新しいプログラミング言語の学び方 ~パーザコンビネータライブラリで学ぶScala~

Scala福岡2019

2019/01/19（土）

水島宏太

自己紹介

- Japan Scala Association代表理事
- 株式会社ドワンゴ所属
  - Scala研修テキスト 主著者
- プログラミング言語好き
  - **Scala**, Nemerle, Racket, etc.
- プログラミング言語作ってます
  - Onion (Suspended)
  - **Klassic**
- プログラミング言語教育に興味があります
  - 『実践Scala入門』共著者

今日のテーマ

新しいプログラミング言語の学び方

（ただし、第二言語以降）

注意

- こういう学び方が絶対に正しいという主張ではありません

よく見る意見（1）

一つの言語を学んでいれば他の言語はすぐわかる

- **具象**構文が異なったりといった**些細**な点でつまづくことも

よく見る意見（2）

- 未知のパラダイムの言語を理解するのは簡単ではない

割と正しい気がするが、（多くの）プログラミング言語全体に共通する特性を理解 できればより楽な気も...

私の意見

- （多くの）プログラミング言語に共通する構造を理解するのが重要
  - 具象構文
  - 抽象構文
  - 型システム
  - 意味論（実行モデル）
- 難解な型システムや意味論を持った言語でなければ大体適用できる
- 言語のおおざっぱな全体像を非常に高速に理解できる
  - …可能性が高い

具象構文

- カジュアルに文法や構文と言うときこれを指すことが多い
- 例：
  - if式の文法は、"if"の後に"("が来て～
  - クラス定義の文法は、"class'の後に～
- 空白文字や、キーワード名など、後で不要になる情報も含む

具象構文とBNF

- 具象構文はBNF（Backus-Naur Form）で定義されることが多い
- 例：

```
<expression> ::= "if" "(" <expression> ")" ...
       | "while" "(" <expression> ")" ...
       | ...;
```

具象構文に関する注意

- 具象構文はあくまで見た目（UI）を定義しているだけ
- 本質的な構造より余分な情報を含んでいる
  - 空白文字
  - セミコロン
  - カンマ
  - , etc.

抽象構文

- 具象構文から、プログラムの解釈に不要な情報を削除したもの
- 例：
  - if式は3つの式からなる
    - 条件式、then式、else式
  - while式は2つの式からなる
    - 条件式、本体式

```
Expression = IF (Expression, Expression, Expression)
           | WHILE (Expression, Expression)
           | ....
```

型システム

- 型がある言語における、型同士の互換性（など）に関する規則集
  - ちょっと語弊あり
- 型がない言語では意味がない
  - 「いわゆる」動的型付き言語
- おおざっぱには次の要素を持つかが重要
  - 派生型（サブタイピング）
    - 構造的部分型と名前的部分型
  - 多相型（ジェネリクス）

派生型

```
class Foo { def foo: String = "Foo" }
class Bar extends Foo
class Hoge { def foo: String = "Foo" }

type FooLike = { def foo: String }
// Bar型の値をFoo型の変数に代入可能
val foo1: Foo = new Bar
// Foo型はfoo:Stringを持っている
val foo2: FooLike = new Foo
// Hoge型はfoo:Stringを持っている
val foo3: FooLike = new Hoge
```

多相型

```
class Cell[A](var value: A)
val c1: Cell[String] = new Cell("Foo")
val c2: Cell[Int] = new Cell(1)

def id[A](v: A): A = v
val i1: String = id("Foo")
val i2: Int = id(1)
```

意味論

- どういう風に実行されるかを定義したもの
- 例（カジュアルな言い方）：
  - 1 + 1 は 1 + 1を計算した値になる
  - return e は：
    - 現在のメソッドの実行を中断
    - e を評価した結果を呼び出し元に返す

私の意見（再）

- プログラミング言語を
  - 構文（具象構文、抽象構文）
  - 型システム
  - 意味論

  に分解して理解することで、素早く全体像をつかめる

最初に書くプログラム

- "Hello, World!" は定番
  - それだけではほとんど意味がない
- 他には？
  - Webアプリケーション？
  - コマンドラインツール？
  - テストフレームワーク？

  アプリケーション固有の部分が多い

私が最初に書くプログラム

パーザコンビネータ

動機

- 言語の**全体像**を素早く把握したい
  - 構文、型システム、意味論
  - 色々な言語機能を使う例が必要
- 言語の抽象化機構を把握したい
  - 関数
  - 高階関数
  - モジュールシステム
  - 型システム（型があれば）
  - ,etc.

パーザコンビネータ？

- パーザ（構文解析器）を書くためのEDSL
- 関数≒パーザ
- パーザとパーザを合成して大きなパーザを作る
- **BNF**っぽく書ける

パーザとBNF

- パーザを書く前にBNFを設計する（ことが多い）

算術式 in 疑似BNF

```
Expression ::= Additive;
Additive ::= Multitive {"+" Multitive | "-" Multitive};
Multitive ::= Primary {"+" Primary | "-" Primary};
Primary ::= "(" Expression ")" | Number;
Nunber ::= "0" | [1-9]{[0-9]};
```

算術式 in パーザコンビネータ

```
def R: Parser[Int] = E
def E: P[Int] = rule(A)
def A: Parser[Int] = rule(chainl(M) {
  $("+").map { op => (lhs: Int, rhs: Int) => lhs + rhs } |
  $("-").map { op => (lhs: Int, rhs: Int) => lhs - rhs }
})
def M: Parser[Int] = rule(chainl(P) {
  $("*").map { op => (lhs: Int, rhs: Int) => lhs * rhs } |
  $("/").map { op => (lhs: Int, rhs: Int) => lhs / rhs }
})
def P: P[Int] = rule{
  (for {
    _ <- $("("); e <- E; _ <- $(")")} yield e) | number
}
def number: P[Int] = rule {
  ('0' to '9').map{c => $(c.toString) ^^ (_.toInt)}.reduce((p1, p2) =
```

パーザコンビネータのデータ構造を考える

- ~: 実質タプル。中置パターンマッチのために定義

```
case class ~[+A, +B](a: A, b: B)

new ~(new ~("A", "B"), "C") match {
  case a ~ b ~ c => println(a + b + c) // ABC
}
```

パーザコンビネータのデータ構造を考える

結果を格納するデータ型を代数的データ型で定義

- 成功（Success）：A型のvalueと残りの文字列next
- 失敗（Failure)：残りの文字列next

```
sealed trait Result[+A] { def next: String }
case class Success[+A](value: A, next: String) extends Result[A]
case class Failure(next: String) extends Result[Nothing]
```

パーザコンビネータのデータ構造を考える

その他

```
// パーザはStringからResultへの関数
type Parser[A] = String => Result[A]
// 短く書くため
type P[A] = Parser[A]
```

パーザコンビネータのデータ構造を考える〜まとめ

```
case class ~[+A, +B](a: A, b: B)
sealed trait Result[+A] { def next: String }
case class Success[+A](value: A, next: String) extends Result[A]
case class Failure(next: String) extends Result[Nothing]
type Parser[A] = String => Result[A]
type P[A] = Parser[A]
```

学べること

- 代数的データ型の定義方法
  - +クラス継承の方法
- ジェネリックなクラスの定義方法
  - 共変クラスの定義方法
- 型の別名付けの方法
- 関数の型の表記方法

基本コンビネータの設計

- 文字列リテラルの処理

```
final def $(literal: String): Parser[String] = {input =>
  if(literal.length > 0 && input.length == 0) {
    Failure("")
  } else if(input.startsWith(literal)) {
    Success(literal, input.substring(literal.length))
  } else {
    Failure(input)
  }
}
```

学べること

- 文字列の簡単な扱い
  - lenght, substring, startsWith
- 条件分岐
  - if
- 無名関数の表記法
  - {a, b, ... => ...}

派生コンビネータの設計

- Parser にメソッドを生やす

```
implicit class RichParser[A](val self: Parser[A]) {
  def ~ ...
  def | ...
  def ? ...
  def * ...
}
```

学べること

- 既存のクラスにメソッドを追加する方法
  - implicit class

派生コンビネータの設計

- 連接コンビネータ

```
def ~[B](right: Parser[B]) : Parser[A ~ B] = {input =>
  self(input) match {
    case Success(value1, next1) =>
      right(next1) match {
        case Success(value2, next2) =>
          Success(new ~(value1, value2), next2)
        case failure@Failure(_) =>
          failure
      }
    case failure@Failure(_) =>
      failure
  }
}
```

学べること

- match式（パターンマッチ）の使い方
- 中置型コンストラクタの表記方法（A ~ B）
- 多相メソッド（~[B]の定義方法）
- f.apply(x) から f(x) への書き換え

派生コンビネータの設計

- 選択コンビネータ

```
def |[B >: A](rhs: Parser[B]): Parser[B] = {input =>
  self(input) match {
    case success@Success(_, _) => success
    case Failure(_) => rhs(input)
  }
}
```

学べること

- 下限境界（B >: A）

派生コンビネータの設計

- 繰り返しコンビネータ

```
def * : Parser[List[A]] = {input: String =>
  def repeat(input: String): Result[List[A]] = self(input) match {
    case Success(value, next1) =>
      repeat(next1) match {
        case Success(result, next2) => Success(value::result, next2)
        case r => sys.error("cannot reach here")
      }
    case Failure(next) => Success(Nil, next)
  }
  repeat(input) match {
    case r@Success(_, _) => r
    case r@Failure(_) => r
  }
}
```

学べること

- 再帰メソッドの定義方法
- メソッド内メソッドの定義方法
- Listの扱い方の基本
  - Nil, ::

派生コンビネータの設計

- 繰り返しコンビネータ（0回または1回）

```
def ? : Parser[Option[A]] = {input =>
  self(input) match {
    case Success(v, next) => Success(Some(v), next)
    case Failure(next) => Success(None, next)
  }
}
```

学べること

- Optionの扱い

派生コンビネータの設計

- 繰り返しコンビネータ（2項演算子用）

```
def chainl[T](p: Parser[T])(q: Parser[(T, T) => T]): Parser[T] = {
  (p ~ (q ~ p).*).map { case x ~ xs =>
    xs.foldLeft(x) { case (a, f ~ b) =>
      f(a, b)
    }
  }
}
```

学べること

- 高階関数 foldLeftの使い方
- 中置パターンマッチ
- タプルのパターンマッチ

派生コンビネータの設計

- 射影コンビネータ
  - for式で使うためだけに定義

```
def map[B](function: A => B): Parser[B] = {input =>
  self(input) match {
    case Success(value, next) => Success(function(value), next)
    case failure@Failure(_) => failure
  }
}
def ^^[B](function: A => B): Parser[B] = map(function)
def flatMap[B](function: A => Parser[B]): Parser[B] = {input =>
  self(input) match {
    case Success(value, next) =>
      function(value)(next)
    case failure@Failure(_) =>
      failure
  }
}
```

規則のためのメソッド

- 単に遅延評価のためだけ

```
final def rule[A](body: => Parser[A]): Parser[A] = {input =>
  body(input)
}
```

学べること

- 名前呼び出し（by-name parameter）の使い方
  - => Parser[A]

全体像（再）

```
def E: P[Int] = rule(A)
def A: Parser[Int] = rule(chainl(M) {
  $("+").map { op => (lhs: Int, rhs: Int) => lhs + rhs } |
  $("-").map { op => (lhs: Int, rhs: Int) => lhs - rhs }
})
def M: Parser[Int] = rule(chainl(P) {
  $("*").map { op => (lhs: Int, rhs: Int) => lhs * rhs } |
  $("/").map { op => (lhs: Int, rhs: Int) => lhs / rhs }
})
def P: P[Int] = rule{
  (for {
    _ <- $("("); e <- E; _ <- $(")")} yield e) | number
}
def number: P[Int] = rule {
  ('0' to '9').map{c => $(c.toString) ^^ (_.toInt)}.reduce((p1, p2) =
}
```

学べること

- for式と map, flatMap の関係
- 相互再帰メソッドの定義方法

テストケース（ScalaTest）

```
val parser = E

var input = ""
input = "1+2*3"
assert(parser(input) == Success(7, ""))
input = "1+5*3/4"
assert(parser(input) == Success(4, ""))
input = "(1+5)*3/2"
assert(parser(input) == Success(9, ""))

input = "1+ "
assert(parser(input) == Success(1, " "))

input = "(1-5) *3/2"
assert(parser(input) == Success(-4, " *3/2"))
```

「型にはめる」ことの重要性

- このパーザコンビネータは、多くの言語で定型的に書ける
  - 細部の違いはある
- 「パーザコンビネータの型」に言語の機能を当てはめれば完成
- 作成過程で、言語ごとの細かい違いをある程度学べる
  - 評価戦略、無名関数、多相型、再帰、分岐、...

開発にかかる時間

- 新しい言語でも、最大で5時間程度（自分の場合）
- 5時間でも結構いろいろなことを学べる

自分なりの「Hello, World」を持とう

- "Hello, World"の次に作るプログラムを決めておく
- 本質に関係の無い要素が入らないサンプルが良い
- 自分の「Hello, World」で学べることを意識する
  - 言語のある部分についてはかなり高速に学べる

まとめ

プログラミング言語の学習（第二言語以降）では：

- 多くのプログラミング言語に共通の性質に着目する
  - 構文、型システム、意味論
- 良い課題プログラムを持っておく

と良い。

- ただし、このやり方がうまくいかない言語もある