どう解く？ in Scala

株式会社ドワンゴ　水島宏太

おさらい（BNF）

```
expression ::= additive

additive ::= multitive ('+' multitive | '-' multitive)*

multitive ::= primary ('*' primary | '/' primary)*

primary ::= '(' expression ')' | number

number ::= '0'|[1-9][0-9]*
```

回答（計算しない場合）

- 10行程度

```
def expression: Parser[Any] = additive

def additive: Parser[Any] = P(
  (multitive ~ ($("+") ~ multitive | $("-") ~ multitive).*)
)

def multitive: Parser[Any] = P(
  (primary ~ ($("*") ~ primary | $("/") ~ primary).*)
)

def primary: Parser[Any] = P(
  ($("(") ~> expression <~ $(")")) | number
)

def number: Parser[Any] = oneOf('0'to'9')
```

- BNFの構造をそのまま反映している

回答（計算する場合）（1）：

```
def expression: Parser[Int] = additive

def additive: Parser[Int] = P((multitive ~
  ($("+") ~ multitive | $("-") ~ multitive).*).map {
  case (l, rs) => rs.foldLeft(l) {
    case (e, ("+", r)) => e + r; case (e, ("-", r)) => e - r
  }
})
```

- a ~ b : 連接。 a に続いて b でパーズ
  - パーズ結果のペアを返す
- a | b : 選択。 a でパーズ。失敗したら b でパーズ
- e.* : 繰り返し。失敗するまで、 e でパーズ
  - パーズ結果の List を返す
- foldLeft で右辺の結果を畳み込む

回答（計算する場合）（2）：

```
def multitive: Parser[Int] = P((primary ~
  ($("*") ~ primary | $("/") ~ primary).*).map {
    case (l, rs) => rs.foldLeft(l) {
      case (e, ("*", r)) => e * r
      case (e, ("/", r)) => e / r
    }
  }
)
```

- foldLeft で右辺の結果を畳み込む

回答（計算する場合）（3）：

```
def primary: Parser[Int] = P(
  ($("(") ~> expression <~ $(")")) | number
)

def number: Parser[Int] = oneOf('0'to'9').*.map {
  digits => digits.mkString.toInt
}
```

- a ~> e <~ b について
  - <~ : ~ と同じだが、右の結果を捨てる
  - ~> : ~ と同じだが、左の結果を捨てる
  - e の結果だけを取り出せる
- oneOf : Unicodeの範囲に含まれるか

テスト

```
assert(3 == calculate("1+2"))
assert(-1 == calculate("1-2"))
assert(2 == calculate("1*2"))
assert(0 == calculate("1/2"))
assert(2 == calculate("1+2*3/4"))
assert(2 == calculate("(1+2)*3/4"))
assert(0 == calculate("(1+2)*(3/4)"))
```

解説（1）：

- 解析結果は Option[A] 型
  - Some[A] か None
- Parser[T] は関数
  - 引数は文字列（ String ）
  - Option[(T, String)] を返す

```
object Parsers {
  type Result[+A] = Option[(A, String)]
  type Parser[+A] = String => Result[A]
}
```

解説（1）：基本的なパーザ

```
def oneOf(seq: Seq[Char]): Parser[String] = {input =>
  if(input.length == 0 || !seq.exists(_ == input.charAt(0))) None
  else Some(input.substring(0, 1) -> input.substring(1))
}

def $(literal: String): Parser[String] = {input =>
  if(input.startsWith(literal)) {
    Some(literal -> input.substring(literal.length))
  else None
}

def P[A](parser: => Parser[A]): Parser[A] = {input =>
  parser(input)
}
```

解説（2）：連接

```
def ~[U](right: Parser[U]): Parser[(T, U)] = {input =>
  self(input) match {
    case Some((value1, next1)) =>
      right(next1) match {
        case Some((value2, next2)) =>
          Some((value1, value2), next2)
        case None => None
      }
    case None => None
  }
}
```

解説（3）：選択

```
def |(right: Parser[T]): Parser[T] = {input =>
  self(input) match {
    case success@Some((_, _)) => success
    case None => right(input)
  }
}
```

解説（3）：繰り返し

```
def * : Parser[List[T]] = {input =>
  def repeat(input: String): (List[T], String) = self(input) match {
    case Some((value, next1)) =>
      val (result, next2) = repeat(next1)
      (value :: result, next2)
    case None =>
      (Nil, input)
  }

  val (result, next) = repeat(input)
  Some(result -> next)
}
```

解説（4）：map

```
def map[U](function: T => U): Parser[U] = {input =>
  self(input) match {
    case Some((value, next)) =>
      Some(function(value) -> next)
    case None => None
  }
}
```

- 結果が Some なら function を結果に適用
  - 結果を加工したいときに使う

解説（5）：~> と <~

```
def ~>[U](that: Parser[U]): Parser[U] = {
  (self ~ that).map { case (l, r) => r }
}

def <~[U](that: Parser[U]): Parser[T] = {
  (self ~ that).map { case (l, r) => l }
}
```

- ~ と map の組み合わせ

まとめ

- 利用側はBNFと同様に記述できた
  - 10行程度
- 作る側も簡潔に記述できた
  - 60行程度
- 第一級関数やパターンマッチの恩恵
- パーザを関数とみなす事の恩恵
  - パーザが自由に合成できる部品となった
  - ~> と <~