されたコードです。

```
def sqrt(x: Double) = {
  def sqrtIter(guess: Double): Double =
    if (isGoodEnough(guess)) guess
    else sqrtIter(improve(guess))
  def improve(guess: Double) =
    (guess + x / guess) / 2
  def isGoodEnough(guess: Double) =
    abs(square(guess) - x) < 0.001
  sqrtIter(1.0)
}
```

4.6 末尾再帰

与えられた２つの数の最大公約数を計算する、次の関数について考えてみましょう。

```
def gcd(a: Int, b: Int): Int = if (b == 0) a else gcd(b, a % b)
```

関数評価の置き換えモデルを用いると、gcd(14, 21) は次のように評価されます。

```
       gcd(14, 21)
→     if (21 == 0) 14 else gcd(21, 14 % 21)
→     if (false) 14 else gcd(21, 14 % 21)
→     gcd(21, 14 % 21)
→     gcd(21, 14)
→     if (14 == 0) 21 else gcd(14, 21 % 14)
→ →  gcd(14, 21 % 14)
→     gcd(14, 7)
→     if (7 == 0) 14 else gcd(7, 14 % 7)
→ →  gcd(7, 14 % 7)
→     gcd(7, 0)
→     if (0 == 0) 7 else gcd(0, 7 % 0)
→ →  7
```

もう一つの再帰関数 factorial の評価と比較して下さい。

```
def factorial(n: Int): Int = if (n == 0) 1 else n * factorial(n - 1)
```

factorial(5) の適用は次のように書き換えます。

```
            factorial(5)
→          if (5 == 0) 1 else 5 * factorial(5 - 1)
→          5 * factorial(5 - 1)
→          5 * factorial(4)
→ . . . → 5 * (4 * factorial(3))
→ . . . → 5 * (4 * (3 * factorial(2)))
→ . . . → 5 * (4 * (3 * (2 * factorial(1))))
→ . . . → 5 * (4 * (3 * (2 * (1 * factorial(0)))))
→ . . . → 5 * (4 * (3 * (2 * (1 * 1))))
→ . . . → 120
```

２つの書き換え列には重要な違いがあります。gcd の書き換え列の項は、同じ形を繰り返しています。評価が進んでもその大きさは一定に抑えられています。対照的に factorial の評価では、オペランドの列はどんどん長くなり、評価列の最後でそれらは掛け合わされます。

Scala の実際の実装が項書き換えで行われていないとしても、やはり、書き換え列と同様の空間的な振る舞いをします。gcd の実装をみると、gcd の再帰呼び出しは評価本体の最後のアクションであることに気付きます。gcd は「**末尾再帰的**」だとも呼ばれます。末尾再帰関数の最後の呼び出しは、関数の先頭への戻りジャンプとして実装できます。その呼び出しの引数は、gcd の現在のインスタンスのパラメータを上書きできるので、新しいスタック空間を必要としません。ですから末尾再帰関数は繰り返し処理であり、一定の空間で実行できます。

対照的に factorial の再帰呼び出しでは、掛け算があとで施されます。ですから新しいスタックフレームが factorial の再帰インスタンス用に割り当てられ、インスタンス終了後に割り当てが除かれます。この階乗関数の設計は末尾再帰的ではありません。実行するには

入力パラメータに比例した空間が必要となります。

より一般的には、もし関数の最後のアクションが他の（同じでも可）関数の呼び出しであれば、両方の関数は一つのスタックフレームだけで充分です。そのような呼び出しは「末尾呼び出し(tail call)」と呼ばれます。原理的には、末尾呼び出しは常に呼び出した関数のスタックフレームを再利用できます。しかし（Java VM のような）いくつかの実行時環境では、末尾呼び出しでのスタックフレーム再利用を効率化するプリミティブに欠けています。ですから製品品質の Scala の実装では、直接的な末尾再帰(最後のアクションが自分自身の呼び出しとなる再帰)のスタックフレーム再利用だけが要求されています。他の末尾呼び出しも最適化されるかもしれませんが、多くの実装でそうなるということは、当てにすべきではありません。

演習 4.6.1 factorial の末尾再帰版をデザインしなさい。

5 第一級の関数

Scala の関数は「**第一級の値**」です。他の値と同じように、関数のパラメータとして渡したり結果として返したりできます。他の関数をパラメータとしてとったり、関数を結果として返す関数は、**高階関数**と呼ばれます。この章では高階関数を紹介し、それがプログラム作成にどれほど柔軟なメカニズムを与えるか示します。

興味をそそる例として、次の３つの関連した課題を考えます。

1. ２つのあたえられた数 a と b の間にある、すべての整数の和を求める関数を書きなさい。

```
def sumInts(a: Int, b: Int): Int =
  if (a > b) 0 else a + sumInts(a + 1, b)
```

2. ２つのあたえられた数 a と b の間にある、すべての整数の２乗の和を求める関数を書きなさい。

```
def square(x: Int): Int = x * x
def sumSquares(a: Int, b: Int): Int =
  if (a > b) 0 else square(a) + sumSquares(a + 1, b)
```

3. ２つのあたえられた数 a と b の間にある、すべての整数 n について 2^nの和を求める関数を書きなさい。

```
def powerOfTwo(x: Int): Int = if (x == 0) 1 else 2 * powerOfTwo(x - 1)
def sumPowersOfTwo(a: Int, b: Int): Int =
  if (a > b) 0 else powerOfTwo(a) + sumPowersOfTwo(a + 1, b)
```

これらの関数はすべて、異なる値 f に対する $\Sigma_a^b f(n)$ のインスタンスです。 関数 sum を定義して、共通パターンを括り出せます。

```
def sum(f: Int => Int, a: Int, b: Int): Int =
  if (a > b) 0 else f(a) + sum(f, a + 1, b)
```

型 Int => Int は、型 Int の引数をとって、型 Int の結果を返す関数の型です。したがって sum は、他の関数をパラメータとしてとる関数です。別の言葉で言えば、sum は**高階関数**です。

sum を使って、和を求める３つの関数を次のように書けます。

```
def sumInts(a: Int, b: Int): Int = sum(id, a, b)
def sumSquares(a: Int, b: Int): Int = sum(square, a, b)
def sumPowersOfTwo(a: Int, b: Int): Int = sum(powerOfTwo, a, b)
```

ただし、

```
def id(x: Int): Int = x
def square(x: Int): Int = x * x
def powerOfTwo(x: Int): Int = if (x == 0) 1 else 2 * powerOfTwo(x - 1)
```

5.1 無名関数

関数をパラメータとすることで、多くの小さな関数が作られます。前の例では id, square, powerOfTwo をそれぞれ別個の関数として定義し、sum に引数として渡せました。

引数用の小さな関数のために名前付きの関数定義を使う代わりに、それらを無名関数として短く書けます。**無名関数**は関数に評価される式で、関数に名前を付けずに定義できます。例として、２乗する無名関数を考えます。

```
(x: Int) => x * x
```

矢印 '=>' の前の部分は関数のパラメータで、'=>' に続く部分が本体です。たとえば、２つの引数を掛け合わせる無名関数は次のようになります。

```
(x: Int, y: Int) => x * y
```

無名関数により名前付きの外部関数を使わずに、先ほどの最初の２つの、和を求める関数を書けます。

```
def sumInts(a: Int, b: Int): Int = sum((x: Int) => x, a, b)
def sumSquares(a: Int, b: Int): Int = sum((x: Int) => x * x, a, b)
```

しばしば、Scala コンパイラは無名関数の文脈からそのパラメータの型を推論でき、その場合はパラメータの型を省略できます。たとえば sumInts や sumSquares では、sum の型から最初のパラメータは Int => Int 型の関数に違いないと分かります。ですから、パラメータの型 Int は冗長であり省略できます。もし、型指定のない、ただ１つのパラメータしかない場合は、その周りの括弧も省略できます。

```
def sumInts(a: Int, b: Int): Int = sum(x => x, a, b)
def sumSquares(a: Int, b: Int): Int = sum(x => x * x, a, b)
```

一般的に Scala の項 (x1:T1, ..., xn:Tn) => E は、パラメータ x1, ..., xn を式 E (E は x1, ..., xn を参照しているでしょう) の結果に対応させる関数を定義します。無名関数は Scala に必須の構文ではありません。なぜなら常に名前付き関数で表現できるからです。実際、無名関数

```
(x1 : T1, ..., xn : Tn ) => E
```

はブロック

```
{ def f (x1 : T1, ..., xn : Tn ) = E ; f _ }
```

と等価です。ただし f はプログラムのどこか他で使われていない新規な名前です。無名関数は「**糖衣構文**」であるとも言います。

5.2 カリー化

和を求める関数の最後の設計はすでにかなり簡潔です。しかし更に可能です。a と b がすべての関数でパラメータまたは引数として現れる一方で、興味深い組み合わせを成していないようです。それらを除去できるでしょうか。

sum を書き直し、境界 a と b をパラメータにしないようにしましょう。

```
def sum(f: Int => Int): (Int, Int) => Int = {
  def sumF(a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sumF(a + 1, b)
  sumF
}
```

この設計では、sum は他の関数、すなわち特別な和を求める関数 sumF を返す関数です。後者の関数がすべての仕事をします。境界 a と b をパラメータとしてとり、sum 関数のパラメータ f をその間のすべての整数に適用し、結果を足し合わせます。

新設計の sum を使って、次のように定義できます。

```
def sumInts = sum(x => x)
def sumSquares = sum(x => x * x)
def sumPowersOfTwo = sum(powerOfTwo)
```

あるいは同等に、値定義を使って

```
val sumInts = sum(x => x)
val sumSquares = sum(x => x * x)
val sumPowersOfTwo = sum(powerOfTwo)
```

sumInts, sumSquares, sumPowersOfTwo は、他の関数と同じように適用できます。たとえば

```
scala> sumSquares(1, 10) + sumPowersOfTwo(10, 20)
unnamed0: Int = 267632001
```

関数を返す関数はどのように適用されるのでしょうか？例として式

```
sum(x => x * x)(1, 10) ,
```

の関数 sum は２乗する関数（x => x * x）に適用されます。そして結果の関数は二番目の引数リスト（1, 10）に適用されます。

関数適用が左結合のため、この記法が可能です。すなわち、もし args1 と args2 が引数リストなら、

```
f(args 1)(args2) は (f(args1))(args2) と等価です。
```

この例では sum(x => x * x)(1, 10) は、式 (sum(x => x * x))(1, 10) と等価です。

関数を返す関数のスタイルはたいへん有用なので、Scala にはそのための特別な構文があります。たとえば次の sum 定義は前のものと等価ですが短くなっています。

```
def  sum(f: Int => Int)(a: Int, b: Int): Int =
  if (a > b) 0 else f(a) + sum(f)(a + 1, b)
```

一般に、カリー化された関数定義

```
def  f (args1) ... (argsn) = E
```

ただし n>1、は次のように展開されます。

```
def  f (args1) ... (argsn-1) = { def g (argsn) = E ; g }
```

ただし g は未使用の識別子です。あるいはもっと短く無名関数を使って

```
def  f (args1) ... (argsn-1) = ( argsn) => E .
```

このステップを n 回繰り返すと次を得ます。

```
def  f (args1) ... (argsn) = E
```

これは次と等価です。

```
 def f = (args1) => ... => (argsn) => E .
```

あるいは同様に値定義を用いて

```
val  f = (args) => ... => (args) => E .
```

この関数定義と適用のスタイルは、創始者である20世紀の論理学者 Haskell B. Curry に因んで**カリー化**と呼ばれています。しかしそのアイデアは Moses Schoenfinkel や Gottlob Frege にまで遡ります。

関数を返す関数の型は、パラメータリストに似た形で表されます。例として sum の最後の設計を見ると、sum の型は (Int => Int) => (Int, Int) => Int です。関数型は右結合なため、これが可能です。すなわち

```
T1 => T2 => T3 は  T1 => (T2 => T3) と等価
```

演習 5.2.1 関数 sum は線形再帰を使ってます。?? の部分を埋めて末尾再帰に書けますか？

```
def sum(f: Int => Double)(a: Int, b: Int): Double = {
  def iter(a, result) = {
    if (??) ??
    else iter(??, ??)
  }
  iter(??, ??)
}
```

演習 5.2.2 与えられた範囲の点の、関数値の積を求める関数 product を書きなさい。

演習 5.2.3 factorial を product を用いて書きなさい。

演習 5.2.4 sum と product の両方を一般化する、より汎用的な関数を書けますか？

5.3 例 : 関数の不動点探索

数 x は次の条件を満たす時に、関数 f の**不動点**と呼ばれます。

```
f(x) = x .
```

いくつかの関数 f では、初期推定値から始めて、値が変化しなくなる（あるいは変化が小さな許容量以下になる）まで f を繰り返し適用することで不動点を求めることができます。それは、数列

```
x, f(x), f(f(x)), f(f(f(x))), ...
```

が、f の不動点に収束すれば可能です。このアイデアを取り入れたのが次の「不動点を求める関数」です。

```
val tolerance = 0.0001
def isCloseEnough(x: Double, y: Double) = abs((x - y) / x) < tolerance
def fixedPoint(f: Double => Double)(firstGuess: Double) = {
  def iterate(guess: Double): Double = {
    val next = f(guess)
    if (isCloseEnough(guess, next)) next
    else iterate(next)
  }
  iterate(firstGuess)
}
```

このアイデアを平方根を求める関数の再設計に応用します。sqrt の仕様から始めましょう。

```
sqrt(x) = y ただし y * y = x
        = y ただし y = x / y
```

したがって sqrt(x) は関数 y=> x / y の不動点です。このことは sqrt(x) は不動点の反復で計算できることを示しています。

```
def sqrt(x: double) = fixedPoint(y => x / y)(1.0)
```

しかし実際にやってみると、計算は収束しないことが分かります。現在の推定値を追跡する print 文付きの不動点関数を実装してみます。

```
def fixedPoint(f: Double => Double)(firstGuess: Double) = {
  def iterate(guess: Double): Double = {
    val next = f(guess)
    println(next)
    if (isCloseEnough(guess, next)) next
    else iterate(next)
  }
  iterate(firstGuess)
}
```

すると sqrt(2) は次のようになります。

```
2.0
1.0
2.0
1.0
2.0
...
```

このような振動を抑える一つの方法は、推定値の急激な変化を防ぐことです。これは元の数列上で、続く値を**平均**すれば可能です。

```
scala> def sqrt(x: Double) = fixedPoint(y => (y + x/y) / 2)(1.0)
sqrt: (Double)Double

scala> sqrt(2.0)
  1.5
  1.4166666666666665
  1.4142156862745097
  1.4142135623746899
  1.4142135623746899
```

実際、fixedPoint 関数を展開すると、前に 4.4 節で定義した不動点の定義と全く同じになります。

前の例では、もし関数を引数として渡すことができれば、言語の表現力が大幅に強化される、ということを示しました。次の例では、関数を返す関数もたいへん有用であることを示します。

不動点の繰り返しをもう一度考えてみましょう。√(x) が関数 y => x / y の不動点であることから始めました。次いで、続く値を平均することで繰り返しを収束させました。この**平均による減衰**というテクニックは一般的ですから、別の関数で包むことにします。

```
def averageDamp(f: Double => Double)(x: Double) = (x + f(x)) / 2
```

averageDump を使って、平方根関数を次のように再設計できます。

```
def sqrt(x: Double) = fixedPoint(averageDamp(y => x/y))(1.0)
```

Exercise 5.3.1 Write a function for cube roots using fixedPoint and averageDamp.

これはアルゴリズムの要素を可能な限り明確に表現しています。

演習 5.3.1 fixedPoint と averageDump を使って、立方根を求める関数を書きなさい。

5.4 まとめ

前の章で、関数は抽象化に必須である、なぜなら計算の一般的な方法を明示できるから、ということ、関数は我々のプログラミング言語において名前の付いた要素であることを見てきました。この章では、それら抽象化は高階関数と組み合わせることで、さらに抽象化できることを示しました。プログラマは抽象化と再利用の機会を探すべきです。最大限に抽象化することが必ずしも最適とは限りませんが、適切な場面で抽象化できるように、抽象化の技法を知ることは重要です。

5.5 ここまでの構文

第 4 章と第 5 章では、プリミティブなデータと関数の、式と型を表現する Scalaの構文を扱いました。下記に、それらの文脈自由文法を拡張 Backus-Naur 記法で与えます。'|' は選択を、[...] はオプション（０あるいは１回の出現)、{...} は繰り返し（０回以上の出現)を表します。

文字:

Scala プログラムは（Unicode）文字の列です。次の文字セットを区別します。

- 空白文字。たとえば ' '、タブ、改行文字
- 文字。'a'-'z'、'A'-'Z'
- 数字。'0'-'9'
- 区切り文字。 . , ; ( ) { } [ ] " '
- 演算子文字。たとえば '#' '+' ':'。 正確には、上記のどれにも含まれない印刷可能な文字。

語彙素:

```
ident   = letter {letter | digit}
        | operator { operator }
        | ident '_' ident
literal = "as in Java"
```

リテラルは Java と同じです。数、文字、文字列、ブール値を定義します。リテラルの例は、0、1.0e10、'x'、"he said "hi!""、true です。

識別子には２つの形式があります。文字で始まり、文字またはシンボルの列（空を許す）が続くものか、演算子文字で始まり演算子文字の列（空を許す）が続くものです。どちらの形式もアンダースコア '_' を含んで構いません。さらに、アンダースコア文字列の後には、どちらの形式の識別子が続いても構いません。したがって、次はすべて正しい識別子です。

```
x   Room10a   +   --   foldl_:   +_vector
```

この規則から、後に続く演算子識別子は空白文字で区切られねばならないということが判ります。たとえば入力 x+-y は３つのトークン列 x, +-, y へと字句解析されます。もし x と、 y の符号を逆にした値との和を表現したいなら、x+ -y のように、最低一つの空白を加える必要があります。

文字 $ はコンパイラ生成識別子用に予約されています。ソースプログラムで使うべきではありません。

以下は予約語です。識別子としては使用できません。

```
abstract case catch class def
do else extends false final
finally for if implicit import
match new null object override
package private protected requires return
sealed super this throw trait
try true type val var
while with yield
_     :    =    =>    <-    <:    <%   >:    #    @
```

型 (Types):

```
Type         = SimpleType | FunctionType
FunctionType = SimpleType '=>' Type | '(' [Types] ')' '=>' Type
SimpleType   = Byte | Short | Char | Int | Long | Float | Double |
               Boolean | Unit | String
Types        = Type {',' Type}
```

型は

- 数値型 Byte、Short、Char、Int、Long、Float、Double (Java と同じ)
- Boolean 型とその値の true と false
- Unit 型とその唯一の値 ()
- String 型
- 関数型、たとえば (Int, Int) => Int や String => Int => String

式 (Expressions):

```
Expr          = InfixExpr | FunctionExpr | if '(' Expr ')' Expr else Expr
InfixExpr     = PrefixExpr | InfixExpr Operator InfixExpr
Operator      = ident
PrefixExpr    = ['+' | '-' | '!' | '~' ] SimpleExpr
SimpleExpr    = ident | literal | SimpleExpr '.' ident | Block
FunctionExpr  = (Bindings | Id) '=>' Expr
Bindings      = '(' Binding {',' Binding} ')'
Binding       = ident [':' Type]
Block         = '{' {Def ';'} Expr '}'
```

式は

- 識別子、たとえば x、isGoodEnugh、*、+-
- リテラル、たとえば 0、1.0、"abc"
- フィールドとメソッド選択、たとえば System.out.println
- 関数適用、たとえば sqrt(x)
- 演算子適用、たとえば -x、y + x
- 条件式、たとえば if (x < 0) -x else x
- ブロック、たとえば { val x = abs(y); x * 2}
- 無名関数、たとえば x => x + 1、(x: Int, y: Int) = x + y

定義 (Definitions):

```
Def        = FunDef | ValDef
FunDef     = 'def' ident {'(' [Parameters] ')'} [':' Type] '=' Expr
ValDef     = 'val' ident [':' Type] '=' Expr
Parameters = Parameter {',' Parameter}
Parameter  = ident ':' ['=>'] Type
```

定義は

- 関数定義、たとえば `def square(x: Int): Int = x * x`
- 値定義、たとえば `val y = square(2)`

6 クラスとオブジェクト

Scala には組み込みの有理数型はありませんが、クラスを使って簡単に定義できます。次が実装例です。

```
class Rational(n: Int, d: Int) {
  private def gcd(x: Int, y: Int): Int = {
    if (x == 0) y
    else if (x < 0) gcd(-x, y)
    else if (y < 0) -gcd(x, -y)
    else gcd(y % x, x)
  }
  private val g = gcd(n, d)
  val numer: Int = n/g
  val denom: Int = d/g
  def +(that: Rational) =
    new Rational(numer * that.denom + that.numer * denom,
    denom * that.denom)
  def -(that: Rational) =
  new Rational(numer * that.denom - that.numer * denom,
    denom * that.denom)
  def *(that: Rational) =
    new Rational(numer * that.numer, denom * that.denom)
  def /(that: Rational) =
    new Rational(numer * that.denom, denom * that.numer)
}
```

有理数を、分子 n と分母 d の２つのコンストラクタ引数をとるクラスとして定義しています。このクラスは有理数上の演算メソッドだけではなく、分子と分母を返すフィールドも提供します。各演算メソッドは操作の右オペランドをパラメータにとります。操作の左オペランドは、常にそのメソッドがメンバであるような有理数です。

プライベート・メンバ 有理数の実装では、２つの整数の最大公約数を計算するプライベートなメソッド gcd を定義し、また、コンストラクタ引数の gcd を格納するプライベートなフィールド g も定義します。これらのメンバはクラス Rational の外からはアクセスできません。それらはクラスの実装において、コンストラクタ引数を公約数で割って、分子と分母を常に正規形(約分された形)にするために使います。

オブジェクト生成とアクセス 有理数の使い方の例として、i の範囲が 1 から 10 まで時に、1/i の和を印刷するプログラムを示します。

```
var i = 1
var x = new Rational(0, 1)
while (i <= 10) {
  x += new Rational(1, i)
  i += 1
}
println("" + x.numer + "/" + x.denom)
```

+ は左オペランドに文字列を、右オペランドに任意の型の値をとります。右オペランドを文字列に変換し、それを左オペランドに追加した結果を返します。

継承とオーバーライド Scala のすべてのクラスは親クラスを持ち、それを継承しています。クラス定義の中で親クラスが指定されていない場合、ルート型 scala.AnyRef が暗黙のうちに仮定されます（Java の実装でいうと、この型は java.lang.Object のエイリアスです）。 たとえばクラス Rational を次のように定義しても同じことです。

```
class Rational(n: Int, d: Int) extends AnyRef {
  ... // as before
}
```

クラスは親クラスのメンバをすべて継承します。継承したメンバを再定義（つまり、**オーバーライド**）できます。たとえばクラス java.lang.Object は、オブジェクトの文字列表現を返すメソッド toString を定義しています。

```
class Object {
  ...
  def toString: String = ...
}
```

Object での toString は、オブジェクトのクラス名と数からなる文字列として実装されています。このメソッドを有理数のオブジェクト用に再定義すると役に立ちます。

```
class Rational(n: Int, d: Int) extends AnyRef {
  ... // as before
  override def toString = "" + numer + "/" + denom
}
```

注意すべき点は、Java とは異なり、定義の再定義では手前に override 修飾子が必要なことです。

もしクラス A がクラス B を継承しているなら、型 A のオブジェクトは、型 B のオブジェクトが期待される所ではどこでも使用できます。このような場合、型 A は型 B に**適合する**(conform)、といいます。たとえば Rational は AnyRef に適合します。したがって Rational 値を AnyRef 型の変数に代入できます。

```
var x: AnyRef = new Rational(1, 2)
```

パラメータなしのメソッド Java とは異なり、Scala のメソッドはパラメータリストを必ずしも必要としません。次の square メソッドがその例です。このメソッドは単に名前を書くだけで呼び出されます。

```
class Rational(n: Int, d: Int) extends AnyRef {
  ... // as before
  def square = new Rational(numer*numer, denom*denom)
}
val r = new Rational(3, 4)
println(r.square) // prints''9/16''*
```

これは、パラメータなしメソッドは numer のような値フィールドと同じようにアクセスされるということです。値とパラメータなしメソッドの違いはそれらの定義にあります。値の

右辺はオブジェクトが作成された時に評価され、以後は値が変化しません。その一方で、パラメータなしメソッドの右辺はメソッドが呼ばれる度に評価されます。フィールドとパラメータなしメソッドのアクセス方法が統一されているのでクラス実装者の自由度が増します。しばしば、あるバージョンではフィールドだったものが次のバージョンでは計算された値だったりします。統一的なアクセスのおかげで、クライアントは変更による書き直しが不要になります。

抽象クラス 整数の集合に対して、２つの操作 `incl` と `contains` を持つクラスを書く課題について考えてみましょう。(`s incl x`) は集合 `s` の要素すべてと、要素 `x` を持つ新しい集合を返すべきです。(`s contains x`) は、集合 `s` が要素 `x` を含む時は `true` を、さもなければ `false` を返すべきです。そのような集合のインターフェイスは次のようになります。

```
abstract class IntSet {
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean
}
```

IntSet は**抽象クラス**と分類されます。これは２つの結果をもたらします。一つは、抽象クラスは**延期された**メンバを持ち、それらは宣言はあっても実装はありません。この場合、`incl` と `contains` の両方がそのようなメンバです。二つ目は、抽象クラスには未実装のメンバがあるかもしれないので、そのクラスのオブジェクトを `new` で生成できません。その一方で、抽象クラスは他のクラスの基底クラスとして使うことができ、継承したクラスでは延期されたメンバを実装します。

トレイト Scala では、`abstract class` の代わりにキーワード `trait` をよく使います。トレイトは、他のクラスに付け加えるための抽象クラスです。これはトレイトが未知の親クラスにメソッドやフィールドをいくつか追加するからです。たとえば、トレイト `Bordered` は様々なグラフィカルコンポーネントに縁を追加するのに使われるでしょう。別の使い方としては、様々なクラスによって提供される機能のシグネチャをトレイトが集約する、という Java のインターフェイスがするようなやり方です。

IntSet はこの部類に入るので、トレイトとしても定義できます。

```
trait IntSet {
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean
}
```

抽象クラスの実装 たとえば集合を二分木で実装することにしましょう。木の形には２つの可能性があります。空集合を表す木と、整数１つと２つの部分木からなる木です。次がその実装です。

```
class EmptySet extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = new NonEmptySet(x, new EmptySet, new EmptySet)
}

class NonEmptySet(elem: Int, left: IntSet, right: IntSet) extends IntSet {
  def contains(x: Int): Boolean =
    if (x < elem) left contains x
```

```
    else if (x > elem) right contains x
    else true
  def incl(x: Int): IntSet =
    if (x < elem) new NonEmptySet(elem, left incl x, right)
    else if (x > elem) new NonEmptySet(elem, left, right incl x)
    else this
}
```

EmptySet と NonEmptySet の両方とも IntSet を継承します。これは、型 EmptySet と NonEmptySet が型 IntSet に適合すること --- 型 EmptySet や NonEmptySet の値は、型 IntSet の値が必要なところではどこでも使えることを意味します。

演習 6.0.1 メソッド union と intersection を２つの集合の結びと交わりとなるように書きなさい。

演習 6.0.2 要素 x を取り除いた集合を返すメソッド

```
def excl(x: Int)
```

を追加しなさい。そうするために便利なテスト用メソッド

```
def isEmpty: Boolean
```

も、集合に対して実装しなさい。

動的束縛（Scala を含む）オブジェクト指向言語はメソッド呼び出しのときに**動的ディスパッチ**を用います。これは、メソッド呼び出しで起動されるコードは、メソッドを含むオブジェクトの実行時型に依存するということです。たとえば式 s contains 7、ただし s は宣言された型 s: IntSet の値だとします。もしそれが EmptySet の値なら、実行されるのはクラス EmptySet の contains の実装であり、NonEmptySet の値の場合も同様です。この振る舞いは評価の置き換えモデルの直接的な帰結です。たとえば

```
  (new EmptySet).contains(7)

->      (クラス EmptySet の contains 本体で置き換えることで)

  false
```

あるいは

```
  new NonEmptySet(7, new EmptySet, new EmptySet).contains(1)

->      (クラス NonEmptySet の contains 本体で置き換えることで)

  if (1 < 7) new EmptySet contains 1
  else if (1 > 7) new EmptySet contains 1
  else true

->      (条件式を書き換えて)

  new EmptySet contains 1
```

```
->     (クラス EmptySet の contains 本体で置き換えることで)

  false .
```

動的メソッドディスパッチは高階関数呼び出しと似ています。どちらの場合にも、実行されるコードの正体は実行時にしか分かりません。この類似性は表面上のものにとどまりません。実際、Scala はすべての関数値をオブジェクトとして表現しています（8.6 節参照)。

オブジェクト 整数集合の以前の実装では、空集合は new EmptySet で表現されていました。つまり空集合値が必要になる度に、毎回新しいオブジェクトが生成されました。値 empty を一度だけ定義して、その値をすべての new EmptySet の出現の代わりに使えば、不必要なオブジェクト生成を避けることができます。たとえば

```
val EmptySetVal = new EmptySet
```

この方法の問題の一つは、このような値定義は Scala の正しいトップレベルでの定義ではない、ということです。これは別のクラスかオブジェクトの一部でなくてはなりません。またクラス EmptySet の定義は少しばかりやり過ぎています。もしたった一つのオブジェクトにだけ関心があるなら、なぜオブジェクト用のクラスを定義するのでしょう？ より直接的な方法は**オブジェクト定義**を使うことです。代わりの、より洗練された空集合の定義は次のようになります。

```
object EmptySet extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = new NonEmptySet(x, EmptySet, EmptySet)
}
```

オブジェクト定義の構文はクラス定義の構文に従います。オプションの extends 節とオプションの本体があります。クラスの場合のように、extends 節はオブジェクトが継承するメンバを定義し、一方で本体はオーバーライドするあるいは新規のメンバを定義します。しかしオブジェクト定義はただ１つのオブジェクトだけを定義するため、同じ構造を持った他のオブジェクトを new で生成できません。したがってオブジェクト定義には、クラス定義にはあるかもしれないコンストラクタパラメータはありません。

オブジェクト定義は Scala プログラムのどこにでも、トップレベルにも、書けます。Scala ではトップレベルのエンティティの実行順序は固定されていないため、オブジェクト定義によって定義されたオブジェクトがいつ生成され初期化されるのか、正確に知りたい人もいるでしょう。その答えは、オブジェクトはそのメンバが最初にアクセスされる時に生成される、というものです。この戦略は**遅延評価**と呼ばれています。

標準クラス Scala は純粋なオブジェクト指向言語です。これは Scala のすべての値がオブジェクトである、ということです。実際 int や boolean のようなプリミティブ型でさえ特別扱いされません。それらはモジュール Predef において、Scala クラスの型エイリアスとして定義されています。

```
type boolean = scala.Boolean
type int = scala.Int
type long = scala.Long
...
```

コンパイラは効率化のために通常、scala.Int 型の値を 32 bit 整数で、scala.Boolean 型の値を Java の boolean で、等と表現します。しかし必要に応じてこれらの特別な表現をオブジェクトに変換します。たとえばプリミティブの Int 値が AnyRef 型のパラメータを

とる関数に渡される時です。したがってプリミティブ値の特別な表現は単に最適化のためであり、プログラムの意味を変えません。

次がクラス Boolean の仕様です。

```
package scala
abstract class Boolean {
  def && (x: => Boolean): Boolean
  def || (x: => Boolean): Boolean
  def !                 : Boolean
  def == (x: Boolean)   : Boolean
  def != (x: Boolean)   : Boolean
  def < (x: Boolean)    : Boolean
  def > (x: Boolean)    : Boolean
  def <= (x: Boolean)   : Boolean
  def >= (x: Boolean)   : Boolean
}
```

Boolean はクラスとオブジェクトだけを使って定義でき、組み込みのブール値や数値の型を参照する必要はありません。Boolean 型の一つの実装を次に示します。これは標準 Scala ライブラリの実際の実装ではありません。効率化のために標準の実装では、組み込みのブール値を使います。

```
package scala
abstract class Boolean {
  def ifThenElse(thenpart: => Boolean, elsepart: => Boolean)

  def && (x: => Boolean): Boolean = ifThenElse(x, false)
  def || (x: => Boolean): Boolean = ifThenElse(true, x)
  def !                 : Boolean = ifThenElse(false, true)

  def == (x: Boolean)   : Boolean = ifThenElse(x, x.!)
  def != (x: Boolean)   : Boolean = ifThenElse(x.!, x)
  def < (x: Boolean)    : Boolean = ifThenElse(false, x)
  def > (x: Boolean)    : Boolean = ifThenElse(x.!, false)
  def <= (x: Boolean)   : Boolean = ifThenElse(x, true)
  def >= (x: Boolean)   : Boolean = ifThenElse(true, x.!)
}
case object True extends Boolean {
  def ifThenElse(t: => Boolean, e: => Boolean) = t
}
case object False extends Boolean {
  def ifThenElse(t: => Boolean, e: => Boolean) = e
}
```

次はクラス Int の仕様の一部です。

```
package scala
abstract class Int extends AnyVal {
  def toLong: Long
  def toFloat: Float
  def toDouble: Double

  def + (that: Double): Double
  def + (that: Float): Float
```

```
  def + (that: Long): Long
  def + (that: Int): Int          // analogous for -, *, /, %

  def << (cnt: Int): Int          // analogous for >>, >>>

  def & (that: Long): Long
  def & (that: Int): Int          // analogous for |, ^

  def == (that: Double): Boolean
  def == (that: Float): Boolean
  def == (that: Long): Boolean    // analogous for !=, <, >, <=, >=
}
```

クラス Int も原理的にはオブジェクトとクラスだけで、組み込みの整数型を参照することなく実装できます。その方法を知るために、少しだけ簡単な問題、自然数を表す型 Nat の実装方法を考えみましょう。次は抽象クラス Nat の定義です。

```
abstract class Nat {
  def isZero: Boolean
  def predecessor: Nat
  def successor: Nat
  def + (that: Nat): Nat
  def - (that: Nat): Nat
}
```

クラス Nat の操作を実装するために、子オブジェクト Zero と子クラス Succ（次の数）を定義します。それぞれの数 N は、Zero に Succ コンストラクタを N 回適用したものです。

```
new Succ( ... new Succ(Zero) ... )
<------- N 回 ------->
```

オブジェクト Zero の定義はそのままです。

```
object Zero extends Nat {
  def isZero: Boolean = true
  def predecessor: Nat = error("negative number")
  def successor: Nat = new Succ(Zero)
  def + (that: Nat): Nat = that
  def - (that: Nat): Nat = if (that.isZero) Zero
                           else error("negative number")
}
```

Zero における、前の数(predecessor)および減算(-)関数の実装は、Error 例外を送出し、与えられたエラーメッセージと共にプログラムをアボートします。

次は「次の数」クラスの実装です。

```
class Succ(x: Nat) extends Nat {
  def isZero: Boolean = false
  def predecessor: Nat = x
  def successor: Nat = new Succ(this)
  def + (that: Nat): Nat = x + that.successor
  def - (that: Nat): Nat = if (that.isZero) this
                           else x - that.predecessor
```

```
}
```

メソッド successor の実装に注意して下さい。ある数の次の数を作るために、Succ コンストラクタにオブジェクト自身を引数として渡す必要があります。オブジェクト自身は予約語 this で参照できます。

+ と - の実装はそれぞれ、コンストラクタ引数を受け手とする再帰呼び出しを含みます。再帰は、受け手が Zero オブジェクトの時（そのように数を構成したので、それが起きることは保証されています）に終わります。

演習 6.0.3 整数の実装 Integer を書きなさい。実装はクラス Nat の操作すべてをサポートし、加えて、次の２つのメソッドもサポートしなさい。

```
def isPositive: Boolean
def negate: Integer
```

最初のメソッドは数が正であれば true を返すものとします。二つ目のメソッドは数の符号を変えます。実装の中で Scala の標準の数値クラスを使ってはいけません（ヒント : Integer の実装方法は二つ可能です。一つは既存の Nat 実装を利用し、整数を自然数と符号で表すものです。あるいは３つの子クラスとして Zero を 0 に、Succ を正の数のため、Pred を負の数のために使い、既知の Nat 実装を Integer へ一般化できます)。

この章で紹介した構文

型 (Types)

```
Type = ... | ident
```

型は、クラスを表す任意の識別子です。

式 (Expressions)

```
Expr = ... | Expr '.' ident | 'new' Expr | 'this'
```

式は、オブジェクト生成、オブジェクト値を持つ式 E のメンバ m を選択する E.m、あるいは予約語の this です。

定義と宣言 (Definitions and Declarations)

```
Def         = FunDef | ValDef | ClassDef | TraitDef | ObjectDef
ClassDef    = ['abstract'] 'class' ident ['(' [Parameters] ')']
              ['extend' Expr] ['{' {TemplateDef} '}']
TraitDef    = 'trait' ident ['extends' Expr] ['{' {TemplateDef} '}']
ObjectDef   = 'object' ident ['extends' Expr] ['{' {ObjectDef} '}']
TemplateDef = [Modifier] (Def | Dcl)
ObjectDef   = [Modifier] Def
Modifier    = 'private' | 'override'
Dcl         = FunDcl | ValDcl
FunDcl      = 'def' ident {'(' [Parameters] ')'} ':' Type
ValDcl      = 'val' ident ':' Type
```

定義は次のようなクラス、トレイト、あるいはオブジェクト定義です。

```
class C(params) extends B { defs }
trait T extends B { defs }
object O extends B { defs }
```

クラス、トレイト、オブジェクト定義内の定義 def の手前に、修飾子 private あるいは override を置けます。

抽象クラスとトレイトに宣言を含めることもできます。それらは**延期された**関数や値を型とともに導入しますが、実装は与えません。延期されたメンバは、抽象クラスやトレイトのオブジェクトを生成する前に、サブクラスで実装しておく必要があります。

7 ケースクラスとパターンマッチング

ところで、数式のインタプリタを書きたいとしましょう。はじめは話を単純にするために、単に数と + 演算だけに制限します。そのような式はあるクラス階層、ルートの抽象基底クラス Expr と、２つのサブクラス Number と Sum を用いて表現できます。すると、式 1 + (3 + 7) は次のように表現されます。

```
new Sum(new Number(1), new Sum(new Number(3), new Number(7)))
```

さて、このような式評価器は、それがどの形式であるか（Sum か Number か）を知る必要があり、式の要素にアクセスする必要もあります。次は必要なメソッドすべての実装です。

```
abstract class Expr {
  def isNumber: Boolean
  def isSum: Boolean
  def numValue: Int
  def leftOp: Expr
  def rightOp: Expr
}
class Number(n: Int) extends Expr {
  def isNumber: Boolean = true
  def isSum: Boolean = false
  def numValue: Int = n
  def leftOp: Expr = error("Number.leftOp")
  def rightOp: Expr = error("Number.rightOp")
}
class Sum(e1: Expr, e2: Expr) extends Expr {
  def isNumber: Boolean = false
  def isSum: Boolean = true
  def numValue: Int = error("Sum.numValue")
  def leftOp: Expr = e1
  def rightOp: Expr = e2
}
```

これらのクラス化とアクセスメソッドによって、評価器関数は簡単に書けます。

```
def eval(e: Expr): Int = {
  if (e.isNumber) e.numValue
  else if (e.isSum) eval(e.leftOp) + eval(e.rightOp)
  else error("unrecognized expression kind")
}
```

しかし、これらすべてのメソッドをクラス Sum と Number に定義するのは、かなり退屈です。さらに、式の新しい型を追加したくなった時に問題は悪化します。たとえば乗算のために新しい式の形式 Prod を追加することを考えてみましょう。既存のクラス化とアクセスメソッドに加えて、新しいクラス Prod を実装しなくてはならないだけではなく、クラス Expr に新しい抽象メソッド isPoduct を導入する必要があり、そのメソッドをサブクラス Number, Sum, Prod に実装する必要があります。システムを拡張する時に、既存コードを修正しなくてはならないのは昔からの問題です。なぜならバージョン化と保守の問題を引き起こすからです。

オブジェクト指向プログラミングの約束することは、「そのような修正は不要です。なぜなら、継承によって既存の未修整のコードを再利用できるから」というものです。実際、問題をよりオブジェクト指向的に分解すれば問題は解決します。そのアイデアは「ハイレベルな」操作である eval を、前に我々がやったように、式クラス階層の外の関数として実装するのではなく、それぞれの式クラスのメソッドにすることです。そうすれば、eval はすべての式ノードのメンバなので、クラス化とアクセスメソッドはすべて不要となり、実装はかなり簡単になります。

```
abstract class Expr {
  def eval: Int
}
class Number(n: Int) extends Expr {
  def eval: Int = n
}
class Sum(e1: Expr, e2: Expr) extends Expr {
  def eval: Int = e1.eval + e2.eval
}
```

さらに、新しい Prod クラスの追加は既存コードに何も変化を引き起こしません。

```
class Prod(e1: Expr, e2: Expr) extends Expr {
  def eval: Int = e1.eval * e2.eval
}
```

この例から導かれる結論は、データ型の拡張可能なシステムを構築する際には、オブジェクト指向分解は選択すべきテクニックである、ということです。しかし他にも式の例を拡張したくなる方法があるかもしれません。式に対して新しい**操作**を追加したくなるかもしれません。たとえば、式の木を標準出力に整形して表示する操作を追加したくなるかもしれません。

もしすべてのクラス化とアクセスメソッドを定義してあれば、そういった操作は簡単に外部の関数として書けます。こんな風にです。

```
def print(e: Expr) {
  if (e.isNumber) Console.print(e.numValue)
  else if (e.isSum) {
    Console.print("(")
    print(e.leftOp)
    Console.print("+")
    print(e.rightOp)
    Console.print(")")
  } else error("unrecognized expression kind")
}
```

しかし、オブジェクト指向分解を選んでいたなら、新しい手続き print を各クラスに追加する必要があるでしょう。

```
abstract class Expr {
  def eval: Int
  def print
}
class Number(n: Int) extends Expr {
  def eval: Int = n
  def print { Console.print(n) }
}
class Sum(e1: Expr, e2: Expr) extends Expr {
```

```
    def eval: Int = e1.eval + e2.eval
    def print {
      Console.print("(")
      print(e1)
      Console.print("+")
      print(e2)
      Console.print(")")
    }
  }
```

したがって、システムに新しい操作を入れて拡張する時には、古典的なオブジェクト指向分解では、既存のすべてのクラスの修正が必要になります。

インタプリタの一つの拡張として、式を単純化したいとしましょう。たとえば、式の形式を a * b + a * c から a * (b + c) へ書き換える関数が欲しいとします。この操作のためには、一つ以上の式木のノードを同時に調べる必要があります。しかし、メソッドが他のノードを調べることができなければ、式の種類ごとのメソッドでは実装できません。ですからこの場合には、クラス化とアクセスメソッドを強いられます。冗長さと拡張性の問題に満ちた四角四面なやり方に逆戻りのようです。

詳しく調べてみると、クラス化とアクセス関数はデータの構成プロセスを**逆転**させるだけが目的だと分かります。それによって最初に、抽象クラスのどのサブクラスが使われたのか、その次にコンストラクタ引数が何であったのか、が決定されます。このような状況はかなり一般的なので、Scala にはそれをケースクラスによって自動化する方法があります。

7.1 ケースクラスとケースオブジェクト

ケースクラスと**ケースオブジェクト**は普通のクラスやオブジェクトのように定義しますが、定義に修飾子 case が手前に付くことだけが違います。たとえば定義

```
abstract class Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
```

は、Number と Sum をケースクラスとして導入します。クラスやオブジェクト定義の前の case 修飾子には、次のような効果があります。

1. ケースクラスは暗黙のうちにコンストラクタ関数を伴い、それはクラスと同じ名前です。この例の場合、２つの関数

   ```
   def Number(n: Int) = new Number(n)
   def Sum(e1: Expr, e2: Expr) = new Sum(e1, e2)
   ```

   が追加されます。したがって式の木をもう少し簡潔に、次のように構成できます。

   ```
   Sum(Sum(Number(1), Number(2)), Number(3))
   ```

2. ケースクラスとケースオブジェクトは暗黙のうちにメソッド toString、 equals、hashCode を伴い、それらはクラス AnyRef の同名のメソッドを オーバーライドします。これらのメソッド実装では、 それぞれのケースクラスのメンバ構造を考慮しています。toString メソッドは式の木が構成された方法を表します。したがって、

```
Sum(Sum(Number(1), Number(2)), Number(3))
```

は、この文字列そのままに変換されます。一方、クラス AnyRef のデフォルトの実装は、一番外側のコンストラクタの名前 Sum と数字からなる文字列を返すでしょう。equals メソッドはケースクラスの ２つのケースメンバを、もし同じコンストラクタで構築され、 かつ、それらの引数がそれぞれ等しいなら、等しいと扱います。 これは == と != の実装にも影響しますが、それらは Scala では equals を 用いて実装されているからです。したがって

```
Sum(Number(1), Number(2)) == Sum(Number(1), Number(2))
```

は、true を返します。もし Sum や Number がケースクラスでないなら、 同じ式は false を返します。なぜならクラス AnyRef の equals の標準実装では、 異なるコンストラクタ呼び出しで生成されたオブジェクトは常に異なる、 と扱うからです。hashCode メソッドも他の２つのメソッドと同じ原則に従います。 デフォルトの hashCode の実装ではハッシュコードを、 オブジェクトのアドレスから計算する代わりに、 ケースクラスのコンストラクタ名と、コンストラクタ引数の ハッシュコードから計算します。

3. ケースクラスは暗黙のうちに、パラメータなしのアクセサメソッドを伴い、 それはコンストラクタ引数を読み出します。例では、Number はアクセサメソッド

   ```
   def n: Int
   ```

   を持ち、コンストラクタパラメータ n を返します。 一方 Sum は２つのアクセサメソッドを持ちます。

   ```
   def e1: Expr, e2: Expr
   ```

   したがって、たとえば型 Sum の値 s に対して、 左オペランドにアクセスするために s.e1 と書けます。しかし、型 Expr の値 e に対して項 e.e1 は正しくありませんが、それは e1 は Sum で定義されて いるのであって基底クラス Expr のメンバではないからです。 では、静的型が基底クラス Expr である値に対してどうやってコンストラクタを 判別してコンストラクタ引数にアクセスすれば良いのでしょうか？ これはケースクラスの四番目にして最後の特徴によって解決されます。

4. ケースクラスを使えば、ケースクラスのコンストラクタを参照する**パターン**を 構築できます。

7.2 パターンマッチング

パターンマッチングは C や Java の switch 文をクラス階層に一般化したものです。switch 文の代わりに標準メソッド match があり、それは Scala のルートクラス Any で定義されていて、したがって全てのオブジェクトで使用できます。match メソッドは引数として複数のケースを取ります。たとえば、次はパターンマッチングを用いた eval の実装です。

```
def eval(e: Expr): Int = e match {
  case Number(n) => n
  case Sum(l, r) => eval(l) + eval(r)
}
```

この例では,2 つのケースがあります。各ケースはパターンと式を関連付けます。パターンはセレクタ値 e に対してマッチされます。この例の最初のパターンでは、Number(n) は形式 Number(v)、ただし v は任意の値、のすべての値にマッチします。この場合、**パターン変数** n は値 v に束縛されます。同様に、パターン Sum(l,r) は形式 Sum(v1,v2) のすべてのセレクタ値にマッチし、パターン変数 l と r を v1 と v2 へそれぞれ束縛します。

一般的に、パターンは次項より構成されます

- ケースクラスのコンストラクタ、たとえば Number, Sum などであり、 その引数もまたパターン
- パターン変数、たとえば n, e1, e2
- 「ワイルドカード」パターン _
- リテラル、たとえば 1, true, "abc"
- 定数識別子、たとえば MAXINT, EmptySet

パターン変数は常に小文字で始まりますが、それは大文字で始まる定数識別子と区別するためです。各変数名は一つのパターンに１回だけ登場できます。たとえば Sum(x, x) は正しくないパターンですが、それはパターン変数 x が２回登場するからです。

パターンマッチングの意味 パターンマッチング式

```
e match { case p1 => e1 ... case pn => en }
```

は、パターン p1,...pn に、書かれた順番で、セレクタ値 e に対してマッチします。

- コンストラクタパターン C(p1,...,pn) は型 C（あるいはそのサブタイプ）であり、パターン p1,...pn にマッチする、C の引数で生成されるすべての値でマッチします。
- 変数パターン x は任意の値にマッチし、変数名をその値に束縛します。
- ワイルドカードパターン '_' は任意の値にマッチしますが、 名前をその値に束縛しません。
- 定数パターン C は C と（==の意味で）等しい値にマッチします。

パターンマッチング式は、セレクタ値が最初にマッチするパターンの、ケースの右辺へと書き換えられます。パターン変数への参照は、対応するコンストラクタ引数で置き換えられます。もしどのパターンにもマッチしなければ、パターンマッチ式は MatchError 例外でアボートされます。

Example 7.2.1 プログラム評価の置き換えモデルはきわめて自然にパターンマッチングへ拡張できます。たとえば次は、簡単な式に適用された eval がどの様に書き換えるかを示します。

```
  eval(Sum(Number(1), Number(2)))
->          (適用を書き換え)
```

```
  Sum(Number(1), Number(2)) match {
    case Number(n) => n
    case Sum(e1, e2) => eval(e1) + eval(e2)
  }

->        (パターンマッチを書き換え)

  eval(Number(1)) + eval(Number(2))

->        (最初の適用を書き換え)

  Number(1) match {
    case Number(n) => n
    case Sum(e1, e2) => eval(e1) + eval(e2)
  } + eval(Number(2))

->        (パターンマッチを書き換え)

  1 + eval(Number(2))

->* 1 + 2 -> 3
```

パターンマッチングとメソッド 前の例では、パターンマッチングを、マッチするクラス階層の外で定義された関数で使用しました。もちろん、そのクラス階層自身の中でもパターンマッチングを定義できます。たとえば eval を基底クラス Expr のメソッドとして定義しても、パターンマッチングをその実装で使えます。

```
abstract class Expr {
  def eval: Int = this match {
    case Number(n) => n
    case Sum(e1, e2) => e1.eval + e2.eval
  }
}
```

演習 7.2.2 整数の木を表現する次の定義について考えなさい。この定義は IntSet の別表現とみることもできます。

```
abstract class IntTree
case object EmptyTree extends IntTree
case class Node(elem: Int, left: IntTree, right: IntTree) extends IntTree
```

次の IntTree の関数 contains と insert の実装を完成させなさい。

```
def contains(t: IntTree, v: Int): Boolean = t match { ...
  ...
}
def insert(t: IntTree, v: Int): IntTree = t match { ...
  ...
}
```

パターンマッチング無名関数 ここまでケース式は常に match 操作と一緒に現れました。しかしケース式だけでも使用できます。次のようなケース式ブロック

```
{ case P1 => E1 ... case Pn => En }
```

は、それ自身が引数をパターン P1,...,Pn にマッチさせ、E1,...,En のどれか一つの結果を生み出す関数（もしどのパターンにもマッチしなければ、関数は代わりに MatchError 例外を投げます）、と見ることもできます。別の言い方をすれば、上の式は、無名関数

```
(x => x match { case P1 => E1 ... case Pn => En })
```

の短縮形、ただし x は式の中で他には使われていない新規の変数、と見ることができます。

8 ジェネリックな型とメソッド

Scala のクラスは型パラメータを持てます。型パラメータの使い方を関数型のスタックを例として示します。さて、整数のスタックのデータ型を、メソッド push、top、pop、isEmpty とともに書きたいとしましょう。それは次のようなクラス階層になります。

```
abstract class IntStack {
  def push(x: Int): IntStack = new IntNonEmptyStack(x, this)
  def isEmpty: Boolean
  def top: Int
  def pop: IntStack
}
class IntEmptyStack extends IntStack {
  def isEmpty = true
  def top = error("EmptyStack.top")
  def pop = error("EmptyStack.pop")
}
class IntNonEmptyStack(elem: Int, rest: IntStack) extends IntStack {
  def isEmpty = false
  def top = elem
  def pop = rest
}
```

もちろん、同様に、文字列スタックの抽象化を定義するのもよいでしょう。そのために、IntStack の既存の抽象化を、StringStack と名前を変えると同時に、型 Int の出現をすべて String に置き換えます。

もっとよい方法は、コードの複製を引き起こさない方法であり、スタック定義を要素の型でパラメータ化することです。パラメータ化によって問題の特定のインスタンスをより一般的なものにできます。今まで、値についてだけパラメータ化してきましたが、型もできます。Stack の**ジェネリック**なバージョンへたどり着けるよう、型パラメータを装備しましょう。

```
abstract class Stack[A] {
  def push(x: A): Stack[A] = new NonEmptyStack[A](x, this)
  def isEmpty: Boolean
  def top: A
  def pop: Stack[A]
}
class EmptyStack[A] extends Stack[A] {
  def isEmpty = true
  def top = error("EmptyStack.top")
  def pop = error("EmptyStack.pop")
}
class NonEmptyStack[A](elem: A, rest: Stack[A]) extends Stack[A] {
  def isEmpty = false
  def top = elem
  def pop = rest
}
```

この定義で、'A' はクラス Stack とそのサブクラスの**型パラメータ**です。型パラメータは任意の名前でよく、値引数と容易に区別できるように、丸括弧ではなく角括弧で囲みます。次はジェネリックなクラスの使い方の例です。

```
val x = new EmptyStack[Int]
val y = x.push(1).push(2)
println(y.pop.top)
```

最初の行は Int のスタックを作成します。形式上の型パラメータ A を置き換える実際の型引数 [Int] に注意して下さい。

メソッドも型でパラメータ化できます。次は、あるスタックが他のスタックのプレフィックスであるか判定する、ジェネリックなメソッドの例です。

```
def isPrefix[A](p: Stack[A], s: Stack[A]): Boolean = {
  p.isEmpty ||
  p.top == s.top && isPrefix[A](p.pop, s.pop)
}
```

このメソッドのパラメータは**多相的**(polimorphic)と言われます。ジェネリックメソッドも同じく**多相的**と言われます。この用語はギリシャ語由来で「多くの形を持つ」という意味です。isPrefix のような多相的メソッドを適用するには、型パラメータを値パラメータと一緒に渡します。たとえば

```
val s1 = new EmptyStack[String].push("abc")
val s2 = new EmptyStack[String].push("abx").push(s1.top)
println(isPrefix[String](s1, s2))
```

局所的な型推論 [Int] や [String] のような型パラメータをいつも渡すのは、ジェネリック関数を多用するアプリケーションにおいては退屈です。多くの場合、型パラメータの情報は冗長です。なぜなら、正しいパラメータ型は、関数の値パラメータや期待される結果型(戻り値型)から決定できるからです。例として、式 isPrefix[String](s1, s2) を考えると、値パラメータは両方とも Stack[String] であることが分かっており、したがって型パラメータは String に違いないと推論できます。このような状況下で、多相的関数とコンストラクタの型パラメータの省略を許す、かなり強力な型推論器が Scala にはあります。先の例では isPrefix(s1, s2) と書け、見えていない型引数 [String] は型推論器によって挿入されます。

8.1 型パラメータの境界

クラスをジェネリックにする方法を知った今では、前に書いたクラスの中には一般化した方が自然なものがあります。たとえばクラス IntSet は、任意の要素型を持てるように一般化できます。やってみましょう。ジェネリックな集合の抽象クラスは簡単に書けます。

```
abstract class Set[A] {
  def incl(x: A): Set[A]
  def contains(x: A): Boolean
}
```

しかしもし二分木として実装し続けたいなら、問題に直面します。メソッド contains と incl はどちらも、要素をメソッド < と > を使って比較します。IntSet については、これは OK です。なぜなら Int はそれら 2 つのメソッドを持っているからです。しかし任意の型パラメータ a に対しては、それは保証できません。したがって先の実装、たとえば contains ではコンパイルエラーが生じるでしょう。

```
  def contains(x: Int): Boolean =
    if (x < elem) left contains x
          ^ < not a member of type A.
```

問題を解決する一つの方法は、型 A と置き換え可能な正当な型を、正しい型のメソッド < と > を持つ型だけに制限することです。Scala 標準ライブラリには、型 A の値と（< と > によって）比較可能な値を表現する、トレイト Ordered[A] があります。このトレイトは次のように定義されています。

```
  /** データを完全に順序づけるためのクラス */
  trait Ordered[A] {

    /** this をオペランド that と比較した結果。
     * returns 'x' where
     *   x < 0    iff this < that
     *   x == 0 iff this == that
     *   x > 0    iff this > that
     */
    def compare(that: A): Int

    def <  (that: A): Boolean = (this compare that) < 0
    def >  (that: A): Boolean = (this compare that) > 0
    def <= (that: A): Boolean = (this compare that) <= 0
    def >= (that: A): Boolean = (this compare that) >= 0
    def compareTo(that: A): Int = compare(that)
  }
```

型が Ordered のサブタイプであることを要求することで、互換性を強制できます。これは Set の型パラメータに上界境界(upper bound)を与えることでなされます。

```
  trait Set[A <: Ordered[A]] {
    def incl(x: A): Set[A]
    def contains(x: A): Boolean
  }
```

パラメータ宣言 A <: Ordered[A] は型パラメータとしての A を、A は Ordered[A] のサブタイプでなくてはならない、として導入します。すなわち、その値は同じ型の値と比較可能でなくてはなりません。

この制限によって、ジェネリックな集合の抽象化を、前の IntSet の場合と同じように実装できます。

```
  class EmptySet[A <: Ordered[A]] extends Set[A] {
    def contains(x: A): Boolean = false
    def incl(x: A): Set[A] = new NonEmptySet(x, new EmptySet[A], new EmptySet[A])
  }

  class NonEmptySet[A <: Ordered[A]]
          (elem: A, left: Set[A], right: Set[A]) extends Set[A] {
    def contains(x: A): Boolean =
      if (x < elem) left contains x
      else if (x > elem) right contains x
      else true
    def incl(x: A): Set[A] =
      if (x < elem) new NonEmptySet(elem, left incl x, right)
```

```
    else if (x > elem) new NonEmptySet(elem, left, right incl x)
    else this
}
```

オブジェクト生成 new NonEmptySet(...) において、型引数を書いていないことに注意して下さい。多相的メソッドと同様に、コンストラクタ呼び出しで型引数が書かれていない時、それは値引数と期待される結果型(戻り値型)から推論されます。

以下はジェネリックな集合の抽象化を使う例です。はじめに、Ordered のサブクラスを次のように作りましょう。

```
case class Num(value: Double) extends Ordered[Num] {
  def compare(that: Num): Int =
    if (this.value < that.value) -1
    else if (this.value > that.value) 1
    else 0
}
```

すると

```
val s = new EmptySet[Num].incl(Num(1.0)).incl(Num(2.0))
s.contains(Num(1.5))
```

これは OK です。なぜなら型 Num はトレイト Ordered[Num] を実装しているからです。しかし次の例はエラーになります。

```
val s = new EmptySet[java.io.File]
                                ^ java.io.File does not conform to type
                                  parameter bound Ordered[java.io.File].
```

型パラメータ境界の問題は、前もって考えておくべきことです。もし Num を Ordered のサブクラスとして宣言していなければ、Num を集合の要素として使用できません。同様に、Java から継承した Int、Double、String といった型は Ordered のサブクラスではないので、それらの型の値は集合の要素として使用できません。

それらの型を要素として許すもっと柔軟なデザインがあります。**可視境界**(view bound)を今までに見てきた単純な型境界の代わりに使うことです。先の例でこれによって起こる変更は、単に型パラメータが次のようになることです。

```
trait Set[A <% Ordered[A]] ...
class EmptySet[A <% Ordered[A]] ...
class NonEmptySet[A <% Ordered[A]] ...
```

可視境界 <% は通常の上限境界 <: より弱いです。可視境界をもつ型パラメータ節 [A <% T] は単に、境界づけられる型 A が境界となる型 T へ、暗黙の型変換を使って**変換可能**なことだけを指定します。

Scala ライブラリは、プリミティブ型や String を含む数多くの型について、暗黙の型変換を事前に定義しています。したがって、再デザインされた集合の抽象化は、これらの型についてもインスタンス化できます。暗黙の変換と可視境界に関する更なる説明は第 15 章にあります。

8.2 変位指定アノテーション

型パラメータとサブタイプ化の組み合わせは興味深い問題を引き起こします。たとえば、Stack[String] は Stack[AnyRef] のサブタイプであるべきでしょうか？ 直感的には OK のように思われます。String のスタックは AnyRef のスタックの特別な場合だからです。より一般的に言うと、もし T が型 S のサブタイプなら、Stack[T] は Stack[S] のサブタイプであるべきです。この性質は**共変**(co-variant)サブタイプ化と呼ばれます。

Scala では、ジェネリック型はデフォルトでは非変(non-variant)サブタイプ化です。それは、前に定義した Stack については、異なる要素型のスタックは決してサブタイプ関係にないということです。しかし、クラス Stack の定義の一行目を次のように変更すれば、共変サブタイプ化を強制できます。

```
class Stack[+A] {
```

形式上の型パラメータの前に + を置くことで、このパラメータについてサブタイプ化が共変であることを示します。+ に加えて、反変サブタイプ化を示す接頭辞 - もあります。もし Stack を class Stack [-T] ... と定義すれば、T が型 S のサブタイプなら、Stack[S] は Stack[T] のサブタイプになります（スタックの場合には驚くべきことでしょう！）。

純粋関数的な世界では、すべての型は共変となることができます。しかし、もしミュータブル(変更可能)なデータを導入すると状況は変わります。Java や .NET の配列について考えてみましょう。そのような配列は、Scala ではジェネリックなクラス Array で表現されます。次はこのクラスの定義の一部です。

```
class Array[A] {
  def apply(index: Int): A
  def update(index: Int, elem: A)
}
```

このクラスは、Scala のユーザープログラムから Scala の配列がどのように見えるかを定義します。Scala コンパイラはこの抽象化を、基礎となっているホストシステムの配列へ、それが可能なほとんどの場合、マップします。

実際 Java では配列は共変です。つまり、参照型 T と S について、もし T が S のサブタイプなら、Array[T] は Array[S] のサブタイプです。これは自然なことに思えるかもしれませんが、特別な実行時チェックを必要とする安全上の問題をもたらします。次がその例です。

```
val x = new Array[String](1)
val y: Array[Any] = x
y(0) = new Rational(1, 2) // これは次の糖衣構文
                          // y.update(0, new Rational(1, 2))
```

最初の行で、新しい文字列配列が生成されます。二行目で、この配列は Array[Any] 型の変数 y に束縛されます。配列が共変だということを仮定すると、これは OK です。なぜなら Array[String] は Array[Any] のサブタイプだからです。最後に、最終行で有理数がこの配列に代入されます。これも OK です。なぜなら型 Rational は、配列 y の要素型 Any のサブタイプだからです。そして結局最後には、文字列配列に有理数が格納され、明らかに型健全性を破ります。

Java はこの問題を、三行目に実行時チェックを導入して、格納された要素と配列が作られた要素型との互換性をテストし、解決します。例で見たようにこの要素型は、必ずしも更新される配列の静的な要素型ではありません。テストが失敗すると ArrayStoreException が発生します。

Scala はその代わりに、この問題を静的に解決します。Scala では、配列は非変サブタイプ化なのでコンパイル時に二行目を許可しないことで、解決します。これは、Scala はどうやって変位指定アノテーションの正しさを検証するのか、という疑問を生じさせます。もし単に配列を共変に宣言したら、どうやって潜在的な問題を見つけるのでしょうか？

Scala は変位指定アノテーションの健全性を検証するために、用心深い推定をします。クラスの共変的型パラメータは、クラス内の共変な場所にだけ現れることができます。クラス内の値の型で共変な場所は、クラス内のメソッドの結果型と他の共変的型への型引数です。形式上のメソッドパラメータの型は共変ではありません。したがって次のクラス定義は却下されます。

```
class Array[+A] {
  def apply(index: Int): A
  def update(index: Int, elem: A)
                              ^ 共変的型パラメータ A が
                                反変的位置に現れる。
}
```

これまでのところは問題ありません。直感的には共変的なクラスの update 手続きを却下する点でコンパイラは正しいです。なぜなら update は状態を変更することができ、したがって共変サブタイプ化の健全性の土台を揺るがすからです。

しかしながら、状態を変更しないメソッドで、型パラメータが反変的な位置に現れるものがあります。例は型 Stack の push です。Scala コンパイラは共変的スタックに対するこのメソッド定義を再び却下します。

```
class Stack[+A] {
  def push(x: A): Stack[A] =
              ^ 共変的型パラメータ A が
                反変的位置に現れる。
```

これは残念です。なぜなら配列と違ってスタックは純粋関数的なデータ構造であり、共変サブタイプ化できるべきだからです。しかしこの問題を、下限境界付きの型パラメータをもつ多相的メソッドで解決する方法があります。

8.3 下限境界

型パラメータの上限境界について見てきました。T <: U のような型パラメータ宣言では、型パラメータ T は型 U のサブタイプの範囲に制限されます。これと対照的なのが Scala における下限境界です。型パラメータ宣言 T >: S において、型パラメータ T は型 S の**スーパータイプ**の範囲に制限されます（上限境界と下限境界を組み合わせることもできます。T >: S <: U のように）。

下限境界を用いると、Stack の push メソッドを次のように一般化できます。

```
class Stack[+A] {
  def push[B >: A](x: B): Stack[B] = new NonEmptyStack(x, this)
```

技術的には、これによって変位指定問題は解決します。なぜなら、型パラメータ A はいまや push メソッドの型パラメータではないからです。その代わり、メソッドの別の型パラメータの下限境界として現れ、その型パラメータは共変的な位置に分類されます。したがって Scala コンパイラは新しい push 定義を受け入れます。

実際、変位指定の技術的な問題を解決しただけではなく、push 定義を一般化しました。以前は、スタックの宣言された要素型に適合する型のみプッシュできました。今では、この型のサブタイプの要素もプッシュできますが、戻り値のスタックの型は適宜変化します。たとえば String のスタックに AnyRef をプッシュできますが、戻るスタックは String のスタックではなく AnyRef のスタックなのです！

要約すると、自分のデータ構造に変位指定アノテーションを付けることを躊躇すべきではありません。なぜならそれによって、リッチで自然なサブタイプ関係が得られるからです。コンパイラは健全性に関する潜在的な問題を見つけます。クラス Stack の push メソッドのようにコンパイラの推定が用心深すぎたとしても、問題となっているメソッドについて、有益な一般化をしばしば示唆します。

8.4 最下層の型

Scala では、オブジェクトを型でパラメータ化することはできません。そういう訳で、たとえ任意の型の空スタックを表すただ一つの値で充分だったとしても、最初にジェネリッククラス EmptyStack[A] を定義しました。しかし、共変的スタックについては、次のイディオムを使えます。

```
object EmptyStack extends Stack[Nothing] { ... }
```

ボトム型 Nothing は値を持ちません。したがって型 Stack[Nothing] は、EmptyStack が要素を含まないことを表現します。さらに、Nothing は他のすべての型のサブタイプです。ですから、共変的スタックではどんな型 T に対しても、Stack[Nothing] は Stack[T] のサブタイプです。これによってユーザーコード中で、単一の空スタックオブジェクトを使用できます。たとえば、

```
val s = EmptyStack.push("abc").push(new AnyRef())
```

この式について、型の割り当てを詳細に解析しましょう。EmpthStack オブジェクトは型 Stack[Nothing] であり、次のメソッドを持っています。

```
push[B >: Nothing](elem: B): Stack[B] .
```

局所的な型推論は、型パラメータ B は EmptyStack の適用において String へとインスタンス化されているに違いない、と決定します。したがって、その適用の結果型(戻り値型)は Stack[String] であり、次のメソッドを持っています。

```
push[B >: String](elem: B): Stack[B] .
```

上記の値定義の最後の箇所は、このメソッドの new AnyRef() への適用です。局所的型推論は今回、型パラメータ b は AnyRef へとインスタンス化されているに違いなく、結果型は Stack[AnyRef] である、と決定します。したがって値 s に割り当てられる型は Stack[AnyRef] です。

Nothing はすべての型のサブタイプでした。ほかに、型 Null というものもあり、これは scala.AnyRef と、AnyRefを継承するすべてのクラスのサブタイプです。 Scala の null リテラルはその型の唯一の値です。これによって null はすべての参照型と互換ですが、Int のような値型とは互換ではありません。

完全に改良したスタック定義で、この章を締めくくります。いまやスタックは共変的サブタイプで、push メソッドは一般化され、空スタックは単一のオブジェクトで表現されます。

```
abstract class Stack[+A] {
  def push[B >: A](x: B): Stack[B] = new NonEmptyStack(x, this)
  def isEmpty: Boolean
  def top: A
  def pop: Stack[A]
}
object EmptyStack extends Stack[Nothing] {
  def isEmpty = true
  def top = error("EmptyStack.top")
  def pop = error("EmptyStack.pop")
}
class NonEmptyStack[+A](elem: A, rest: Stack[A]) extends Stack[A] {
  def isEmpty = false
  def top = elem
  def pop = rest
}
```

Scala ライブラリ内の多くのクラスはジェネリックです。ジェネリックなクラスの系統でよく使われる、タプルと関数を紹介します。他のよく使われるクラスであるリストは、次の章に回します。

8.5 タプル

関数から１つ以上の結果を返す必要がたまに生じます。たとえば、与えられた２つの整数引数の整数商と余りを返す関数 divmod を考えましょう。もちろん、divmod の２つの結果を保持するクラスを次のようにも定義できます。

```
case class TwoInts(first: Int, second: Int)
def divmod(x: Int, y: Int): TwoInts = new TwoInts(x / y, x % y)
```

しかし、結果型のペアー つ一つに対し、新しいクラスを定義しなくてはならないのでは、あまりに面倒です。Scala ではその代わりに、次のように定義されたジェネリッククラス Tuple2 を使えます。

```
package scala
case class Tuple2[A, B](_1: A, _2: B)
```

Tuple2 を使って divmod メソッドは次のように書けます。

```
def divmod(x: Int, y: Int) = new Tuple2[Int, Int](x / y, x % y)
```

コンストラクタへの型パラメータは、値引数から推論可能な場合、通常省略できます。異なる要素数ごとにタプルクラスが存在します（現在の Scala 実装では、要素数をある適正値に制限しています）。

タプルの要素にどうやってアクセスするのでしょうか？ タプルはケースクラスなので、２つの可能性があります。次の例のように、タプルのフィールドにコンストラクタパラメータ名 _i を用いてアクセスできます。

```
val xy = divmod(x, y)
println("quotient: " + xy._1 + ", rest: " + xy._2)
```

あるいは次の例のように、タプルに対するパターンマッチを用います。

```
divmod(x, y) match {
  case Tuple2(n, d) =>
    println("quotient: " + n + ", rest: " + d)
}
```

型パラメータはパターンでは決して使われないことに注意して下さい。「case Tuple2[Int, Int](n, d)」と書くのは文法違反です。(訳注 : たとえば、「case x:Tuple2[Int,Int] =>・・・」と書くのは OK です)

タプルは非常に便利なので、Scala では特別な構文が用意されています。n 要素 x1,...,xn のタプルは、(x1,...,xn) と書けます。これは Tuplen(x1,...,xn) と等価です。(...) 構文は、型やパターンについても同じように働きます。タプル構文を使って、divmod の例は次のように書けます。

```
def divmod(x: Int, y: Int): (Int, Int) = (x / y, x % y)

divmod(x, y) match {
  case (n, d) => println("quotient: " + n + ", rest: " + d)
}
```

8.6 関数

関数が第一級クラス値であるという点で Scala は関数型言語です。Scala はすべての値がオブジェクトであるという点でオブジェクト指向言語です。したがって Scala では関数はオブジェクトです。たとえば、型 String から型 Int への関数は、トレイト Function1[String, Int] のインスタンスとして表現されます。Function1 トレイトは次のように定義されます。

```
package scala
  trait Function1[-A, +B] {
  def apply(x: A): B
}
```

Function1 の他にも、異なるすべてのアリティ(項数)の関数に対して,定義があります (現在は、適正値までしか実装されていません) 。つまり、関数のパラメータの数ごとに定義があるということです。Scala の関数型構文 (T1,...,Tn) => S は、単に、パラメータ化された型 Functionn[T1,...,Tn,S] の省略形です。

Scala は f がメソッドか関数かに関わらず、同じ構文 f(x) を関数適用に用います。これは、「 f が (メソッドではなく)オブジェクトである時、関数適用 f(x) は f.apply(x) の省略形であるとみなす」という規則に従うことで可能となります。したがって関数型の apply メソッドは必要なときに自動的に挿入されます。

これは 8.2 節で、配列の添字化を apply メソッドで定義したのと同じです。あらゆる配列 a について、添字操作 a(i) は a.apply(i) の省略とみなされます。

関数は反変の型パラメータ宣言の有用な例です。たとえば次のコードを考えてみましょう。

```
val f: (AnyRef => Int) = x => x.hashCode()
val g: (String => Int) = f
g("abc")
```

型 String => Int の値 g を、型 AnyRef => Int の f に束縛するのは健全です。実際、型 String => Int の関数を使ってできることは、整数を得るために文字列を渡すことなのですから。明らかに関数 f は同じように動作します。文字列（あるいは任意のオブジェクト）を渡せば整数を得ます。これによって関数のサブタイプ化は、引数については反変ですが、結果型に対しては共変です。簡単に言うと、S' が S のサブタイプで、T が T' のサブタイプなら、S=>T は S'=>T' のサブタイプです。

Example 8.6.1 次の Scala コードについて考察しなさい。

```
val plus1: (Int => Int) = (x: Int) => x + 1
plus1(2)
```

これは次のオブジェクトコードに展開されます。

```
val plus1: Function1[Int, Int] = new Function1[Int, Int] {
  def apply(x: Int): Int = x + 1
}
plus1.apply(2)
```

ここでオブジェクト生成 new Function1[Int,Int]{...} は、**無名クラス**のインスタンスを表しています。これは新しい Function1 オブジェクトの生成と apply メソッド（Function1 では抽象メソッド）の実装を結びつけます。同じことを冗長にはなりますが、局所クラスを使っても書けます。

```
val plus1: Function1[Int, Int] = {
  class Local extends Function1[Int, Int] {
    def apply(x: Int): Int = x + 1
  }
  new Local: Function1[Int, Int]
}
plus1.apply(2)
```

9 リスト (Lists)

リストは多くの Scala プログラムで重要なデータ構造です。要素 x1,...,xn を含むリストは List(x1,...,xn) と書きます。例は

```
val fruit = List("apples", "oranges", "pears")
val nums = List(1, 2, 3, 4)
val diag3 = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty = List()
```

リストは C や Java 言語などの配列と似ていますが、3つの重要な違いがあります。第一に、リストはイミュータブル(変更不能)です。つまりリストの要素は代入によって変更できません。第二に、リストは再帰的な構造を持っていますが、配列はフラットです。第三に、リストは配列がふつう持っているよりずっと豊富な操作をサポートしています。

9.1 リストの使用

リスト型 配列と同じくリストは**等質**です。つまり、リストの要素はすべて同じ型です。要素型 T のリストの型は List[T] と書きます(Java の T[] は配列です)。

```
val fruit: List[String] = List("apples", "oranges", "pears")
val nums : List[Int] = List(1, 2, 3, 4)
val diag3: List[List[Int]] = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty: List[Int] = List()
```

リストのコンストラクタ すべてのリストは2つの基本的なコンストラクタである Nil と :: ("cons" と発音する) から作られます。Nil は空リストを表します。中置演算子 :: は、リストの拡張を表します。つまり x :: xs は、最初の要素が x で、その後にリスト xs (の要素) が続くリストを表します。したがって、先ほどのリストの値は、次のようにも定義できます (実際、前述の定義は次の単なる糖衣構文です)。

```
val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
val nums = 1 :: (2 :: (3 :: (4 :: Nil)))
val diag3 = (1 :: (0 :: (0 :: Nil))) ::
            (0 :: (1 :: (0 :: Nil))) ::
            (0 :: (0 :: (1 :: Nil))) :: Nil
val empty = Nil
```

'::' 操作は右結合です。A :: B :: C は A :: (B :: C) と解釈されます。したがって先ほどの定義で、括弧は省略できます。たとえば次のように短く書けます。

```
val nums = 1 :: 2 :: 3 :: 4 :: Nil
```

リストの基本操作 リストのすべての操作は、次の三つを使って表現できます。

これらの操作は、リストオブジェクトのメソッドとして定義されています。操作対象のリストから、それらの操作を選択して呼び出してみましょう。たとえば

```
empty.isEmpty = true
fruit.isEmpty = false
fruit.head = "apples"
fruit.tail.head = "oranges"
diag3.head = List(1, 0, 0)
```

head と tail メソッドは非空リストに対してだけ定義されています。空リストから選択された場合は、例外を投げます。

リストを扱う例として、数のリストを昇順にソートすることを考えましょう。ソートする一つの簡単な方法は**挿入ソート**であり、次のようにします。最初の要素が x で残りが xs であるような非空リストをソートするには、残りの xs をソートして、要素 x をその結果の正しい場所に挿入します。空リストのソートは空リストになります。Scala のコードで表現すると

```
def isort(xs: List[Int]): List[Int] =
  if (xs.isEmpty) Nil
  else insert(xs.head, isort(xs.tail))
```

演習 9.1.1 書かれていない関数 insert を実装しなさい。

リストパターン 実際のところ、:: は Scala の標準ライブラリでケースクラスとして定義されています。ですから、Nil と :: コンストラクタからなるパターンを用いた、パターンマッチングでリストを分解できます。たとえば isort は次のようにも書けます。

```
def isort(xs: List[Int]): List[Int] = xs match {
  case List() => List()
  case x :: xs1 => insert(x, isort(xs1))
}
```

ただし、

```
def insert(x: Int, xs: List[Int]): List[Int] = xs match {
  case List() => List(x)
  case y :: ys => if (x <= y) x :: xs else y :: insert(x, ys)
}
```

9.2 リストクラスの定義 I :一階メソッド

リストは Scala では組み込み型ではありません。抽象クラス List として定義され、2 つのサブクラス :: と Nil があります。以降では、クラス List のツアーに出かけます。

```
package scala
abstract class List[+A] {
```

List は抽象クラスであり、空の List コンストラクタを呼び出して（たとえば new List ）、要素を定義することはできません。クラスには型パラメータ a があります。このパラメータについて共変、つまり S <: T であるような型 S と T について、List[S] <: List[T] です。クラスはパッケージ scala にあります。このパッケージは Scala の最も重要で標準的なクラスを含んでいます。List は多くのメソッドを定義しており、以下でそれを説明します。

リストの分解 最初に、３つの基本的なメソッド isEmpty, head, tail があります。パターンマッチングによるそれらの実装は簡単です。

```
def isEmpty: Boolean = this match {
  case Nil => true
  case x :: xs => false
}
def head: A = this match {
  case Nil => error("Nil.head")
  case x :: xs => x
}
def tail: List[A] = this match {
  case Nil => error("Nil.tail")
  case x :: xs => xs
}
```

次の関数はリストの長さを計算します。

```
def length: Int = this match {
  case Nil => 0
  case x :: xs => 1 + xs.length
}
```

演習 9.2.1 length の末尾再帰版を設計しなさい。

次の２つの関数は head と tail を補完するものです。

```
def last: A
def init: List[A]
```

xs.last はリスト xs の最後の要素を返し、一方で xs.init は最後以外の xs の全要素を返します。両関数ともリスト全体をたどる必要があるので、類似の head や tail より効率的ではありません。次は last の実装です。

```
def last: A = this match {
  case Nil => error("Nil.last")
  case x :: Nil => x
  case x :: xs => xs.last
}
```

init の実装も似ています。

次の３つの関数は、リストの先頭部、末尾部、あるいはその両方を返します。

```
def take(n: Int): List[A] =
  if (n == 0 || isEmpty) Nil else head :: tail.take(n-1)
```

```
  def drop(n: Int): List[A] =
    if (n == 0 || isEmpty) this else tail.drop(n-1)

  def split(n: Int): (List[A], List[A]) = (take(n), drop(n))
```

(xs take n) は、リスト xs の最初の n 要素、あるいは長さが n 未満の場合はリスト全体を返します。(xs drop n) は、最初の n 要素を除いた xs の全要素を返します。最後に (xs split n) は、xs take n と xs drop n からなるリストのペアを返します。

次の関数はリストの、与えられたインデックス要素を返します。つまり配列の添字と似ています。インデックスは 0 から始まります。

```
  def apply(n: Int): A = drop(n).head
```

apply メソッドは Scala では特別な意味を持ちます。apply メソッドを持つオブジェクトはあたかも関数のように引数へ適用されます。たとえば、リスト xs の第 3 要素を選び出すには xs.apply(3) とも xs(3) とも書けます --- 後の式は前の式に展開されます。

take と drop を使って、元のリストの連続した要素からなるサブリストを取り出せます。リスト xs のサブリスト xsm,...,xsn-1 を取り出すには、

```
  xs.drop(m).take(n - m)
```

リストの ZIP 次の関数は２つのリストを組み合わせて、ペアからなる一つのリストを作ります。２つのリスト

```
  xs = List(x1, ..., xn ) , and
  ys = List(y1, ..., yn ) ,
```

に対して、xs zip ys はリスト List((x1,y1),...(xn,yn)) を作ります。もし２つのリストの長さが違う場合は、長い方が切り捨てられます。次は zip の定義です。多相的メソッドであることに注意してください。

```
  def zip[B](that: List[B]): List[(A,B)] =
    if (this.isEmpty || that.isEmpty) Nil
    else (this.head, that.head) :: (this.tail zip that.tail)
```

リストの CONS すべての中置演算子と同じように、:: もオブジェクトのメソッドとして実装されています。この場合、そのオブジェクトは拡張されるリストです。Scala では、文字 ':' で終わる演算子は特別に扱われるため、それが可能です。そのような演算子は、右オペランドのメソッドとして扱われます。たとえば

```
  x :: y = y.::(x)   一方   x + y = x.+(y)
```

しかし、どちらの場合も二項演算のオペランドは左から右に評価されることに注意して下さい。もし D と E が副作用の可能性のある式なら、D :: E は オペランド評価において、左から右への順序を維持するために、{val x = D; E.::(x)} と変換されます。

':' で終わる演算子と他の演算子とのもう一つの違いは、結合性に関するものです。':' で終わる演算子は右結合で、他の演算子は左結合です。たとえば、

```
x :: y :: z = x :: (y :: z)   一方   x + y + z = (x + y) + z
```

クラス List のメソッドとしての :: の定義は、次のようです。

```
def ::[B >: A](x: B): List[B] = new scala.::(x, this)
```

:: は、x の型 B がリストの要素型 A のスーパータイプであるような、型 B のすべての要素 x と、List[A] 型のリストに対して定義されていることに注意して下さい。この場合、結果は要素型 B のリストになります。このことは :: のシグネチャにおいて、下限境界 A を持つ型パラメータ B、として表現されています。

リストの連結 :: と似た操作はリストの連結で、':::' と書きます。(xs ::: ys) の結果は xs の全要素に ys の全要素が続いたリストです。コロンで終わるため、::: は右結合であり右オペランドのメソッドとみなされます。したがって、

```
xs ::: ys ::: zs = xs ::: (ys ::: zs)
                 = zs.:::(ys).:::(xs)
```

以下は ::: メソッドの実装です。

```
def :::[B >: A](prefix: List[B]): List[B] = prefix match {
  case Nil => this
  case p :: ps => this.:::(ps).::(p)
}
```

リストの逆転 ほかにも有用な操作として、リストの逆転があります。List にはそのためのメソッド reverse があります。実装を与えてみましょう。

```
def reverse[A](xs: List[A]): List[A] = xs match {
  case Nil => Nil
  case x :: xs => reverse(xs) ::: List(x)
}
```

この実装には単純であるという利点がありますが、あまり効率的とは言えません。実際のところ、リストの各要素に対して１回の連結が実行されます。List の連結は、最初のオペランドの長さに比例する時間がかかります。したがって reverse(xs) の計算量は

```
n + (n - 1) + ... + 1 = n(n + 1)/2
```

ただし n は xs の長さです。reverse をもっと効率的に実装できますか？線形な計算量しか持たない異なる実装をあとで見ます。

9.3 例 : マージソート

この章で前に示した挿入ソートは、書くのは簡単ですがあまり効率的ではありません。その平均的な計算量は入力リストの長さの２乗に比例します。さて、挿入ソートより効率的にリスト要素をソートするプログラムをデザインしましょう。そのためのよいアルゴリズムは**マージソート**です。次のように動きます。

はじめに、もしリストが 0 個あるいは 1 個の要素を持つなら、それはすでにソートされているので、そのままリストを返します。長いリストは 2 つのサブリストに分割され、それぞれが元のリストの半分の要素を含むようにします。各サブリストはソート関数への再帰呼び出しでソートされ、得られる 2 つのソート済みリストは、マージ操作にて結合されます。

マージソートを汎用的な実装とするために、ソートされるリストの要素型だけではなく、要素の比較に使う関数も指定できるようにすべきです。それら 2 つの要素をパラメータとして渡すことで、可能な限り汎用性のある関数が得られます。以上から次の実装を得ます。

```
def msort[A](less: (A, A) => Boolean)(xs: List[A]): List[A] = {
  def merge(xs1: List[A], xs2: List[A]): List[A] =
    if (xs1.isEmpty) xs2
    else if (xs2.isEmpty) xs1
    else if (less(xs1.head, xs2.head)) xs1.head :: merge(xs1.tail, xs2)
    else xs2.head :: merge(xs1, xs2.tail)
  val n = xs.length/2
  if (n == 0) xs
  else merge(msort(less)(xs take n), msort(less)(xs drop n))
}
```

msort の計算量は O(N log (N))、ただし N は入力リストの長さです。理由を見てみます。リストを二つに分割し、ソートされた 2 つのリストをマージするには、それぞれに引数のリストの長さに比例した時間を要します。msort の再帰呼び出しの度に入力の要素数は半分になり、リスト長が 1 に達するまでに O(log (N)) 回の再帰呼び出しが行われます。しかし長い方のリストについては、各呼び出しで更に 2 回の呼び出しが生じます。すべてを足し合わせると、O(log(N)) 回の呼び出しレベルのそれぞれに対して、元のリストの全要素が 1 回の分割操作と 1 回のマージ操作に関わります。すなわち各呼び出しレベルは、全部で O(N) に比例するコストがかかります。O(log(N)) の呼び出しレベルがあるため、全体で O(N log(N)) のコストとなります。このコストはリストの要素の初期分布には依存せず、最悪ケースのコストは平均ケースと同じです。このためマージソートは、リストのソートとして魅力的なアルゴリズムです。

次は msort を使う例です。

```
msort((x: Int, y: Int) => x < y)(List(5, 7, 1, 3))
```

msort の定義はカリー化されていて、特定の比較関数を使って簡単に特化できます。たとえば

```
val intSort = msort((x: Int, y: Int) => x < y)
val reverseSort = msort((x: Int, y: Int) => x > y)
```

9.4 リストクラスの定義 II :高階メソッド

ここまでに見た例が示すことは、リストに対する関数はしばしば似た構造をしているということです。リストに対する計算パターンをいくつか見ることができます。たとえば、

- ある方法で全ての要素を変換する
- 条件を満たす全ての要素を取り出す。
- ある演算子を用いて全ての要素を結合する

関数型プログラミング言語では高階関数を使うことで、パターンを実装する汎用的な関数を書けます。よく使われる高階関数について議論しますが、それらはクラス List のメソッドとして実装されています。

リストのマッピング リストの各要素を変換して結果のリストを得ることは、よくある操作です。たとえば、リストの各要素を与えられた数だけ倍するのは、

```
def scaleList(xs: List[Double], factor: Double): List[Double] = xs match {
  case Nil => xs
  case x :: xs1 => x * factor :: scaleList(xs1, factor)
}
```

このパターンは、クラス List の map メソッドとして一般化できます。

```
abstract class List[A] { ...
  def map[B](f: A => B): List[B] = this match {
  case Nil => this
  case x :: xs => f(x) :: xs.map(f)
}
```

map を用いると、scaleList は次のように簡潔になります。

```
def scaleList(xs: List[Double], factor: Double) =
  xs map (x => x * factor)
```

ほかの例として、行のリストとして表現されている行列の、指定された列を返す問題、ただし各行もまたリストである、を考えましょう。これは次の関数 column で与えられます。

```
def column[A](xs: List[List[A]], index: Int): List[A] =
  xs map (row => row(index))
```

map と近い関係にあるのは foreach メソッドで、リストの全要素に関数を適用しますが、結果のリストを組み立てません。この関数はですから、副作用のためだけのものです。foreach は次のように定義されています。

```
def foreach(f: A => Unit) {
  this match {
    case Nil => ()
    case x :: xs => f(x); xs.foreach(f)
  }
}
```

たとえば、この関数はリストの要素すべてを表示するのに使えます。

```
xs foreach (x => println(x))
```

演習 9.4.1 リストの要素すべてを 2 乗して、結果のリストを返す関数について考察しなさい。次の 2 つの等価な定義を完成させなさい。

```
def squareList(xs: List[Int]): List[Int] = xs match {
  case List() => ??
  case y :: ys => ??
```

```
}
def squareList(xs: List[Int]): List[Int] =
  xs map ??
```

リストのフィルタリング 他のよくある操作は、リストの、与えられた条件を満たすすべての要素を取り出す操作です。たとえば整数リスト中の正の要素からなるリストを返すには、

```
def posElems(xs: List[Int]): List[Int] = xs match {
  case Nil => xs
  case x :: xs1 => if (x > 0) x :: posElems(xs1) else posElems(xs1)
}
```

このパターンは、クラス List の filter メソッドへ一般化できます。

```
def filter(p: A => Boolean): List[A] = this match {
  case Nil => this
  case x :: xs => if (p(x)) x :: xs.filter(p) else xs.filter(p)
}
```

filter を使うと、posElems は次のように簡潔に書けます。

```
def posElems(xs: List[Int]): List[Int] =
  xs filter (x => x > 0)
```

フィルタに関係した操作として、リストのすべての要素が、ある条件を満たすかをテストするものがあります。対を成すものとして、リストに、ある条件を満たす要素があるか否か、という問題にも興味があるかもしれません。これらの操作はクラス List の forall と exists という高階関数として具体化されます。

```
def forall(p: A => Boolean): Boolean =
  isEmpty || (p(head) && (tail forall p))
def exists(p: A => Boolean): Boolean =
  !isEmpty && (p(head) || (tail exists p))
```

forall の使い方を示すために、ある数が素数であるか否かという問題を考えましょう。数 n が素数であるとは、１と自分自身でのみ割り切れるということを思い出して下さい。最も直接的にこの定義を翻訳すると、２以上 n 未満のすべての数で割った余りが非ゼロであることをテストすることです。数のリストは、次のようにオブジェクト List で定義されている List.range 関数で生成できます。

```
package scala
object List { ...
  def range(from: Int, end: Int): List[Int] =
    if (from >= end) Nil else from :: range(from + 1, end)
```

たとえば List.range(2,n) は２以上 n 未満の全整数からなるリストを生成します。関数 isPrime は次のように簡単に定義できます。

```
def isPrime(n: Int) =
  List.range(2, n) forall (x => n % x != 0)
```

素数性の数学的定義が直接 Scala コードに翻訳されています。

演習 : filter を使って forall と exists を定義しなさい。

リストの畳み込み 他のよくある操作は、リストの要素をある演算子で結合することです。たとえば

```
sum(List(x1, ..., xn )) = 0 + x1 + ... + xn
product(List(x1, ..., xn )) = 1 * x1 * ... * xn
```

もちろん、どちらの関数も再帰的スキームで実装できます。

```
def sum(xs: List[Int]): Int = xs match {
  case Nil => 0
  case y :: ys => y + sum(ys)
}
def product(xs: List[Int]): Int = xs match {
  case Nil => 1
  case y :: ys => y * product(ys)
}
```

しかし、このプログラムスキームの一般化、具体的にはクラス List の reduceLeft メソッドを利用しても実装できます。このメソッドは与えられた二項演算子を、与えられたリストの隣りあう要素の間に挿入します。たとえば

```
List(x1, ..., xn ).reduceLeft(op) = (...(x1 op x2 ) op ... ) op xn
```

reduceLeft を使って、sum と product の共通パターンを明らかにできます。

```
def sum(xs: List[Int]) = (0 :: xs) reduceLeft {(x, y) => x + y}
def product(xs: List[Int]) = (1 :: xs) reduceLeft {(x, y) => x * y}
```

次が reduceLeft の実装です。

```
def reduceLeft(op: (A, A) => A): A = this match {
  case Nil => error("Nil.reduceLeft")
  case x :: xs => (xs foldLeft x)(op)
}
def foldLeft[B](z: B)(op: (B, A) => B): B = this match {
  case Nil => z
  case x :: xs => (xs foldLeft op(z, x))(op)
}
```

reduceLeft メソッドは、一般的で役に立つ別のメソッド foldLeft を使って定義されていることがわかります。後者は、追加のパラメータとして**積算器** z をとり、foldLeft が空リストに適用された時は、その値が返ります。つまり、

```
(List(x1, ..., xn ) foldLeft z)(op) = (...(z op x1 ) op ... ) op xn
```

メソッド sum と product は foldLeft を使って、次のようにも定義できます。

```
def sum(xs: List[Int]) = (xs foldLeft 0) {(x, y) => x + y}
def product(xs: List[Int]) = (xs foldLeft 1) {(x, y) => x * y}
```

右畳み込み foldLeft と reduceLeft を適用すると、左に傾いた木(left-leaning tree)ができます。双対的な foldRight と reduceRight では、右に傾いた木ができます。

```
List(x1, ..., xn ).reduceRight(op) = x1 op ( ... (xn-1 op xn )...)
(List(x1, ..., xn ) foldRight acc)(op) = x1 op ( ... (xn op acc)...)
```

それらは次のように定義されます。

```
def reduceRight(op: (A, A) => A): A = this match {
  case Nil => error("Nil.reduceRight")
  case x :: Nil => x
  case x :: xs => op(x, xs.reduceRight(op))
}
def foldRight[B](z: B)(op: (A, B) => B): B = this match {
  case Nil => z
  case x :: xs => op(x, (xs foldRight z)(op))
}
```

クラス List では、foldLeft と foldRight の省略形記号も定義しています。

```
def /:[B](z: B)(f: (B, A) => B): B = foldLeft(z)(f)
def :\[B](z: B)(f: (A, B) => B): B = foldRight(z)(f)
```

このメソッド名(記号)は、順スラッシュあるいは逆スラッシュによって、畳み込み操作の木を左/右に傾いた木の絵で表しています。それぞれの : はリスト引数を指し示し、スラッシュは積算器（あるいはゼロ）引数 z を指し示します。つまり

```
(z /: List(x1, ..., xn ))(op) = (...(z op x1 ) op ... ) op xn
(List(x1, ..., xn ) :\ z)(op) = x1 op ( ... (xn op acc)...)
```

結合則および交換則をみたす演算子(op)に対しては、/: と :\ は等価です（ただし、効率には違いがありますが)。

演習 9.4.2 リストを要素とするリストを引数とする関数 flatten を書く問題を考察しなさい。flatten の結果は、すべての要素リストを一つのリストへと連結したものです。次は :\ を用いた、このメソッドの実装です。

```
def flatten[A](xs: List[List[A]]): List[A] =
  (xs :\ (Nil: List[A])) {(x, xs) => x ::: xs}
```

flatten の本体を次のように置き換えた場合を考察しなさい。

```
((Nil: List[A]) /: xs) ((xs, x) => xs ::: x)
```

flatten の２つのバージョンでの漸近的計算量の違いはどうなりますか？

実際には、flatten は他の有用な関数と共に Scala ライブラリの List オブジェクトで定義されています。ユーザープログラムからは List.flatten でアクセスできます。flatten はクラス List のメソッドでないことに注意してください --- それでは意味がありません。なぜならそれは一般のリストにではなく、リストのリストにしか適用できないからです。

リストの逆転 再考 9.2 節で、メソッド reverse の実装と、その実行時間がリストの長さの二次式になることを見ました。ここで、線形なコストをもつ reverse の新しい実装を考えましょう。アイデアは、次のようなプログラムスキームに基づく foldLeft 操作を用いるこ

とです。

```
class List[+A] { ...
  def reverse: List[A] = (z? /: this)(op?)
```

あとは z? と op? 部分を埋めるだけです。例から推測してみましょう。

```
    Nil
= Nil.reverse // 仕様より
= (z /: Nil)(op) // reverse のテンプレートより
= (Nil foldLeft z)(op) // /: の定義より
= z // foldLeft の定義より
```

したがって z? は Nil でなくてはなりません。二つ目のオペランドを推測するために、長さが 1 のリストを考えます。

```
    List(x)
= List(x).reverse // 仕様より
= (Nil /: List(x))(op) // reverse のテンプレートと z = Nil より
= (List(x) foldLeft Nil)(op) // /: の定義より
= op(Nil, x) // foldLeft の定義より
```

したがって op(Nil, x) は List(x) に等しく、それは x :: Nil と等しい。これから op は :: で、オペランドを交換したものだと分かります。以上から、次のような、線形な計算量をもつ reverse の実装を得ます。

```
def reverse: List[A] =
  ((Nil: List[A]) /: this) {(xs, x) => x :: xs}
```

(注釈 : Nil の型アノテーションは、型推論させるために必要です)

演習 9.4.3 書かれていない式を埋めて、次の基本的なリスト操作定義を畳み込み操作として完成しなさい。

```
def mapFun[A, B](xs: List[A], f: A => B): List[B] =
  (xs :\ List[B]()){ ?? }
def lengthFun[A](xs: List[A]): int =
  (0 /: xs){ ?? }
```

ネストしたマップ 命令型言語において通常、ループのネストで書かれる多くの計算を、高階リスト操作関数を使って書くことができます。

例として次のような問題を考えましょう。与えられた正整数 n に対し、正整数 i と j のすべての組 (ただし 1 <= j < i < n で、i+j が素数)を求めなさい。たとえば n = 7 であれば対は、

```
   i    2  3  4  4  5  6  6
   j    1  2  1  3  2  1  5
-----------------------------
 i + j  3  5  5  7  7  7  11
```

問題を解く自然な方法は 2 つのステップからなります。最初のステップでは、1<=j < i < n を満たす整数の対 (i,j) の列を作ります。第二のステップでは、すべての対 (i,j) の列か

ら、i+j が素数であるものをフィルタします。

最初のステップをより詳細に見ましょう。対の列を生成する自然な方法は 3 つのサブステップからなります。最初に、1 以上 n未満の整数を i のために作ります。

次に、1 以上 n 未満の各 i に対して、(i,1) から (i,i-1) までの対のリストを作ります。これは range と map の組み合わせによって可能です。

```
List.range(1, i) map (x => (i, x))
```

最後に、すべてのサブリストを foldright と ::: を使って連結します。すべてを組み合わせると次の式が得られます。

```
List.range(1, n)
  .map(i => List.range(1, i).map(x => (i, x)))
  .foldRight(List[(Int, Int)]()) {(xs, ys) => xs ::: ys}
  .filter(pair => isPrime(pair._1 + pair._2))
```

マップの平坦化 マッピングと、マップで得られたサブリストの連結という組み合わせは、非常によく使うので、クラス List には特別なメソッドがあります。

```
abstract class List[+A] { ...
  def flatMap[B](f: A => List[B]): List[B] = this match {
    case Nil => Nil
    case x :: xs => f(x) ::: (xs flatMap f)
  }
}
```

flatMap を使うと「和が素数となる対」の式は、次のようにさらに簡潔に書けます。

```
List.range(1, n)
  .flatMap(i => List.range(1, i).map(x => (i, x)))
  .filter(pair => isPrime(pair._1 + pair._2))
```

9.5 まとめ

この章では、リストをプログラミングの基本的なデータ構造として紹介しました。リストはイミュータブル(変更不能)なので、関数型プログラミング言語では一般的なデータ型です。命令型言語における配列に匹敵する役割を持っています。しかし配列とリストではアクセス方法に大きな違いがあります。配列へのアクセスは常にインデックスによりますが、リストではインデックスは滅多に使いません。scala.List にはインデックスによるアクセスのための apply メソッドが定義されていましたが、この操作は配列の場合よりずっとコストがかかります（一定時間に対して線形)。通常、リストはインデックスの代わりに再帰的にアクセスされ、再帰のステップは、たいていの場合、リスト上のパターンマッチに基づきます。また、豊富な高階コンビネータ群により、リストに対する事前定義された計算パターンをインスタンス化できます。

10 For 内包表記

前の章では map, flatMap, filter のような高階関数がリストを扱う強力な道具となることを示しました。しかし時に、これらの関数が要求する抽象化レベルゆえに、プログラムが理解しにくくなることもあります。

理解しやすくするために、 Scala には高階関数適用の共通パターンを簡単にする特別な記法があります。この記法は数学における集合の内包表記と、C や Java のような命令型言語の for ループとの架け橋となるものです。また関係データベースのクエリ(問い合わせ)記法にもよく似ています。

最初の例として、name と age をフィールドにもつ人(person)のリスト persons があるとしましょう。そのリストの中で、年齢が 20 を超える人の名前を表示するのは、次のように書けます。

```
for (p <- persons if p.age > 20) yield p.name
```

これは高階関数 filter と map を使った次の式と等価です。

```
persons filter (p => p.age > 20) map (p => p.name)
```

for 内包表記は命令型言語における for ループにすこし似ていますが、各繰り返しの結果値をすべて集めてリストを作る点が違います。

一般的に、for 内包表記は次の形式です。

```
for ( s ) yield e
```

ここで s は**ジェネレータ**、**定義**、**フィルタ**のシーケンス(並び)です。**ジェネレータ**とは val x <- e という形（ただし e は値がリストとなる式）です。それによって x はリスト中の値で次々に束縛されます。**定義**とは、val x = e という形です。残りの内包表記において、x は e の値の名前として導入されます。**フィルタ**とは Boolean 型の式 f です。f が false となるような束縛はすべて無視されます。シーケンス s は常にジェネレータで始まります。もしいくつかのジェネレータがシーケンスに含まれる場合、後にあるジェネレータは前にあるものよりも先に変化します。

シーケンス s は丸括弧 () ではなく波括弧 {} で囲むこともでき、その場合はジェネレータ、定義、フィルタの間のセミコロンは省略できます。

for 内包表記を使う例を二つ示します。最初に、前の章の例を繰り返しましょう。与えられた正整数 n に対し、正整数 i と j のすべての組（ただし 1 <= j < i < n で、i+j が素数）を求めなさい。for 内包表記を使うと、この問題は次のように解けます。

```
for { i <- List.range(1, n)
      j <- List.range(1, i)
      if isPrime(i+j) } yield {i, j}
```

前に考えた map, flatMap, filter を使った解よりずっと明快であるのは間違いないでしょう。

二つ目の例として、２つのベクトル xs と ys のスカラー積の計算を考えましょう。for 内包表記を使って次のように書けます。

```
sum(for ((x, y) <- xs zip ys) yield x * y)
```

10.1 N クィーン問題

for 内包表記は組み合わせパズルを解くのに特に役立ちます。そのようなパズルの一つの例が８クイーン問題 : 標準的なチェス盤に、８個のクイーンをお互いにチェックしない（クイーンは他の駒が同じ行、列、斜めにある時にチェックできます）ように置け、です。この問題の解法を考えますが、一般化してチェス盤を任意の大きさにします。したがって問題は、n個のクイーンを n x n の大きさのチェス盤に置け、となります。

問題を解くには、クイーンは各行に置かなくてはならないことに注意しましょう。ですから、クイーンを各行に置き、その度に新しく置いたクイーンが、すでに置かれた他のクイーンからチェックされないことを確認します。探索の過程において、行 k のどの場所にクイーンを置いても、それが行 1 から行 k-1 までのどれかのクイーンによってチェックされるかもしれません。そのような場合にはその部分の探索を止め、列 1 から列 k-1 のクイーンの異なる配置で探索を続けます。

以上から、再帰的なアルゴリズムが示唆されます。サイズ n x n の盤に k-1 個のクイーンを置いた解がすでにあるとしましょう。そのような解は、長さ k-1 の列番号のリスト（1 から n の範囲の値）として表現できます。この部分解リストをスタックのように扱います。リストの最初は k-1 行のクイーンの列番号、二番目は k-2 行のクイーンの列番号です。スタックの底は、盤の最初の行のクイーンの列番号です。すべての解はリストのリストとして表現され、各要素が個々の解です。

さて、k 番目のクイーンを置くために、前の解にクイーンを一つ追加し、可能なすべての拡張を作ります。これは長さ k の次の解のリストとなります。このプロセスをチェス盤のサイズ n に達するまで繰り返します。このアルゴリズムは次の関数 placeQueens に表されます。

```
def queens(n: Int): List[List[Int]] = {
  def placeQueens(k: Int): List[List[Int]] =
    if (k == 0) List(List())
    else for { queens <- placeQueens(k - 1)
               column <- List.range(1, n + 1)
               if isSafe(column, queens, 1) } yield column :: queens
  placeQueens(n)
}
```

演習 10.1.1 次の関数を書きなさい。

```
def isSafe(col: Int, queens: List[Int], delta: Int): Boolean
```

この関数は与えられた列 col に置くクイーンが、すでに置かれているクイーンに対して安全か否かを判定します。ここで delta は、クイーンを置く行と、リスト中の最初のクイーンの行との差です。

10.2 For 内包表記によるクエリ

for 内包表記は、データベースのクエリ言語による一般的操作と本質的に同じです。たとえば、データベース books があり、本のリストとして表現されているとしましょう。ただし Book は次のように定義されるものとします。

```
case class Book(title: String, authors: List[String])
```

次はデータベースの小さなサンプルです。

```
val books: List[Book] = List(
  Book("Structure and Interpretation of Computer Programs",
       List("Abelson, Harold", "Sussman, Gerald J.")),
  Book("Principles of Compiler Design",
       List("Aho, Alfred", "Ullman, Jeffrey")),
  Book("Programming in Modula-2",
       List("Wirth, Niklaus")),
  Book("Introduction to Functional Programming"),
       List("Bird, Richard")),
  Book("The Java Language Specification",
       List("Gosling, James", "Joy, Bill", "Steele, Guy", "Bracha, Gilad")))
```

このとき、著者の姓が "Ullman" である本すべての書名を探すには、

```
for (b <- books; a <- b.authors if a startsWith "Ullman")
yield b.title
```

(ここで startsWith は java.lang.String のメソッド)。あるいは書名が "Program" で始まる本の書名を探すには

```
for (b <- books if (b.title indexOf "Program") >= 0)
yield b.title
```

あるいは、少なくとも二冊の本を書いたすべての著者の名前をデータベース上で探すには、

```
for (b1 <- books; b2 <- books if b1 != b2;
      a1 <- b1.authors; a2 <- b2.authors if a1 == a2)
yield a1
```

最後のコードはまだ完全ではありません。なぜなら著者が結果リストに複数回現れるからです。結果リストから重複した著者を除く必要があります。これは次の関数を使えばできます。

```
def removeDuplicates[A](xs: List[A]): List[A] =
  if (xs.isEmpty) xs
  else xs.head :: removeDuplicates(xs.tail filter (x => x != xs.head))
```

メソッド removeDuplicates の最後の式は for 内包表記を使って、等価に次のように表現できます。

```
xs.head :: removeDuplicates(for (x <- xs.tail if x != xs.head) yield x)
```

10.3 For 内包表記の変換

すべての for 内包表記は 3 つの高階関数 map, flatMap, filter で表現できます。これが翻訳のスキーム(枠組み)であり、Scala コンパイラも使っています。

・単純な **for**内包表記

```
for (x <- e) yield e'
```

これは次のように翻訳されます。

```
e.map(x => e')
```

・**for** 内包表記

```
for (x <- e if f; s) yield e'
```

ただし f はフィルタで、s は（空でも良い）ジェネレータあるいはフィルタの列。
これは次のように翻訳されます。

```
for (x <- e.filter(x => f); s) yield e'
```

そして後者の式に対して翻訳が続きます。

・**for** 内包表記

```
for (x <- e; y <- e'; s) yield e''
```

ただし s は（空でも良い）ジェネレータあるいはフィルタの列。
これは次のように翻訳されます。

```
e.flatMap(x => for (y <- e'; s) yield e'')
```

そして後者の式に対して翻訳が続きます。

たとえば、「和が素数になる整数の組」を例にとると、

```
for { i <- range(1, n)
      j <- range(1, i)
      if isPrime(i+j)
} yield {i, j}
```

この式を翻訳すると次が得られます。

```
range(1, n)
  .flatMap(i =>
    range(1, i)
      .filter(j => isPrime(i+j))
      .map(j => (i, j)))
```

逆に、関数 map, flatMap, filter を for 内包表記で表すこともできます。3 つの関数を今度は for 内包表記を使って実装してみます。

```
object Demo {
  def map[A, B](xs: List[A], f: A => B): List[B] =
    for (x <- xs) yield f(x)

  def flatMap[A, B](xs: List[A], f: A => List[B]): List[B] =
```

```
    for (x <- xs; y <- f(x)) yield y

  def filter[A](xs: List[A], p: A => Boolean): List[A] =
    for (x <- xs if p(x)) yield x
}
```

驚くことではありませんが、Demo.map の本体の for 内包表記の翻訳は、クラス List の map を呼び出します。同様に、Demo.flatMap と Demo.filter は、クラス List の flatMap と filterの呼び出しへ翻訳されます。

演習 10.3.1 次の関数を for を用いて定義しなさい。

```
def flatten[A](xss: List[List[A]]): List[A] =
  (xss :\ (Nil: List[A])) ((xs, ys) => xs ::: ys)
```

演習 10.3.2 次を高階関数を用いて翻訳しなさい。

```
for (b <- books; a <- b.authors if a startsWith "Bird") yield b.title
for (b <- books if (b.title indexOf "Program") >= 0) yield b.title
```

10.4 For ループ

for 内包表記は命令型言語の for ループに似ていますが、結果のリストを生成します。時には、結果のリストはいらないがリストに対するジェネレータとフィルタの柔軟性が欲しいことがあります。これは for 内包表記の変種、for ループで表現できます。

```
for ( s ) e
```

この構文は標準的な for 内包表記と同じですが、キーワード yield がありません。for ループは、ジェネレータとフィルタの列 s から生成される各要素に対して、式 e を実行します。

例として、次の式はリストのリストとして表現される行列の全要素を表示します。

```
for (xs <- xss) {
  for (x <- xs) print(x + "\t")
  println()
}
```

for ループからクラス List の高階メソッドへの翻訳は、for 内包表記の翻訳と似ていますが、より単純です。for 内包表記で map と flatMap へと翻訳するところで、for ループではそれらを foreach へ翻訳します。

10.5 For の一般化

for 内包表記の翻訳が、メソッド map、flatMap、filter の存在だけに依存していることを見てきました。したがって、リスト以外のオブジェクトを生成するジェネレータについても、同じ記法を適用できます。それらのオブジェクトは 3 つのキーとなる関数 map、flatMap、filter だけをサポートすればよいのです。

標準 Scala ライブラリには、それら 3 つのメソッドと for 内包表記をサポートする抽象化が他にもいくつかあります。それらのいくつかは後の章で目にするでしょう。プログラマは自分が定義する型について for 内包表記を使えるようにするために、この原則を利用できます。そのような型は、単にメソッド map、flatMap、filter だけが必要なのです。

このことが役に立つ例はいくつもあります。たとえばデータベース・インタフェース、XML 木、オプション値などです。

一つ注意すべき点を。for 内包表記の翻訳結果に型整合性があることは、自動的には保証されません。これを保証するためには、map、flatMap、filter の型が、クラス List 中のそれらメソッドの型と本質的に似ていることが必要です。

より正確に言えば、for 内包表記を可能にしたい、パラメータ化されたクラス C[A] があるとしましょう。すると C は次のような型を持った map、flatMap、filter を定義しなくてはなりません。

```
def map[B](f: A => B): C[B]
def flatMap[B](f: A => C[B]): C[B]
def filter(p: A => Boolean): C[A]
```

これらの型を Scala コンパイラの中で静的に強制することは、よい考えに思えます。たとえば for 内包表記をサポートするすべての型が、これらのメソッドを持つ標準トレイトを実装するよう要求するとかです(*1)。問題は、そのような標準的なトレイトはクラス C の同一性に対して抽象化、たとえば C を型パラメータとしてとるとか、をしなくてはならないことです。このパラメータは型コンストラクタであり、map や flatMap メソッドのシグネチャにおいて**複数の異なる**型に適用される、ということに注意すべきです。不運にも Scala の型システムはこのコンストラクトを表現するには弱過ぎます。なぜなら完全に適用される型である、型パラメータしか扱えないからです。

*1 プログラミング言語 Haskell には同様のコンストラクトがあり、この抽象化は "monad with zero" と呼ばれています。

11 ミュータブルな状態

これまで示してきたほとんどのプログラムには副作用(*1)はありませんでした。したがって、**時間**の記述は問題になりませんでした。終了するプログラムでは、アクションの列は同じ結果をもたらします。これは計算の置き換えモデルにも反映されており、書き換えステップは項のどこにでも適用可能で、終了する様な書き換えは同じ答えをもたらします。実際、この**合流性**はλ計算、関数型プログラミングの基礎をなす理論の深遠な結論です。

この章では副作用を持つ関数を導入し、その振る舞いを調べます。その結果、これまで用いてきた計算の置き換えモデルを根本から修正する必要に迫られるでしょう。

11.1 状態を持つオブジェクト

我々は世界をオブジェクトの集合としてとらえ、オブジェクトの中には時間と共に変化する状態を持つものもあります。通常、状態は計算の過程で変化しうる変数の集合と結びついています。プログラミング言語の特定の構文に触れずに、状態を抽象化して言うと、「もしオブジェクトの振る舞いがその履歴に影響されるなら、オブジェクトは**状態を持つ**（**ステートフル**である)。」となります。

たとえば、銀行口座オブジェクトは状態を持ちます。なぜなら「100スイスフラン引き出せるか？」は口座の存続期間、異なる答を持ちうるからです。

Scala ではすべてのミュータブルな状態は、結局のところ、変数から作られます。変数定義は値定義と同じように書きますが、val の代わりに var で始まります。たとえば、次の２つの定義は２つの変数、x と count を導入し初期化します。

```
var x: String = "abc"
var count = 111
```

値定義と同様に変数定義は、名前と値を結びつけます。しかし変数定義の場合、この結びつきは後から代入によって変更できます。そのような代入は C や Java と同じように書きます。たとえば

```
x = "hello"
count = count + 1
```

Scala では定義する変数はすべて、定義の時点で初期化しければなりません。たとえば文、 var x: Int; は変数定義とみなされません。なぜなら初期値が無いからです(*2)。適切な初期値を知らない、あるいは気にしない場合は、代わりにワイルドカードを使えます。たとえば、

```
val x: T = _
```

---

*1 プログラムの中には標準出力に出力するものもあり、厳密にはそれは副作用 である、ということをここでは無視します。

*2 もしこのような文がクラス中に現れたなら、そうではなく変数宣言とみなされ、変数に対する抽象アクセスメソッドを導入しますが、これらのメソッドを 状態とは結びつけません。

は、x を何かデフォルト値（参照型には null を、論理型には false を、数値型には適切な 0 を）で初期化します。

実世界のオブジェクトは、Scala では変数をメンバに持つオブジェクトとして表現されます。たとえば銀行口座を表現するクラスです。

```
class BankAccount {
  private var balance = 0
  def deposit(amount: Int) {
    if (amount > 0) balance += amount
  }
  def withdraw(amount: Int): Int =
    if (0 < amount && amount <= balance) {
      balance -= amount
      balance
    } else error("insufficient funds")
}
```

このクラスは、口座の現在の残高を入れておく変数 balance を定義しています。メソッド deposit と withdraw は、代入によってこの変数の値を変更します。クラス BankAccount において、balance が private であることに注意して下さい。この結果、クラスの外からは直接アクセスできません。

銀行口座を作成するには、通常のオブジェクト生成の記法を使います。

```
val myAccount = new BankAccount
```

Example 11.1.1 次は銀行口座を扱う scala セッションです。

```
scala> :l bankaccount.scala
Loading bankaccount.scala...
defined class BankAccount
scala> val account = new BankAccount
account: BankAccount = BankAccount$class@1797795
scala> account deposit 50
unnamed0: Unit = ()
scala> account withdraw 20
unnamed1: Int = 30
scala> account withdraw 20
unnamed2: Int = 10
scala> account withdraw 15
java.lang.Error: insufficient funds
   at scala.Predef$error(Predef.scala:74)
   at BankAccount$class.withdraw(<console>:14)
   at <init>(<console>:5)
scala>
```

この例は、同じ操作（withdraw 20）を口座に２回適用して、異なる結果が生じることを示します。したがって口座がステートフルなオブジェクトであることは明らかです。

同一性と変化 代入は２つの式が「同じ」かを判定する際に新しい問題をもたらします。もし代入が排除されている場合には、次のよう、

```
val x = E; val y = E
```

ただし E は何か任意の式であるとする、と書いた場合、x と y は無理なく同じと考えられます。つまり次のように書いても等価です。

```
val x = E; val y = x
```

(この性質は通常、**参照透過性**と呼ばれます）しかしもし代入を許したなら、上の２つの定義は異なります。次を考えて下さい。

```
val x = new BankAccount; val y = new BankAccount
```

この問題、x と y は同一であるか、に答えるには、「同一」の意味をより明確にしなければなりません。この意味は**操作的等価性**としてとらえられ、くだけた言い方をするなら、次のようになります。

仮に x と y の２つの定義があるとします。x と y とが同じ値を定義しているか調べるには、次のようにします。

- 定義を実行し、引き続いて x と y とを含む任意の操作列 S を実行し、（もしあれば）結果を調べる。
- 次いで、定義を実行し、S 中のすべての y の出現を x で置き換えた、 S から得られた列 S' を実行する。
- もし S' を実行した結果が異なれば、x と y は確かに異なる。
- もしすべての可能な操作列の組 {S, S'} に対して、同じ結果が得られるなら、 x と y は同一である。

別の言い方をすれば、操作的等価性は２つの定義 x と y を、もしどんな実験でも x と y を区別できないなら、同じ値だとみなします。この文脈における実験とは、x あるいは y を用いる任意の２つのプログラムです。

この定義を用いて、

```
val x = new BankAccount; val y = new BankAccount
```

が、値 x と y を等しく定義するか確かめましょう。定義を再度行い、次いでテスト列を。

```
> val x = new BankAccount
> val y = new BankAccount
> x deposit 30
30
> y withdraw 20
java.lang.RuntimeException: insufficient funds
```

さて、y のすべての出現を x に置き替えます。すると

```
> val x = new BankAccount
> val y = new BankAccount
> x deposit 30
30
> x withdraw 20
10
```

最終結果が異なるので、x と y が異なることが証明されました。その一方で、もし

```
val x = new BankAccount; val y = x
```

と定義すると、どんな操作も x と y を区別できません。したがってこの場合、x と y は等しくなります。

代入と置き換えモデル これらの例が示すことは、以前の計算の置き換えモデルは、もう使えないということです。結局のところ、このモデルでは変数の名前を定義式で常に置き換え可能なのですから。たとえば

```
val x = new BankAccount; val y = x
```

の、y の定義中の x は new BankAccount で置き換え可能です。しかしその変更は異なるプログラムを導くことを見てきました。したがって置き換えモデルは、代入を加えると、妥当ではありません。

11.2 命令型制御構造

Scala には C や Java で知られる while や do-while のループ構文があります。また else-節のない片側の if や、関数を途中で中止する return 文もあります。これらを使えば、従来の命令型スタイルのプログラミングも可能です。たとえば次の関数は、与えられたパラメータ x の n 乗を計算しますが、while と片側 if を使って実装されています。

```
def power(x: Double, n: Int): Double = {
  var r = 1.0
  var i = n
  var j = 0
  while (j < 32) {
    r = r * r
    if (i < 0)
      r *= x
    i = i << 1
    j += 1
  }
  r
}
```

これらの命令的な制御構文は、ユーザの便宜を図って言語に入れてあります。それらを無くすこともできました。なぜなら同じ構文を単に関数として実装できるからです。たとえば while ループを関数的に実装してみましょう。whileLoop は 2 つのパラメータ、Boolean 型の条件と Unit 型のコマンド、をとる関数のはずです。条件とコマンドは名前渡しである必要があります。そうすれば各ループ毎に繰り返し評価されます。以上から、次の whileLoop 定義を得ます。

```
def whileLoop(condition: => Boolean)(command: => Unit) {
  if (condition) {
    command; whileLoop(condition)(command)
  } else ()
}
```

whileLoop は末尾再帰的であり、一定のスタック領域で動作することに注意しましょう。

演習 11.2.1 repeatLoop 関数を書きなさい。次のように適用されるものとします。

```
repeatLoop { command } ( condition )
```

また次のようなループ構文は可能ですか？

```
repeatLoop { command } until ( condition )
```

C や Java で知られるいくつかの制御構文は Scala にはありません。ループ中にジャンプする break や continue はありません。また Java の意味での for ループもありません。それらは 10.4 節で議論した、より一般的な for ループ構文で置き換えられています。

11.3 高度な例 : 離散イベントシミュレーション

代入と高階関数がどのように興味深い形で結びつくか、例をとおして考えます。ここでは、デジタル回路シミュレータを構築します。

この例は Abelson と Sussman の本[ASS96]から借りました。彼らの基本的な（Scheme の）コードを、継承でコードを再利用できるオブジェクト指向によって拡張しています。この例は、一般的な離散イベントのシミュレーションプログラムが、どのように構造化され構築されるかも示します。

デジタル回路を記述する小さな言語から始めます。デジタル回路は**結線**と **function box** から構築されます。結線は信号を運び、function box は信号を変換します。信号は ture と false の真偽値で表現します。

次は基本的な function box です。

- **インバーター** : 信号を反転します。
- **AND ゲート** : 入力の論理積を出力に設定します。
- **OR ゲート** : 入力の論理和を出力に設定します。

ほかの function box は、これらの組み合わせで作れます。

ゲートには**遅延**があり、ゲートの出力変化は、入力からいくらか時間が経ってから起きます。

デジタル回路用言語 デジタル回路の要素を、次の Scala クラスと関数によって記述します。まず、結線を表す Wire クラスです。結線は次のようにして生成できます。

```
val a = new Wire
val b = new Wire
val c = new Wire
```

次に、手続きです。

```
def inverter(input: Wire, output: Wire)
def andGate(a1: Wire, a2: Wire, output: Wire)
def orGate(o1: Wire, o2: Wire, output: Wire)
```

これらの手続きは、われわれが必要とする基本ゲートを（副作用として）「作成」します。これらを使ってもっと複雑な function box を作れます。たとえば半加算器は、次のように

定義できます。

```
def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire) {
  val d = new Wire
  val e = new Wire
  orGate(a, b, d)
  andGate(a, b, c)
  inverter(c, e)
  andGate(d, e, s)
}
```

この抽象化は、それ自体で使用できます。たとえば、全加算器の定義で

```
  def fullAdder(a: Wire, b: Wire, cin: Wire, sum: Wire, cout: Wire) {
    val s = new Wire
    val c1 = new Wire
    val c2 = new Wire
    halfAdder(a, cin, s, c1)
    halfAdder(b, s, sum, c2)
    orGate(c1, c2, cout)
}
```

クラス Wire および関数 inverter、andGate、orGate は、このようにユーザーがデジタル回路を定義できる小さな言語を表現しています。では、回路をシミュレートできるように、このクラスと関数群に実装を与えましょう。実装は、離散イベントシミュレーション用のシンプルで一般的な API に基づきます。

シミュレーション API 離散イベントシミュレーションは、ユーザーが定義した**アクション**を指定された**時刻**に実行します。**アクション**は、パラメータをとらず Unit を返す関数として表されます。

```
type Action = () => Unit
```

時刻はシミュレーション上のものであり、「壁時計」が指す現実の時刻ではありません。

具体的なシミュレーションは、抽象クラス Simulation を継承したオブジェクトの内部で行われます。このクラスは次のシグネチャを持ちます。

```
abstract class Simulation {
   def currentTime: Int
   def afterDelay(delay: Int, action: => Action)
   def run()
}
```

currentTime は、現在のシミュレーション上の時刻を整数値として返します。afterDelay は、currentTime から指定された時間が経過したときにアクションが実行されるよう、スケジュールします。run は、実行するアクションがなくなるまでシミュレーションを実行します。

結線クラス 結線は 3 つの基本的なアクションをサポートする必要があります。

getSignal : 結線の現在の信号を返します。

setSignal(sig:Boolean) : 結線の信号を sig に更新します。

addAction(p:Action) : 指定された手続き p を結線の**アクション**に付加します。結線の信号が変更されるたびに、 付加されたすべてのアクション手続きを実行します。

Wire クラスの実装例を示します。

```
class Wire {
  private var sigVal = false
  private var actions: List[Action] = List()
  def getSignal = sigVal
  def setSignal(s: Boolean) =
  if (s != sigVal) {
      sigVal = s
      actions.foreach(action => action())
  }
  def addAction(a: Action) {
      actions = a :: actions; a()
  }
}
```

２つのプライベート変数が結線の状態を作り上げます。変数 sigVal は結線の現在の信号を表し、変数 actions は現在結線に付加されているアクション手続きを表します。

インバータクラス インバータ(Not 回路)を、入力の結線にアクションをインストールすることで実装します。アクションは入力信号を反転して出力信号とします。アクションが効果を現すのは、入力が変化してからシミュレーション時間 InverterDelay 経過後でなくてはなりません。以上より、次の実装が導かれます。

```
def inverter(input: Wire, output: Wire) {
  def invertAction() {
    val inputSig = input.getSignal
    afterDelay(InverterDelay) { output setSignal !inputSig }
  }
  input addAction invertAction
}
```

AND ゲートクラス AND ゲートは、インバーターの類推で実装できます。andGate のアクションは、入力信号の論理積を出力することです。それは、２つの入力のいずれかが変化してから、シミュレーション時間 AndGateDelay 経過後におきる必要があります。実装は次のようになります。

```
def andGate(a1: Wire, a2: Wire, output: Wire) {
  def andAction() {
    val a1Sig = a1.getSignal
    val a2Sig = a2.getSignal
    afterDelay(AndGateDelay) { output setSignal (a1Sig & a2Sig) }
  }
  a1 addAction andAction
  a2 addAction andAction
}
```

演習 11.3.1 OR ゲートの実装を書きなさい

演習 11.3.2 OR ゲートの別の定義方法として、インバーターと AND ゲートを組み合わせる方法があります。関数 orGate を、andGate とinverter を使って定義しなさい。また、この関数の遅延(delay)時間は？

シミュレーションクラス さあ、あとは Simulation クラスを実装すれば出来あがりです。考え方としては、我々が Simulation オブジェクト内部の面倒を見て、**agenda**(予定表)のアクションを実行するようにしてやります。agenda はアクションと、実行すべき時刻のペアのリストとして表現されます。agenda リストは実行すべき時刻でソートしておいて、早いアクションが先にくるようにします。

```
abstract class Simulation {
  case class WorkItem(time: Int, action: Action)
  private type Agenda = List[WorkItem]
  private var agenda: Agenda = List()
```

また、プライベート変数 curtime によって、シミュレーション上の時刻を追跡します。

```
  private var curtime = 0
```

メソッド afterDelay(delay, block) を適用すると、要素 WorkItem(currentTime + delay, () => block) は agenda リストの適切な場所に挿入されます。

```
  private def insert(ag: Agenda, item: WorkItem): Agenda =
    if (ag.isEmpty || item.time < ag.head.time) item :: ag
    else ag.head :: insert(ag.tail, item)

  def afterDelay(delay: Int)(block: => Unit) {
    val item = WorkItem(currentTime + delay, () => block)
    agenda = insert(agenda, item)
  }
```

メソッド run を適用すると、agenda から要素を取り除いてそのアクションを実行する、ということが繰り返されます。それは agenda が空になるまで続きます。

```
  private def next() {
    agenda match {
      case WorkItem(time, action) :: rest =>
        agenda = rest; curtime = time; action()
      case List() =>
    }
  }

  def run() {
    afterDelay(0) { println("*** simulation started ***") }
    while (!agenda.isEmpty) next()
  }
```

シミュレータの実行 シミュレータを走らせる前に、結線上の信号変化を追跡する方法を用意します。そのための関数 probe を書きます。

```
  def probe(name: String, wire: Wire) {
    wire addAction {
```

```
      println(name + " " + currentTime + " new_value = " + wire.getSignal)
    }
  }
```

さあ、シミュレータを動かします。まず、４つの結線を定義し、そのうちの二つに probe をセットしましょう。

```
scala> val input1, input2, sum, carry = new Wire

scala> probe("sum", sum)
sum 0 new_value = false
scala> probe("carry", carry)
carry 0 new_value = false
```

半加算器の回路を定義してみましょう。

```
scala> halfAdder(input1, input2, sum, carry)
```

最後に、２つの入力結線の信号を順番に true に設定して、シミュレータを走らせましょう。

```
scala> input1 setSignal true; run
*** simulation started ***
sum 8 new_value = true

scala> input2 setSignal true; run
carry 11 new_value = true
sum 15 new_value = false
```

11.4 まとめ

この章では、Scala において状態をモデル化する手段 --- 変数、代入、そして手続き的な制御構造 --- を見てきました。状態と代入は、計算に関する我々のメンタルモデルを複雑にします。特に、参照透過性は失われます。一方で、代入はプログラムをエレガントに書く新しい方法を提供します。いつものことですが、純粋関数型プログラミングと代入を使うプログラミングの、どちらがベストであるかは状況によります。

12 ストリームによる計算

前の章で、変数、代入、および状態を持つオブジェクトを紹介しました。時間とともに変化する実世界のオブジェクトを、計算において変数の状態を変化させることで、モデル化できることを見ました。実世界では時刻は変化します。そのように、プログラム実行における時刻の変化によって、実世界の時刻変化をモデル化できます。もちろん、そのような時刻変化は、通常伸びたり縮んだりしますが、相対的な順番は守られます。これはまったく自然に見えますが、注意すべき大事なことがあります。 いったん変数と代入を導入すると、我々のシンプルでパワフルな関数ベースの計算の置き換えモデルは、もはや適用できないのです。

ほかに方法はないのでしょうか？ 実世界の状態変化を、状態を持たない関数を使ってモデル化できないでしょうか？ 数学の導きによると、答えは明らかに Yes です。時間変化する量は、時刻 t をパラメータにとる関数 f(t) によって、シンプルにモデル化できます。モデル化だけでなく計算においても、これはうまくいきます。変数を次々と書き換える代わりに、それらすべての値をリストの連続的な要素として表現できます。つまり、ミュータブルな変数 var x: T は、イミュータブルの値 val x: List[T] で置き換えることができます。ある意味、空間と時間の取引です。変数に代入される様々な値は、リスト中に異なる要素として同時に存在することになります。リストベースモデルの利点の一つは、「タイムトラベル」、つまり変数に代入される連続的な値を同時に見ること、ができることです。ほかの利点としては、強力なリスト処理関数ライブラリを利用して、しばしば計算をシンプルにできることです。たとえば、特定の範囲にある素数すべての和を計算する、命令型プログラムを考えてみましょう。

```
def sumPrimes(start: Int, end: Int): Int = {
  var i = start
  var acc = 0
  while (i < end) {
    if (isPrime(i)) acc += i
    i += 1
  }
  acc
}
```

変数 i が、範囲[start .. end-1]のすべての値を「経験」していることに注意してください。

より関数的な方法では、変数 i の値のリストを range(start, end) によって直接に表現します。

```
def sumPrimes(start: Int, end: Int) =
  sum(range(start, end) filter isPrime)
```

あきらかにプログラムは短くて明快になりました！しかし、この関数型プログラムは効率の点でかなり劣ります。範囲内のすべての数からなるリストを作り、さらにそのうちの素数すべてからなるリストを作るからです。効率の点ではさらに悪いことがあります。次の例を見てください。

1000から10000の間の二番目の素数を見つける。

```
range(1000, 10000) filter isPrime at 1
```

これは、1000 から 10000 までのすべての数からなるリストが作りますが、そのリストのほとんどの要素は顧みられません！

しかし、あるトリックによって、これらのような例を効率的に実行できます。

> シーケンスの後の要素が実際には計算に必要なければ、その計算を
> 回避できるのです。

このようなシーケンスのために新しいクラスを定義します。それを Stream と呼びます。Stream は定数 empty とコンストラクタ cons を使って生成します。これらは scala.Stream モジュールで定義されています。たとえば、次の式は 1 と 2 を要素とするストリームを生成します。

```
Stream.cons(1, Stream.cons(2, Stream.empty))
```

ほかの例として、List.range の類似物、ただしリストの代わりにストリームを返すものは、次のようになります。

```
def range(start: Int, end: Int): Stream[Int] =
  if (start >= end) Stream.empty
  else Stream.cons(start, range(start + 1, end))
```

(この関数は、上のモジュール Stream でも定義されています) Stream.range と List.range は似ていますが、その実行時の振る舞いはまったく違います。

Stream.range は、最初の要素が start である Stream オブジェクトを直ちに返します。そのほかのすべての要素は、tail メソッド呼び出しによって、それらが**必要**になったときにのみ計算されます (tail メソッドはまったく呼ばれないかもしれません)。

ストリームは単なるリストとしてアクセスされます。リストと同様、基本的なアクセスメソッドは isEmpty と head と tail です。たとえば次のようにして、ストリームのすべての要素を表示できます。

```
def print(xs: Stream[A]) {
  if (!xs.isEmpty) { Console.println(xs.head); print(xs.tail) }
}
```

ストリームは、リストに対して定義されている他のほぼすべてのメソッドもサポートしています (これらがサポートするメソッドの差分については、下記を参照してください)。たとえば、1000 から 10000 の範囲のストリームに filter と apply を適用して、1000 から 10000 の間の二番目の素数を見つけることができます。

```
Stream.range(1000, 10000) filter isPrime at 1
```

先のリストベースの実装との違いは、もはや不必要に三番目以降を構築して素数判定しないことです。

ストリームの CONS と連結 クラス List のメソッドのうち、クラス Stream ではサポートされていないものは、:: と ::: の 2 つです。理由は、これらのメソッドは右側の引数に対して呼ばれるからです。右側の引数に対して呼ばれるということは、その引数は、メソッドが呼ばれる前に評価される必要があるということです。たとえばリストの x :: xs の場合、後部 xs は :: が呼ばれる前に評価される必要があり、新しいリストが構築される場合があります。これはストリームではうまくいきません。ストリームの後部は、それが tail オペレーションによって必要となるまでは評価されてはなりません。リストの連結 ::: をストリームに持ち込めないのも、同じ理由です。

x :: xs の代わりに、最初の要素 x と (未評価の) 後部 xs からなるストリームを構築するには、Stream.cons(x, xs) を使います。xs ::: ys の代わりに、オペレーション xs append ys を使います。

13 イテレータ

イテレータは、ストリームの命令型版です。ストリームのように、イテレータには無限リストを記述する能力があります。しかし、イテレータの成分を含むデータ構造はありません。代わりに、２つの抽象メソッド next と hasNext を使って、イテレータはシーケンスをたどることができます。

```
trait Iterator[+A] {
  def hasNext: Boolean
  def next: A
```

メソッド next は、次の要素を返します。メソッド hasNext は next で返すべき要素がまだあるかどうかを示します。イテレータは他にもいくつかメソッドをサポートしていますが、それは後ほど説明します。

例として、1 から 100 までの数の平方を表示してみます。

```
val it: Iterator[Int] = Iterator.range(1, 100)
while (it.hasNext) {
  val x = it.next
  println(x * x)
}
```

13.1 イテレータメソッド

イテレータは、next と hasNext のほかにも豊富なメソッドをサポートしており、それを次で説明します。それらメソッドの多くは、リスト機能の対応するものに似ています。

Append メソッド append は、新しいイテレータを構築します。構築されるイテレータは、元のイテレータの最後まで到達すると、与えられたイテレータで継続します。

```
def append[B >: A](that: Iterator[B]): Iterator[B] = new Iterator[B] {
  def hasNext = Iterator.this.hasNext || that.hasNext
  def next = if (Iterator.this.hasNext) Iterator.this.next else that.next
}
```

append 定義にある Iterator.this.next という項と Iterator.this.hasNext という項は、それを取り囲む Iterator クラスで定義されている、対応するメソッドを呼び出します。もし、this に対するプレフィックス Iterator がなければ、hasNext と next は、append の結果(オブジェクト)に定義されているメソッド自身を再帰的に呼び出してしまいます。これは我々の望むことではありません。

Map、FlatMap、Foreach メソッド map は、元のイテレータのすべての要素を、与えられた関数 f で変換して返すイテレータを構築します。

```
def map[B](f: A => B): Iterator[B] = new Iterator[B] {
  def hasNext = Iterator.this.hasNext
```

```
    def next = f(Iterator.this.next)
  }
```

メソッド flatMap はメソッド map と似ていますが、変換する関数 f がイテレータを返す点が違います。flatmap の結果は、f を順に呼んでいって返されてくるイテレータ達を結合したものです。

```
  def flatMap[B](f: A => Iterator[B]): Iterator[B] = new Iterator[B] {
    private var cur: Iterator[B] = Iterator.empty
    def hasNext: Boolean =
      if (cur.hasNext) true
      else if (Iterator.this.hasNext) { cur = f(Iterator.this.next); hasNext }
      else false
    def next: B =
      if (cur.hasNext) cur.next
      else if (Iterator.this.hasNext) { cur = f(Iterator.this.next); next }
      else error("next on empty iterator")
  }
```

map に深く関係するのが foreach メソッドです。与えられた関数をイテレータのすべての要素に適用しますが、結果のリストを構築しません。

```
  def foreach(f: A => Unit): Unit =
    while (hasNext) { f(next) }
```

Filter メソッド filter は、元のイテレータのすべての要素のうち、基準 p を満たすものを返すイテレータを構築します。

```
  def filter(p: A => Boolean) = new BufferedIterator[a] {
    private val source =
      Iterator.this.buffered
    private def skip
      { while (source.hasNext && !p(source.head)) { source.next } }
    def hasNext: Boolean =
      { skip; source.hasNext }
    def next: A =
      { skip; source.next }
    def head: A =
      { skip; source.head }
  }
```

filter はイテレータの「バッファされた」サブクラスのインスタンスを返します。BufferedIterator オブジェクトは、そのほかに head メソッドを持っています。このメソッドは、head（訳注 : next の誤り？）が返すはずの要素を返しますが、next のように、要素をそれ以降に進めることはありません。そのため、head が返す要素は、次の head または next の呼び出しで再び返されます。BufferedIterator トレイトの定義は次のとおりです。

```
  trait BufferedIterator[+A] extends Iterator[A] {
    def head: A
  }
```

map、flatmap、filter、foreach がイテレータに存在するので、for 内包表記 と for ループ がイテレータに対しても使えます。たとえば、1 から 100 までの数の平方を表示する適

用は、次のように等価に表現できます。

```
for (i <Iterator.range(1, 100))
  println(i * i)
```

Zip メソッド zip は、他のイテレータをとって、２つのイテレータから返される要素のペアからなるイテレータを返します。

```
    def zip[B](that: Iterator[B]) = new Iterator[(A, B)] {
      def hasNext = Iterator.this.hasNext && that.hasNext
      def next = {Iterator.this.next, that.next}
    }
  }
```

13.2 イテレータの構築

具体的なイテレータは、クラス Iterator の２つの抽象メソッド next と hasNext の実装を提供する必要があります。もっとも単純なイテレータは、常に空の列を返す Iterator.empty です。

```
object Iterator {
  object empty extends Iterator[Nothing] {
    def hasNext = false
    def next = error("next on empty iterator")
  }
```

もっと面白みのあるイテレータは、配列のすべての要素を列挙するものです。このイテレータはオブジェクト Iterator で定義されている fromArray メソッドで構築されます。

```
def fromArray[A](xs: Array[A]) = new Iterator[A] {
  private var i = 0
  def hasNext: Boolean =
    i < xs.length
  def next: A =
    if (i < xs.length) { val x = xs(i); i += 1; x }
    else error("next on empty iterator")
}
```

ほかのイテレータとして、範囲内の整数を列挙するものがあります。Iterator.range 関数は、与えられた範囲の整数値をたどるイテレータを返します。

```
object Iterator {
  def range(start: Int, end: Int) = new Iterator[Int] {
    private var current = start
    def hasNext = current < end
    def next = {
      val r = current
      if (current < end) current += 1
      else error("end of iterator")
```

```
      r
    }
  }
}
```

ここまで見てきたイテレータはいずれ終わりますが、永遠に続くイテレータも定義できます。たとえば、次のイテレータは初期値からずっと続く整数を返します(*1)。

```
def from(start: Int) = new Iterator[Int] {
  private var last = start 1
  def hasNext = true
  def next = { last += 1; last }
}
```

13.3 イテレータの使用

イテレータの使い方の例をさらに２つあげます。最初は、配列 xs: Array[Int]のすべての要素を表示するもので、次のように書けます。

```
Iterator.fromArray(xs) foreach (x => println(x))
```

または for 内包表記を使っても書けます。

```
for (x <Iterator.fromArray(xs))
  println(x)
```

二つ目の例として、double の配列のうち、要素がある値より大きいすべてのインデックスを求めることを考えます。インデックスはイテレータとして返されるものとします。これは、次の式で実現できます。

```
import Iterator._
fromArray(xs)
.zip(from(0))
.filter(case (x, i) => x > limit)
.map(case (x, i) => i)
```

または for 内包表記を使っても実現できます。

```
import Iterator._
for ((x, i) <fromArray(xs) zip from(0); x > limit)
yield i
```

*1 int 型が有限の表現であるため、2^{31}で数は元に戻ります。

14 遅延評価val

遅延評価val（lazy value）は、値の初期化を最初にアクセスされるまで遅延させる方法です。これは実行中に必要とならないかもしれない、計算コストが高い値を扱う場合に有用です。最初の例として、従業員のデータベースを考えましょう。各従業員ごとにマネージャーとチームが決まっています。

```
case class Employee(id: Int,
                    name: String,
                    managerId: Int) {
  val manager: Employee = Db.get(managerId)
  val team: List[Employee] = Db.team(id)
}
```

上記の Employee クラスは、直ちにそのフィールドをすべて初期化し、従業員テーブル全体をメモリにロードします。これは明らかに最善ではなく、フィールドを lazy にすることで簡単に改善できます。このようにして、データベースアクセスを、本当に必要になるまで、また、初めて必要になるまで遅らせます。

```
case class Employee(id: Int,
                    name: String,
                    managerId: Int) {
  lazy val manager: Employee = Db.get(managerId)
  lazy val team: List[Employee] = Db.team(id)
}
```

実際に何が起きているか、いつレコードがフェッチされるか表示するモックアップのデータベースを使って、見てみましょう。

```
object Db {
  val table = Map(1 -> (1, "Haruki Murakami", 1),
                  2 -> (2, "Milan Kundera", 1),
                  3 -> (3, "Jeffrey Eugenides", 1),
                  4 -> (4, "Mario Vargas Llosa", 1),
                  5 -> (5, "Julian Barnes", 2))

  def team(id: Int) = {
    for (rec <- table.values.toList; if rec._3 == id)
      yield recToEmployee(rec)
  }

  def get(id: Int) = recToEmployee(table(id))

  private def recToEmployee(rec: (Int, String, Int)) = {
    println("[db] fetching " + rec._1)
    Employee(rec._1, rec._2, rec._3)
  }
}
```

一人の従業員を取り出すプログラムを実行すると、確かにデータベースは遅延評価Val を参照するときのみアクセスされることが、出力によって確認できます。

ほかの 遅延評価val の使い方は、いくつかのモジュールからなるアプリケーションの初期化の順番を解決することです。遅延評価Val が導入される前は、同様のことを object 定義を使用することで実現していました。二つ目の例として、いくつかのモジュールからなるコンパイラを考えてみましょう。最初に、シンボルのためのクラスと２つの事前定義された関数を定義している、単純なシンボルテーブルを見てみます。

```
class Symbols(val compiler: Compiler) {
  import compiler.types._

  val Add = new Symbol("+", FunType(List(IntType, IntType), IntType))
  val Sub = new Symbol("-", FunType(List(IntType, IntType), IntType))

  class Symbol(name: String, tpe: Type) {
  override def toString = name + ": " + tpe
  }
}
```

symbols モジュールは、Compiler インスタンスでパラメータ化されています。Compiler インスタンスは、types モジュールなどのほかのサービスへのアクセスを提供します。この例では、事前定義された関数が２つ（加算と減算）だけあり、それらの定義は types モジュールに依存しています。

```
class Types(val compiler: Compiler) {
  import compiler.symtab._

  abstract class Type
  case class FunType(args: List[Type], res: Type) extends Type
  case class NamedType(sym: Symbol) extends Type
  case object IntType extends Type
}
```

２つのコンポーネントをつなぐため、コンパイラオブジェクトを作成して、２つのコンポーネントへ引数として渡します。

```
class Compiler {
  val symtab = new Symbols(this)
  val types = new Types(this)
}
```

残念ながら、この実直的なアプローチは実行時に失敗します。symtab モジュールが types モジュールを必要としているからです。一般的に、モジュール間の依存は複雑になりがちで、正しい順番で初期化するのは難しく、循環があるために不可能なことさえあります。簡単な対処は、そのようなフィールドを lazy にして、正しい順番は compiler に任せてしまうことです。

```
class Compiler {
  lazy val symtab = new Symbols(this)
  lazy val types = new Types(this)
}
```

これで、２つのモジュールは最初のアクセスで初期化され、compiler は期待通りに動くでしょう。

構文

lazy 修飾子は、具体的な値定義でのみ指定できます。値定義におけるすべての型付け規則が遅延評価valにも適用されますが、ひとつだけ制限が取り払われています。それは、再帰的なローカル値が許されることです。

15 暗黙のパラメータと型変換

暗黙のパラメータと暗黙の型変換は、既存ライブラリのカスタマイズや高レベルの抽象に役立つ強力なツールです。例として、未特定の演算 add を持つ半群の抽象クラスからはじめましょう。

```
abstract class SemiGroup[A] {
  def add(x: A, y: A): A
}
```

SemiGroup に unit 要素を加えたサブクラス Monoid はこうなります。

```
abstract class Monoid[A] extends SemiGroup[A] {
  def unit: A
}
```

次はモノイドの２つの実装です。

```
object stringMonoid extends Monoid[String] {
  def add(x: String, y: String): String = x.concat(y)
  def unit: String = ""
}

object intMonoid extends Monoid[Int] {
  def add(x: Int, y: Int): Int = x + y
  def unit: Int = 0
}
```

任意のモノイド上で動作する sum メソッドは、純粋な Scala で次のように書けます。

```
def sum[A](xs: List[A])(m: Monoid[A]): A =
  if (xs.isEmpty) m.unit
  else m.add(xs.head, sum(m)(xs.tail)
```

この sum メソッドは、次のように呼び出せます。

```
sum(List("a", "bc", "def"))(stringMonoid)
sum(List(1, 2, 3))(intMonoid)
```

これらはみんな動作しますが、いまいちです。問題は、それを使うコードすべてにモノイドの実装を渡さなくてはならないことです。型引数を推論してくれるのと同様に、システムが自動的に正しい型引数を判別してくれたらいいのに、と思う場面がときどきあります。それこそが、暗黙のパラメータが提供するものです。

暗黙のパラメータ ： 基本

Scala 2では、パラメータリストのはじまりで使用できる新しいキーワード implicit があります。

```
ParamClauses ::= {'(' [Param {',' Param}] ')'}
                 ['(' implicit Param {',' Param} ')']
```

このキーワードがある場合、そのリストのすべてのパラメータが暗黙のパラメータとなります。たとえば、次のバージョンの sum では m は暗黙の引数となっています。

```
def sum[A](xs: List[A])(implicit m: Monoid[A]): A =
  if (xs.isEmpty) m.unit
  else m.add(xs.head, sum(xs.tail))
```

この例を見ると分かるように、通常の引数と暗黙のパラメータは組み合わせることができます。ただし暗黙のパラメータリストは、１つのメソッドまたはコンストラクタに対してただ１回だけ指定でき、かつ、それは末尾にこなくてはいけません。

さらに、implicit は、定義や宣言に対する修飾子としても使えます。例をあげます。

```
implicit object stringMonoid extends Monoid[String] {
  def add(x: String, y: String): String = x.concat(y)
  def unit: String = ""
}
implicit object intMonoid extends Monoid[Int] {
  def add(x: Int, y: Int): Int = x + y
  def unit: Int = 0
}
```

暗黙のパラメータの基本的な考え方は、メソッド呼び出しからその引数指定を省略する、というものです。もし、暗黙のパラメータセクションに対応する引数がなければ、Scala コンパイラによって推論されます。

暗黙のパラメータに渡されうる実際の引数は、メソッド呼び出しの時点でアクセス可能で、暗黙の定義やパラメータであると印が付けられた、プレフィックスのつかないすべての識別子 X です。

もし、型が適合して暗黙のパラメータに渡しうる引数が複数ある場合、Scala コンパイラは静的オーバーロード解決の標準的な規則に従い、最も下位の特定的な引数を選びます。たとえば、次の呼び出しにおいて、

```
sum(List(1, 2, 3))
```

stringMonoid と intMonoid が 見えている状況を考えてみます。sum の型パラメータ a は、int にインスタンス化される必要があります。形式上の暗黙のパラメータの型 Monoid[Int] に適合する値は intMonoid のみであり、この object が暗黙のパラメータとして渡されます。

この議論は、暗黙のパラメータの推論はすべての型引数が推論された後に行われる、ということも示しています。

暗黙の型変換

型 S であることが期待される型 T の式 E を考えましょう。T は、S に適合せず、事前定義された変換でも S に変換できないものとします。その場合、Scala コンパイラは、最後の手段として暗黙の型変換 I(E) の適用を試みます。ここで I は、暗黙定義または暗黙のパラメータであることを示す識別子で、変換の時点でプレフィックスなしでアクセスでき、型 T である引数に適用可能で、変換の結果が期待されている型 S に適合するものです。

暗黙の型変換はメンバーの選択にも適用できます。選択 E.x が与えられ、ただしx は型 E のメンバーではない場合、Scala コンパイラは x が I(E) のメンバーとなるよう、暗黙の型変換 I(E).x の挿入を試みます。

整数をクラス scala.Ordered のインスタンスに変換する、暗黙の型変換関数の例をあげます。

```
implicit def int2ordered(x: Int): Ordered[Int] = new Ordered[Int] {
  def compare(y: Int): Int =
    if (x < y1) 1
    else if (x > y1) 1
    else 0
}
```

可視境界

可視境界は、暗黙のパラメータのための便利な糖衣構文です。たとえば、ジェネリックなソートメソッドを考えてみましょう。

```
def sort[A <% Ordered[A]](xs: List[A]): List[A] =
  if (xs.isEmpty || xs.tail.isEmpty) xs
  else {
    val {ys, zs} = xs.splitAt(xs.length / 2)
    merge(ys, zs)
  }
```

可視境界をもつ型パラメータ [a <% Orderd[a]] は、型 a から Orderd[A] への暗黙の型変換が存在する型 a のリストに対して、sort が適用できることを表します。この定義は、次のような暗黙のパラメータを持つメソッドシグネチャの略記として扱われます。

```
def sort[A](xs: List[A])(implicit c: A => Ordered[A]): List[A] = ...
```

(ここで、パラメータ名 c は、プログラム内のほかの名前と衝突しない方法で任意に選ばれます)

もっと詳しい例として、上記の sort に付随している merge メソッドを考えましょう。

```
def merge[A <% Ordered[A]](xs: List[A], ys: List[A]): List[A] =
  if (xs.isEmpty) ys
  else if (ys.isEmpty) xs
  else if (xs.head < ys.head) xs.head :: merge(xs.tail, ys)
  else if ys.head :: merge(xs, ys.tail)
```

可視境界が展開され暗黙の型変換が挿入されると、このメソッドの実装は次のようになります。

```
def merge[A](xs: List[A], ys: List[A])
            (implicit c: A => Ordered[A]): List[A] =
  if (xs.isEmpty) ys
  else if (ys.isEmpty) xs
  else if (c(xs.head) < ys.head) xs.head :: merge(xs.tail, ys)
  else if ys.head :: merge(xs, ys.tail)(c)
```

このメソッド定義の終わりの２行が、暗黙のパラメータ `c` の２種類の使用例になっています。最後から２行目の条件における変換で適用され、最終行では `merge` の再帰呼び出しで暗黙の引数として渡されます。

16 Hindley/Milner 型推論

この章では、Hindely/Milner スタイル[Mil78]の型推論システムの開発を通して、Scala のデータ型とパターンマッチングを見てみます。型推論器の対象言語は、mini-ML と呼ばれる let 構成子を持つ λ計算です。mini-ML の抽象構文木は、次のデータ型 Term によって表現されます。

```
abstract class Term {}
case class Var(x: String) extends Term {
  override def toString = x
}
case class Lam(x: String, e: Term) extends Term {
  override def toString = "(\\" + x + "." + e + ")"
}
case class App(f: Term, e: Term) extends Term {
  override def toString = "(" + f + " " + e + ")"
}
case class Let(x: String, e: Term, f: Term) extends Term {
  override def toString = "let " + x + " = " + e + " in " + f
}
```

木のコンストラクタが４つあります。変数を表す Var、関数抽象を表す Lam、関数適用を表す App、そして let 式を表す Let などです。それぞれのケースクラスでは、Any クラスのメソッド toString をオーバーライドして、項をわかりやすく表示しています。

次に、推論システムによって計算される型を定義します。

```
sealed abstract class Type {}
case class Tyvar(a: String) extends Type {
  override def toString = a
}
case class Arrow(t1: Type, t2: Type) extends Type {
  override def toString = "(" + t1 + ">"  + t2 + ")"
}
case class Tycon(k: String, ts: List[Type]) extends Type {
  override def toString =
    k + (if (ts.isEmpty) "" else ts.mkString("[", ",", "]"))
}
```

型コンストラクタが３つあります。型変数を表す Tyvar、関数の型を表す Arrow、そして Boolean や List などのような型コンストラクタを表す Tycon などです。型コンストラクタは構成要素として型パラメータのリストを持ちます。このリストは Boolean のような型定数については空になります。型コンストラクタもまた、型をわかりやすく表示するために toString メソッドを実装しています。

Type が sealed(封印された)クラスである事に注意してください。これは、Type を拡張するサブクラスやデータコンストラクタを、Type が定義されている定義列の外部では作れない事を意味します。これにより、Type は**クローズ**な代数データ型で、代替物が確実に3つである事が保証されます。逆に、型 Term は**オープン**な代数型であり、さらなる代替物を定義できます。

型推論器の主な部品は object typeInfer に含まれます。新しい型変数を生成するユーティリティ関数から始めましょう。

```
object typeInfer {
  private var n: Int = 0
  def newTyvar(): Type = { n += 1; Tyvar("a" + n) }
```

次に、代入を表すクラスを定義します。代入は、型変数から型への冪等な（何度同じ操作をしても同じ結果となる）関数です。有限な数の型変数を何らかの型にマップし、ほかのすべての型変数はそのまま変えません。代入の意味は、各点でのマッピングから、複数の型から複数の型へのマッピングへと拡張されます。

```
abstract class Subst extends Function1[Type,Type] {

  def lookup(x: Tyvar): Type

  def apply(t: Type): Type = t match {
    case tv @ Tyvar(a) => val u = lookup(tv); if (t == u) t else apply(u)
    case Arrow(t1, t2) => Arrow(apply(t1), apply(t2))
    case Tycon(k, ts) => Tycon(k, ts map apply)
  }

  def extend(x: Tyvar, t: Type) = new Subst {
    def lookup(y: Tyvar): Type = if (x == y) t else Subst.this.lookup(y)
  }
}
val emptySubst = new Subst { def lookup(t: Tyvar): Type = t }
```

代入は、Type => Type 型の関数で表します。クラス Subst を１変数の関数型 Function1[Type, Type](*1) を継承させる事で実現できます。この型のインスタンスであるためには、代入 s は、引数として Type をとり結果として他の Type を戻す apply メソッドを実装する必要があります。これにより、関数適用 s(t) は s.apply(t) と解釈されます。

lookup は Subst クラスの抽象メソッドです。代入には２つの具体的な形態があり、このメソッドの実装方法が異なります。一つは、emptySubst 値によって定義されているもの、もう一つはクラス Subst の extend メソッドによって定義されているものです。

次のデータ型は、型のスキームを記述するもので、型と型の変数名リストからなり、型変数は型スキームにおいて普遍量化されて登場します。たとえば、型スキーム ∀a∀b.a→b は、型チェッカーにおいて次のように表現されます。

```
TypeScheme(List(Tyvar("a"), Tyvar("b")), Arrow(Tyvar("a"), Tyvar("b")))、
```

型スキームのクラス定義には、extends 節がありません。これは、型スキームはクラス AnyRef を直接拡張しているということです。型スキームを構築できる方法がひとつしかなくても、ケースクラス表現を選びました。この型のインスタンスを部分に分解する便利な方法が必要だったからです。

---

*1 このクラスは、関数型を直接のスーパークラスではなくミックスイン として継承しています。これは、Scala の現在の実装では Function1 型が、他のクラスの直接のスーパークラスとして使用できない、 Java の interface であるためです。

```
case class TypeScheme(tyvars: List[Tyvar], tpe: Type) {
  def newInstance: Type = {
    (emptySubst /: tyvars) ((s, tv) => s.extend(tv, newTyvar())) (tpe)
  }
}
```

型スキームオブジェクトには、newInstance メソッドがあります。これは、普遍量化された型変数を新しい変数に変名した後、そのスキームに含まれる型を返します。このメソッドの実装では、拡張オペレーションによって型スキームの型変数を（ /: で）畳み込みます。拡張オペレーションは、与えられた代入 s について、与えられた型変数 tv を新しい型変数に変名することで、s を拡張します。得られた代入は、型スキームのすべての型変数を新しい名前に変名します。次にこの代入は、型スキームの型部分に適用されます。

型推論器に必要な最後の型は、環境を表す Env です。変数名と型スキームを関連づけます。これは、typeInfer モジュール内の型エイリアス Env によって表されます。

```
type Env = List[(String, TypeScheme)]
```

「環境」には２つのオペレーションがあります。 lookup 関数は、与えられた名前に関連づけられた型スキームを返します。名前がその環境に記録されていない場合は null を返します。

```
def lookup(env: Env, x: String): TypeScheme = env match {
  case List() => null
  case (y, t) :: env1 => if (x == y) t else lookup(env1, x)
}
```

gen 関数は与えられた型を、その環境になく、かつ、型で自由であるすべての型変数を量化することで、型スキームに変えます。

```
def gen(env: Env, t: Type): TypeScheme =
  TypeScheme(tyvars(t) diff tyvars(env), t)
```

ある型の自由な型変数の集合は、単に、その型に現れているすべての型変数です。ここでは、次のように構築される型変数のリストとして表現されます。

```
def tyvars(t: Type): List[Tyvar] = t match {
  case tv @ Tyvar(a) =>
    List(tv)
  case Arrow(t1, t2) =>
    tyvars(t1) union tyvars(t2)
  case Tycon(k, ts) =>
    (List[Tyvar]() /: ts) ((tvs, t) => tvs union tyvars(t))
}
```

最初のパターンにある文法 tv @ ... は、それに続くパターンに束縛された、変数を導入します。また、三つ目の節の式にある、明示的な型パラメータ [Tyvar] は、ローカルな型推論を働かせるために必要です。

型スキームの自由な型変数の集合は、量化された型変数を除いた、その型コンポーネントの自由な型変数の集合です。

```
def tyvars(ts: TypeScheme): List[Tyvar] =
  tyvars(ts.tpe) diff ts.tyvars
```

結局、環境の自由な型変数の集合は、その環境に記録されたすべての型スキームの自由な型変数の集合です。

```
  def tyvars(env: Env): List[Tyvar] =
    (List[Tyvar]() /: env) ((tvs, nt) => tvs union tyvars(nt._2))
```

Hindley/Milner型チェックの中心的な操作は単一化であり、それは２つの与えられた型を等しくする代入計算です（そのような代入は**単一化代入**と呼ばれます）。関数 mgu は、事前の代入 s 下の２つの与えられた型 t と u の最も大きな単一化代入を計算します。それは、s を拡張する最も大きな代入 s' を返し、s'(t) と s'(u) を同じ型にします。

```
  def mgu(t: Type, u: Type, s: Subst): Subst = (s(t), s(u)) match {
    case (Tyvar(a), Tyvar(b)) if (a == b) =>
    case (Tyvar(a), _) if !(tyvars(u) contains a) =>
      s.extend(Tyvar(a), u)
    case (_, Tyvar(a)) =>
      mgu(u, t, s)
    case (Arrow(t1, t2), Arrow(u1, u2)) =>
      mgu(t1, u1, mgu(t2, u2, s))
    case (Tycon(k1, ts), Tycon(k2, us)) if (k1 == k2) =>
      (s /: (ts zip us)) ((s, tu) => mgu(tu._1, tu._2, s))
    case _ =>
      throw new TypeError("cannot unify " + s(t) + " with " + s(u))
  }
```

もし単一化代入がなければ mgu 関数は TypeError 例外を送出します。このことは、２つの型が対応する場所で異なる型コンストラクタを持つとき、あるいは型変数が型変数自身からなる型と単一化されるときに起こり得ます。そのような例外を、あらかじめ定義された Exception クラスから継承する、ケースクラスのインスタンスとしてモデル化してみます。

```
  case class TypeError(s: String) extends Exception(s) {}
```

型チェックの主な仕事は関数 tp によって実装されます。この関数はパラメータとして環境 env、項 e、プロトタイプ t、事前の代入 s をとります。関数 tp は s を拡張して代入 s' をもたらし、s'(env) ├ e:s'(t) を Hindler/Milner 型システム[Mil78]の推論規則に従って推論可能な型判定に変えます。もしそのような代入が存在しなければ TypeError 例外が送出されます。

```
  def tp(env: Env, e: Term, t: Type, s: Subst): Subst = {
    current = e
    e match {
      case Var(x) =>
        val u = lookup(env, x)
        if (u == null) throw new TypeError("undefined: " + x)
        else mgu(u.newInstance, t, s)

      case Lam(x, e1) =>
        val a, b = newTyvar()
        val s1 = mgu(t, Arrow(a, b), s)
        val env1 = {x, TypeScheme(List(), a)} :: env
        tp(env1, e1, b, s1)

      case App(e1, e2) =>
        val a = newTyvar()
        val s1 = tp(env, e1, Arrow(a, t), s)
```

```
        tp(env, e2, a, s1)

      case Let(x, e1, e2) =>
        val a = newTyvar()
        val s1 = tp(env, e1, a, s)
        tp({x, gen(env, s1(a))} :: env, e2, t, s1)
    }
  }
  var current: Term = null
```

エラー診断を支援するため、tp 関数は直近に解析されたサブ項を現変数に保存します。したがって、もし型チェックが TypeError 例外でアボートされれば、この変数が問題を引き起こしたサブ項を含みます。

型推論モジュールの最後の関数 typeOf は、tp の簡略化版です。与えられた環境 env 内の与えられた項 e の型を計算します。新しい型変数 a を生成し、型付け代入を計算して env ├ e:a を推論可能な型判定にし、代入を a に適用した結果を返します。

```
    def typeOf(env: Env, e: Term): Type = {
      val a = newTyvar()
      tp(env, e, a, emptySubst)(a)
    }
  }// end typeInfer
```

型推論を適用するにあたり、よく使われる定数まわりの環境を事前に定義しておくと便利です。事前定義されたモジュールでは、boolean、数字、リスト等の型について、それらのプリミティブな操作とともに、関係するものを含む環境 env を定義しています。 不動点操作 fix も定義しており、再帰を表現するのに使えます。

```
  object predefined {
    val booleanType = Tycon("Boolean", List())
    val intType = Tycon("Int", List())
    def listType(t: Type) = Tycon("List", List(t))

    private def gen(t: Type): typeInfer.TypeScheme = typeInfer.gen(List(), t)
    private val a = typeInfer.newTyvar()
    val env = List(
      {"true", gen(booleanType)},
      {"false", gen(booleanType)},
      {"if", gen(Arrow(booleanType, Arrow(a, Arrow(a, a))))},
      {"zero", gen(intType)},
      {"succ", gen(Arrow(intType, intType))},
      {"nil", gen(listType(a))},
      {"cons", gen(Arrow(a, Arrow(listType(a), listType(a))))},
      {"isEmpty", gen(Arrow(listType(a), booleanType))},
      {"head", gen(Arrow(listType(a), a))},
      {"tail", gen(Arrow(listType(a), listType(a)))},
      {"fix", gen(Arrow(Arrow(a, a), a))}
    )
  }
```

型推論の使い方の例を見てみましょう。あらかじめ定義された環境 Predefined.env 内で計算される、与えられた項の型を返す関数 showType を定義しましょう。

```
object testInfer {
  def showType(e: Term): String =
    try {
      typeInfer.typeOf(predefined.env, e).toString
    } catch {
      case typeInfer.TypeError(msg) =>
        "\n cannot type: " + typeInfer.current +
        "\n reason: " + msg
    }
```

アプリケーションを起動すると

```
> testInfer.showType(Lam("x", App(App(Var("cons"), Var("x")), Var("nil"))))
```

次のように応答するでしょう。

```
> (a6->List[a6])
```

演習 16.0.1 Mini-ML 型推論器を、letrec 構成子で再帰関数を許容するように拡張しなさい。構文 :

```
letrec ident "=" term in term .
```

letrec の形は、定義された識別子が定義している式中に見えていることを除き、let と同じです。letrec を使えば、リスト用の length 関数を次のように定義できます。

```
letrec length = \xs.
  if (isEmpty xs)
    zero
    (succ (length (tail xs)))
in ...
```

17 並列処理の抽象

この章では一般的な並列プログラミングパターンを調べ、それらが Scala においてどのように実装されるかを示します。

17.1 シグナルとモニター

Example 17.1.1 モニターは Scala におけるプロセスの相互排他処理の基本的な手段を提供します。 AnyRef クラスの各インスタンスは、次の１つあるいは複数のメソッドを呼ぶことでモニターとして使えます。

```
def   synchronized[A] (e: => A): A
def   wait()
def   wait(msec: Long)
def   notify()
def   notifyAll()
```

同期メソッドは相互排他的に、つまり同時にはただ１つのスレッドだけが与えられたモニターの同期化引数を実行できる方式で、その引数である計算 e を実行します。

スレッドはシグナルをウェイトすることで、モニター内でサスペンドできます。 wait メソッドを呼び出すスレッドは、同じオブジェクトの notify メソッドが他のスレッドからその後呼ばれるまで、ウェイトします。

ウェイトの制限時間つき形態もあり、シグナルを受信するか、あるいは指定された時間(ミリ秒で与えられる)が過ぎるまでブロックします。さらにまた、シグナルを待つ全てのスレッドをアンブロックする notifyAll メソッドもあります。これらのメソッドは Monitor クラスと同様に、Scala ではプリミティブです。つまり、それらは実行時システムによって実装されています。

典型的には、スレッドはある条件が確立するまでウェイトします。もしその条件が wait をコールするまでに確立していなければ、そのスレッドは、他のスレッドがその条件を確立するまで、ブロックします。notify あるいは notifyAll を発行してウェイトしているプロセスを起動するのは、他のスレッドの責任です。しかし、ウェイトしているプロセスが notify 発行後すぐに実行されるという保証はありません。他のプロセスが最初に実行され、その条件を無効にすることもあり得ます。ですから、条件 C の確立をウェイトする正しい形は、while ループを使うことです。

```
while (!C ) wait()
```

モニターの使用例として、境界付きバッファの実装をあげておきます。

```
class BoundedBuffer[A](N: Int) {
  var in = 0, out = 0, n = 0
  val elems = new Array[A](N)

  def put(x: A) = synchronized {
    while (n >= N) wait()
    elems(in) = x ; in = (in + 1) % N ; n = n + 1
```

```
    if (n == 1) notifyAll()
  }

  def get: A = synchronized {
    while (n == 0) wait()
    val x = elems(out) ; out = (out + 1) % N ; n = n - 1
    if (n == N - 1) notifyAll()
    x
  }
}
```

境界付きバッファを使って生産プロセスと消費プロセスとの間でコミュニケーションをとるプログラムを見てみましょう。

```
import scala.concurrent.ops._
...
val buf = new BoundedBuffer[String](10)
spawn { while (true) { val s = produceString ; buf.put(s) } }
spawn { while (true) { val s = buf.get ; consumeString(s) } }
}
```

spawn メソッドは、パラメータで与えられた式を実行する新しいスレッドを生成します。それはオブジェクト scala.concurrent.ops において、次のように定義されています。

```
def spawn(p: => Unit) {
  val t = new Thread() { override def run() = p }
  t.start()
}
```

17.2 同期変数

同期化された変数(あるいは縮めて、同期変数)は変数の読み出しと設定用に、get と put 操作を提供します。get 操作は、変数が定義されるまでブロックします。unset 操作は、変数を未定義状態にリセットします。

次は同期変数の標準的な実装です。

```
package scala.concurrent
class SyncVar[A] {
  private var isDefined: Boolean = false
  private var value: A = _
  def get = synchronized {
    while (!isDefined) wait()
    value
  }
  def set(x: A) = synchronized {
    value = x; isDefined = true; notifyAll()
  }
  def isSet: Boolean = synchronized {
    isDefined
  }
```

```
  def unset = synchronized {
    isDefined = false
  }
}
```

17.3 フューチャー

フューチャーは、他のクライアントスレッドにおいて並列に計算される値であり、クライアントスレッドによっていつか使用されます。フューチャーは並行プロセスのリソースを有効に利用するために使います。典型的な使い方は次のようです。

```
import scala.concurrent.ops._
...
val x = future(someLengthyComputation)
anotherLengthyComputation
val y = f(x()) + g(x())
```

future メソッドはオブジェクト scala.concurrent.ops において、次のように定義されています。

```
def future[A](p: => A): Unit => A = {
  val result = new SyncVar[A]
  fork { result.set(p) }
  (() => result.get)
}
```

futureメソッドは、実行すべき計算 p をパラメータにとります。計算の型は任意であり、そのことは future の型パラメータ a によって表現されています。future メソッドでは計算結果を表すパラメータを取る、ガード result を定義します。つぎに、新しいスレッドを fork して、答えを計算し終了時に result ガードを起動します。このスレッドに並行して、関数（訳注 : future）は型 A の無名関数を返します。この無名関数は、呼ばれたときに result ガードが起動されるまでウェイトし、そして、一度 result ガードが起動されると結果の引数を返します。同じ時に、関数は同じ引数で result ガードを再起動しますが、関数の future 起動は結果をすぐに返せます。

17.4 並列計算

次の例では、関数 par は計算のペアをパラメータとしてとり、計算結果を別のペアで返します。２つの計算は並列実行されます。

この関数はオブジェクト scala.concurrent.ops において、次のように定義されています。

```
def par[A, B](xp: => A, yp: => B): (A, B) = {
  val y = new SyncVar[B]
  spawn { y set yp }
  (xp, y.get)
}
```

同じ場所で replicate 関数が定義されていて、多数の計算の複製を並列実行しま す。各複製インスタンスには、それを識別する整数値が渡されます。

```
def replicate(start: Int, end: Int)(p: Int => Unit) {
  if (start == end)
    ()
  else if (start + 1 == end)
    p(start)
  else {
    val mid = (start + end) / 2
    spawn { replicate(start, mid)(p) }
    replicate(mid, end)(p)
  }
}
```

次の関数は、配列のすべての要素について並列計算を実行するために、replicate を使っています。

```
def parMap[A,B](f: A => B, xs: Array[A]): Array[B] = {
  val results = new Array[B](xs.length)
  replicate(0, xs.length) { i => results(i) = f(xs(i)) }
  results
}
```

17.5 セマフォ

プロセス同期の一般的なメカニズムは**ロック**（あるいは**セマフォ**）です。ロックは２つのアトミックなアクションを提供します。すなわち、**獲得**と**解放**です。次は Scala におけるロックの実装です。

```
package scala.concurrent

class Lock {
  var available = true
  def acquire = synchronized {
    while (!available) wait()
    available = false
  }
  def release = synchronized {
    available = true
    notify()
  }
}
```

17.6 リーダー/ライター

より複雑な同期の形態では、共通リソースを変更することなくアクセスする**リーダー**と、アクセスと変更の両方が可能な**ライター**を区別します。リーダーとライターを同期させるために startRead、startWrite、endRead、endWrite、その他を実装する必要があります。

- 多数の並行リーダーがある
- 同時にはただ１つのライターのみ可能
- ペンディング中の書き込み要求は、ペンディング中の読み込み要求 よりも優先するが、実行中のリード操作をプリエンプトしない

次のリーダー/ライター ロックの実装は**メールボックス**に基づいています（17.10節参照）。

```
import scala.concurrent._

class ReadersWriters {
  val m = new MailBox
  private case class Writers(n: Int), Readers(n: Int) { m send this }
  Writers(0); Readers(0)
  def startRead = m receive {
    case Writers(n) if n == 0 => m receive {
      case Readers(n) => Writers(0); Readers(n+1)
    }
  }
  def startWrite = m receive {
    case Writers(n) =>
      Writers(n+1)
      m receive { case Readers(n) if n == 0 => }
  }
  def endRead = m receive {
    case Readers(n) => Readers(n-1)
  }
  def endWrite = m receive {
    case Writers(n) => Writers(n-1); if (n == 0) Readers(0)
  }
}
```

17.7 非同期チャネル

プロセス内通信の基本的な方法には非同期チャネルがあります。その実装では次の単純な連結リストを利用しています。

```
class LinkedList[A] {
  var elem: A = _
  var next: LinkedList[A] = null
}
```

連結リストへの要素の挿入・追加を助けるために、連結リストの中への参照はすべて、リストの先頭を概念的に表すノードの手前のノードを指します。空きの連結リストはダミーノー

ドで始まり、次の要素は nullです。

チャネルクラスは、送られたがまだ読み出されていないデータを保持するのに連結リストを使います。反対の端では、空きのチャネルから読み出そうとするスレッドは nreaders フィールドをインクリメントすることで自分を登録し、通知されるのを待ちます。

```
package scala.concurrent

class Channel[A] {
  class LinkedList[A] {
    var elem: A = _
    var next: LinkedList[A] = null
  }
  private var written = new LinkedList[A]
  private var lastWritten = written
  private var nreaders = 0

    def write(x: A) = synchronized {
      lastWritten.elem = x
      lastWritten.next = new LinkedList[A]
      lastWritten = lastWritten.next
      if (nreaders > 0) notify()
    }

    def read: A = synchronized {
      if (written.next == null) {
        nreaders = nreaders + 1; wait(); nreaders = nreaders - 1
      }
      val x = written.elem
      written = written.next
      x
    }
}
```

17.8 同期チャネル

同期チャネルの実装例を見てみましょう。ここではメッセージの送信者はメッセージが受け取られるまでブロックします。同期チャネルは、転送するメッセージを保持するのにただ1つの変数を必要としますが、リーダーとライタープロセスをうまく連動させるために3つの信号を使います。

```
package scala.concurrent

class SyncChannel[A] {
  private var data: A = _
  private var reading = false
  private var writing = false

    def write(x: A) = synchronized {
      while (writing) wait()
      data = x
      writing = true
```

```
        if (reading) notifyAll()
        else while (!reading) wait()
    }

    def read: A = synchronized {
      while (reading) wait()
      reading = true
      while (!writing) wait()
      val x = data
      writing = false
      reading = false
      notifyAll()
      x
    }
}
```

17.9 ワーカー

Scala における**計算サーバー**の実装例を見てみましょう。サーバーは future メソッドを実装し、与えられた式をその呼び出し者と並列に評価します。 17.3節の実装とは異なり、サーバーはスレッドの事前定義された数字だけを持って future を計算します。サーバーの一つのあり得る実装では、別々のプロセッサ上で各スレッドを走らせることができ、シングルプロセッサ上で複数のスレッドを切り替える際のコンテキストスイッチングに伴うオーバーヘッドを回避できます。

```
import scala.concurrent._, scala.concurrent.ops._

class ComputeServer(n: Int) {

  private abstract class Job {
    type T
    def task: T
    def ret(x: T)
  }

  private val openJobs = new Channel[Job]()

  private def processor(i: Int) {
    while (true) {
      val job = openJobs.read
      job.ret(job.task)
    }
  }

  def future[A](p: => A): () => A = {
    val reply = new SyncVar[A]()
    openJobs.write{
      new Job {
        type T = A
        def task = p
        def ret(x: A) = reply.set(x)
```

```
        }
      }
      () => reply.get
    }

    spawn(replicate(0, n) { processor })
  }
```

計算すべき式(つまり、future を呼び出すときの引数)は openJobs チャネルに書かれます。**ジョブ**(job) は次を備えた オブジェクトです。

- job の計算結果を表す抽象型 T
- 計算すべき式を表す、パラメータのない、型 T の taskメソッド
- 前に一度計算した結果をとる ret メソッド

計算サーバーはその初期化中に n 個の processor プロセスを生成します。各プロセスは未処理の jobを繰り返し取り出し、job の task メソッドを評価して job の ret メソッドに結果を渡します。多相的な future メソッドは、reply という名前のガードを使って ret メソッドを実装している新しい job を生成し、その job を未処理 job の集合に入れます。そして、対応する reply ガードが呼ばれるまでウェイトします。

この例は抽象型の使い方を示しています。抽象型 T は、job の結果型を記録し、異なる job に対しては異なる型をとることもできます。抽象型がなければ、ユーザは型キャストと動的な型テストへの信頼なしに、静的に型安全な方法で同じクラスを作ることはできないでしょう。次に、式 41 + 1 を評価する計算サーバーを使うコードを示します。

```
object Test with Executable {
  val server = new ComputeServer(1)
  val f = server.future(41 + 1)
  println(f())
}
```

17.10 メールボックス

メールボックスはプロセス同期と通信のための、高度で柔軟な構造物です。メッセージの送受信が可能です。ここで**メッセージ**とは任意のオブジェクトを指します。シグナルのタイムアウトに使う TIMEOUT という特別のメッセージがあります。

```
case object TIMEOUT
```

メールボックスは次のシグネチャを実装します。

```
class   MailBox {
  def   send(msg: Any)
  def   receive[A](f: PartialFunction[Any, A]): A
  def   receiveWithin[A](msec: Long)(f: PartialFunction[Any, A]): A
}
```

メールボックスの状態はメッセージのマルチセットから成ります。メッセージは send メソッドで メールボックスへ追加されます。メッセージは receive メソッドで メールボック

スから取り除かれ、メッセージプロセッサ f の引数に渡されます。 f はメッセージから何らかの結果型への部分関数です。通常、この関数はパターンマッチ式で実装されます。 receive メソッドは、そのメッセージプロセッサが定義されたメールボックスにメッセージが届くまで、ブロックされます。マッチしたメッセージはメールボックスから取り除かれ、ブロックされていたスレッドは再スタートしてメッセージプロセッサをそのメッセージに適用します。送られたメッセージとレシーバの双方とも時間順に並べられます。レシーバ r は、マッチしたメッセージ m へ適用されますが、それは、各コンポーネントを時間順に並べた個別の順序中で、m, r に先立つ {メッセージ,レシーバ}ペアが他にないときに限ります。

メールボックスの使い方の簡単な例として one-place バッファを考えてみましょう。

```
class OnePlaceBuffer {
  private val m = new MailBox            // An internal mailbox
  private case class Empty, Full(x: Int) // Types of messages we deal with
  m send Empty                           // Initialization
  def write(x: Int)
    { m receive { case Empty => m send Full(x) } }
  def read: Int =
    m receive { case Full(x) => m send Empty; x }
}
```

メールボックスクラスは、次のようにも実装できます。

```
class MailBox {
  private abstract class Receiver extends Signal {
    def isDefined(msg: Any): Boolean
    var msg = null
  }
```

テストメソッド isDefined を備えたレシーバ用の内部クラスを定義し、与えられたメッセージに対してレシーバが定義されているかどうかを示すようにします。レシーバは Signal クラスから、レシーバスレッドを起動するのに使われる nofify メソッドを継承します。レシーバスレッドが起動されると、適用すべきメッセージは Reciever の msg 変数に保存されます。

```
private   val   sent = new LinkedList[Any]
private   var   lastSent = sent
private   val   receivers = new LinkedList[Receiver]
private   var   lastReceiver = receivers
```

メールボックスクラスは２つの連結リストを保持していて、一つは、送信されたけれど取り出されていないメッセージ用で、もう一つは、ウェイトしているレシーバ用のものです。

```
def send(msg: Any) = synchronized {
  var r = receivers, r1 = r.next
  while (r1 != null && !r1.elem.isDefined(msg)) {
    r = r1; r1 = r1.next
  }
  if (r1 != null) {
    r.next = r1.next; r1.elem.msg = msg; r1.elem.notify
  } else {
    lastSent = insert(lastSent, msg)
  }
}
```

send メソッドは最初に、ウェイトしているレシーバがその送信されたメッセージに適用可能かどうかチェックします。もしそうなら、レシーバに通知されます。そうでなければ、メッセージは送信されたメッセージの連結リストに追加されます。

```
  def receive[A](f: PartialFunction[Any, A]): A = {
    val msg: Any = synchronized {
      var s = sent, s1 = s.next
      while (s1 != null && !f.isDefinedAt(s1.elem)) {
        s = s1; s1 = s1.next
      }
      if (s1 != null) {
        s.next = s1.next; s1.elem
      } else {
        val r = insert(lastReceiver, new Receiver {
          def isDefined(msg: Any) = f.isDefinedAt(msg)
        })
        lastReceiver = r
        r.elem.wait()
        r.elem.msg
      }
    }
    f(msg)
  }
```

recieve メソッドは最初に、メッセージプロセッサ関数 f が、既に送信されたけれどもまだ取り出されていないメッセージに適用可能かどうかチェックします。もしそうなら、スレッドは続けてすぐに f をそのメッセージに適用します。そうでなければ、新しいレシーバが作られてレシーバリストへリンクされ、スレッドはそのレシーバ上の通知を待ちます。スレッドは再び起動されると、f をそのレシーバに保存されたメッセージに適用します。 連結リストの insert メソッドは次のように定義されています。

```
  def insert(l: LinkedList[A], x: A): LinkedList[A] = {
    l.next = new LinkedList[A]
    l.next.elem = x
    l.next.next = l.next
    l
  }
```

メールボックスクラスは、指定された最大時間だけブロックする receiveWithin メソッドも提供しています。メッセージを指定された時間(ミリ秒で与えられる)以内に受信しなければ、メッセージプロセッサ引数 f は TIMEOUT という特別のメッセージでアンブロックされます。 recieveWithin の実装は receive とほとんど同じです。

```
    def receiveWithin[A](msec: Long)(f: PartialFunction[Any, A]): A = {
      val msg: Any = synchronized {
        var s = sent, s1 = s.next
        while (s1 != null && !f.isDefinedAt(s1.elem)) {
          s = s1; s1 = s1.next
        }
        if (s1 != null) {
          s.next = s1.next; s1.elem
        } else {
          val r = insert(lastReceiver, new Receiver {
            def isDefined(msg: Any) = f.isDefinedAt(msg)
          })
```

```
            lastReceiver = r
            r.elem.wait(msec)
            if (r.elem.msg == null) r.elem.msg = TIMEOUT
            r.elem.msg
          }
        }
        f(msg)
      }
    } // end MailBox
```

違いは、制限時間つきの wait コールと、その後の文だけです。

17.11 アクター

第 3 章で電子オークションサービスのプログラム実装例をざっと見ました。そのサービスは、パターンマッチングを使ってメールボックス中のメッセージを調べて動く、高度なアクタープロセスに基づいています。アクターの洗練、最適化された実装は scala.actors パッケージに収められています。ここで、アクターライブラリの簡略化バージョンを見てみましょう。

次のコードは scala.actors パッケージの実装とは異なります。アクターの簡略バージョンをどのように実装できるか、の例として見てください。アクターが実際にどのように定義されているか、あるいは標準 Scala ライブラリにおける実装を記述するものではありません。後者については Scala APIドキュメントを参照してください。

簡略化されたアクターは、通信プリミティブがメールボックスのそれであるようなスレッドです。そのようなアクターは、MailBox クラスを備えた Java の標準 Threadクラスのミックスイン合成拡張として定義できます。 私たちは Thread クラスの run メソッドもオーバライドして、その act メソッドで定義されたアクター動作を実行させます。! メソッドは MailBox の send メソッドを呼び出すだけです。

```
abstract class Actor extends Thread with MailBox {
  def act(): Unit
  override def run(): Unit = act()
  def !(msg: Any) = send(msg)
}
```

参考文献

ASS96 Harold Abelson, Gerald Jay Sussman, and Julie Sussman. The Structure and Interpretation of Computer Programs, 2nd edition. MIT Press, Cambridge,Massachusetts, 1996.

Mil78 Robin Milner. A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences, 17:348-375, Dec 1978. Journal of Computer and System Sciences, 17:348-375, Dec 1978.