Parser Combinators in Scala

Adriaan Moors *Frank Piessens*
Martin Odersky

Report CW 491, Feb 2008

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Parser Combinators in Scala

Adriaan Moors *Frank Piessens*
Martin Odersky

Report CW 491, Feb 2008

Department of Computer Science, K.U.Leuven

Abstract

Parser combinators are well-known in functional programming languages such as Haskell. In this paper, we describe how they are implemented as a library in Scala, a functional object-oriented language. Thanks to Scala's flexible syntax, we are able to closely approximate the EBNF notation supported by dedicated parser generators. For the uninitiated, we first explain the concept of parser combinators by developing a minimal library from scratch. We then turn to the existing Scala library, and discuss its features using various examples.

Chapter 1

Introduction

In this paper we describe our Scala [9] implementation of an embedded domain-specific language (DSL) for specifying grammars in a EBNF-like notation. We use the parser combinators approach [11, 5, 2, 10, 7] to implement this language as a Scala library.

The next chapter provides a tutorial on how to implement a library of parser combinators from scratch. No prior knowledge of functional programming is required. We do assume familiarity with Scala, and basic notions of parsing. Chapter 3 exemplifies the utility of the actual library of parser combinators that is part of the standard Scala distribution.

By defining the parser directly in the general-purpose language that is used to process the parser's results, the full power of that language is at our disposal when constructing the grammar. There is no need to learn a separate language for things that can already be expressed in the general-purpose language. Thus, the only bump in the learning curve is understanding the combinators offered by the library.

Any sufficiently complicated DSL is doomed to re-invent many of the mechanisms already available in a general-purpose language. For example, Bracha shows how to leverage inheritance in his Executable Grammars, so that the concrete grammar and the construction of the corresponding abstract syntax trees can be decoupled [1] in Newspeak, a unityped [4] language. It remains an open question to devise a type system that can harness this scheme.

The downsides of not having a special-purpose language for describing the grammar are limited: essentially, the syntax may be more verbose and performance may be affected. We will show that our library minimises the syntactical overhead. We leave performance benchmarks and optimisations for future work. The Parsec library in Haskell was shown to be quite efficient [7], so we expect similar results can be achieved in our library if practical use indicates optimisation is necessary.

Our parser combinators produce back-tracking top-down parsers that use recursive descent with arbitrary look-ahead and semantic predicates. The library provides combinators for ordered choice and many other high-level constructs for repetition, optionality, easy elimination of left-recursion, and so on. In Chapter 3, we will show how to incorporate variable scoping in a grammar.

There are a few known limitations on the expressible grammars. Left-recursion is not supported directly, although we provide combinators that largely obviate the need for it. In principle, it is possible to implement a different back-end that adds support for left recursion, while exposing the same interface. Recent work has shown how to implement support for left-recursion in Packrat parsers [12]. We have investigated using Packrat parsing [3], but defer a full implementation until we focus on optimising the performance of our library. Besides the implementation effort, Packrat parsing assumes parsers are pure (free from effects), but this

cannot (yet) be enforced in Scala.

Furthermore, the choice operator is sensitive to the order of the alternatives. To get the expected behaviour, parsers that match "longer" substrings should come first. To increase performance, the ordered choice combinator commits to the first alternative that succeeds. In practice, this seems to work out quite nicely.

Chapter 2

Parser Combinators from the Ground Up

2.1 Intuitions

As a first approximation, a parser consumes input. In functional programming, we model an "input consumer" as a function that takes some input and returns the rest of the input that has not been consumed yet.

Thus, the type of a parser that examines a string (as its input) can be written as `String ⇒ String`. The identity function represents a parser that does not consume any input. Another example is a parser that always consumes the first character of its input: `(in: String) ⇒ in.substring(1)`. Now, what should we do when the input is empty? Or, how can we implement a parser that refuses certain input?

Naturally, a parser does not accept just any input – it has to conform to a certain *grammar*. A parser should not just denote how much input it consumed, but also whether it considered the input valid or not. Furthermore, for valid input, a parser typically returns a result based on that input. Invalid input gives rise to an error message.

Let us refine our model of a parser so that it meets these criteria: a parser is a function that takes some input – generalising this to be of the abstract type `Input` – and that produces a result, which is modelled by the type `Result[T]`. Listing 2.1 implements this in Scala.

Given this `SimpleResults` component, we will model a parser that produces results of type `T` as a function of type `Input ⇒ Result[T]`. First, we examine listing 2.1 more carefully.

Listing 2.1: A component for modelling results

```
trait SimpleResults {
  type Input

  trait Result[+T] {
    def next: Input
  }

  case class Success[+T](result: T, next: Input) extends Result[T]
  case class Failure(msg: String, next: Input)
                                              extends Result[Nothing]
}
```

Listing 2.2: Parsing 'x'

```
object XParser extends SimpleResults {
  type Input = String
  val acceptX: Input ⇒ Result[Char] = {(in: String) ⇒
    if(in.charAt(0) == 'x') Success('x', in.substring(1))
    else Failure("expected an x", in)
  }
}
```

A result can be a success, and then it contains a result value of type `T`, or a failure, that provides an error message. In either case, a result specifies how much input was consumed by tracking the input that should be supplied to the following parser.

The declaration **`trait`** `Result[+T]` says `Result` is a *type constructor* that is *covariant* in its first type argument. Because it is a type constructor, we must apply `Result` to a concrete type argument, such as `String`, in order to *construct* a type that can be instantiated, such as `Result[String]` (this is necessary, but not sufficient, as `Result` is an abstract class).

Because of the covariance annotation (the '+'), `Result[A]` and `Result[B]` are related with respect to subtyping in the same way as `A` and `B`. That is, `Result[A] <: Result[B]` if `A <: B` and vice versa. Note that `Nothing` is a subtype of any well-formed type.

Before we study more complicated parsers, consider the parser that only accepts the character `'x'`, as shown in listing 2.2.

Exercise 2.1 (Experimenting) *What happens when you apply this parser to the input* `"xyz"`*? (That is, what is the result of* `acceptX("xyz")`*?) Try to work out the result on paper before pasting*[1] *the listings in the Scala interpreter to verify your expectations.*

Notice how the parser denotes that it consumed the first character of the input: the `next` field of the result is set to the input minus its first character (the substring that starts after the first character).

Exercise 2.2 (Generalisation and Robustness) *Generalise* `acceptX` *so that it can be used to make a parser that matches on other characters than* `'x'`*. Improve it further so that it deals with the empty input.*

2.2 Sequence and Alternation

Now we know how to define parsers that accept a single element of input, we will see how these can be combined into parsers that recognise more complex grammars. Once it is clear how to implement the two most typical ways of combining parsers, we shall gradually improve our implementation.

Listing 2.3 shows a straightforward implementation of alternation and sequencing. A `Parser` is a subclass of `Input ⇒ Result[T]`, which is syntactic sugar for `Function1[Input, Result[T]]`. Thus, an instance of `Parser[T]` is (an object that represents) a function that takes an instance of `Input` to a `Result[T]`. As a reminder, the abstract `apply` method, which is inherited from `Function1`, is included explicitly in listing 2.3.

[1] Note that => is typeset as ⇒.

If `p` and `q` are `Parser`'s, `p | q` is a `Parser` that first tries `p`. If this is successful, it returns `p`'s result. `q` is only tried if `p` fails. Similarly, `p ~ q` results in a parser that succeeds if `p` *and then* `q` succeeds. There is nothing special about `~` and `|`: they are just methods that happen to have names that consist solely of symbols. `p ~ q | r` is syntactic sugar for `(p.~(q)).|(r)`. The precedence and associativity of method names is defined in the Scala reference [8, Sec. 6.12.3].

The `|` method takes an argument `p`, which is a parser that produces results of type `U`. `U` must be a super-type of the type of the results produced by the parser on which the `|` method is called (denoted as `Parser.this`). The method returns a new parser that produces results of type `U` by first trying `Parser.this`, or else `p`.

When the parser that is returned by `|`, is applied to input, it passes this input on to `Parser.this` (the parser on which `|` was called originally). The result of this parser is examined using pattern matching. The first case is selected when `Parser.this` failed. Then (and only then – see below), the alternative parser is computed and applied to the same input as `Parser.this`. The outcome of this parser determines the result of the combined parser. In case `Parser.this` succeeded, this result is simply returned (and `p` is never computed).

Note that `p` is passed call-by-name (CBN) [2]. When `|` is called, the compiler silently wraps a zero-argument function around its argument, so that its value is not yet computed. This is delayed until `p` is "forced" in the body of the method.

More concretely, *every time* `p`'s actual value is *required*, the wrapper function is applied (to the empty list of arguments). Consider the expression `q | p`. When the `|` combinator is called on `q`, `p`'s value need not be known, as the method simply returns a new parser. This combined parser textually contains an occurrence of `p`, but its actual value does not become relevant until the combined parser is applied to input *and* `q` fails.

The sequence combinator is implemented by the `~` method. The main difference is that we must be more careful with the input that is passed to each parser: the first one receives the input supplied to the combined parser, and the second parser (`p`) is applied to the input that was left over after the first one. If both parsers succeed, their results are combined in the pair `(x, x2)`. The first parser to fail determines the unsuccessful outcome of the combined parser.

Exercise 2.3 (Cycles) *What happens when you leave off the '⇒' of the types of the arguments of `|` and `~`? Write down a grammar that relies on the arguments being call-by-name.*

With these combinators in place, let us construct our first working parser by combining a simple parser that accepts a single character into one that accepts the string that consists of one or more times "oxo", where subsequent occurrences are separated by a white space.

First, we generalise our `acceptX` parser generator in listing 2.4. The only non-trivial difference is that we allow checking for the end of input using the parser generated by the method `eoi`. Internally, we use `0` to denote the end of the input has been reached.

Finally, the object `OXOParser` (listing 2.5) constitutes a valid Scala program whose first argument must be exactly `"oxo"`, `"oxo␣oxo"`, or `"oxo␣oxo␣oxo"`, and so on.

To emphasise that this grammar is expressed purely as method calls, listing 2.6 reformulates `oxo` and `oxos` in a more traditional syntax.

[2] Call-by name arguments are denoted by prefixing the argument type with ⇒.

Listing 2.3: Combinators for Alternation and Sequencing

```
trait SimpleParsers extends SimpleResults {
  trait Parser[+T] extends (Input ⇒ Result[T]) {
    def apply(in: Input): Result[T]

    def | [U >: T](p: ⇒ Parser[U]): Parser[U]
      = new Parser[U]{ def apply(in: Input) =
          Parser.this(in) match {
            case Failure(_, _) ⇒ p(in)
            case Success(x, n) ⇒ Success(x, n)
          }
        }

    def ~ [U](p: ⇒ Parser[U]): Parser[Pair[T, U]]
      = new Parser[Pair[T, U]]{ def apply(in: Input) =
          Parser.this(in) match {
            case Success(x, next) ⇒ p(next) match {
              case Success(x2, next2) ⇒ Success((x, x2), next2)
              case Failure(m, n) ⇒ Failure(m, n)
            }
            case Failure(m, n) ⇒ Failure(m, n)
          }
        }
  }
}
```

Listing 2.4: Parsing Strings

```
trait StringParsers extends SimpleParsers {
  type Input = String
  private val EOI = 0.toChar

  def accept(expected: Char) = new Parser[Char]{
    def apply(in: String) =
      if(in == "") {
        if(expected == EOI)
          Success(expected, "")
        else
          Failure("no more input", in)
      } else if(in.charAt(0) == expected)
        Success(expected, in.substring(1))
      else
        Failure("expected '"+expected+"'", in)
  }

  def eoi = accept(EOI)
}
```

Listing 2.5: oxo oxo ... oxo

```
object OXOParser extends StringParsers {
  def oxo = accept('o') ~ accept('x') ~ accept('o')
  def oxos: Parser[Any] =
    ( oxo ~ accept(' ') ~ oxos
    | oxo
    )

  def main(args: Array[String]) = println((oxos ~ eoi)(args(0)))
}
```

Listing 2.6: oxo oxo ... oxo (hold the syntactic sugar)

```
  def oxo = accept('o').~(accept('x')).~(accept('o'))
  def oxos: Parser[Any] = oxo.~(accept(' ')).~(oxos).|(oxo)
```

Exercise 2.4 *Verify the correctness of listing 2.6 by compiling the version of listing 2.5 using* `scalac -Xprint:typer`*, which prints the compiled source after type checking. At that point, syntactic sugar has been expanded and the omitted types have been inferred.*

Exercise 2.5 *Write down the order in which the various parsers are executed for a given input. Work out at least one example for input on which the parser should fail. Verify your solution using the* `log` *combinator of listing 2.7. What changes when you omit the* `~ eoi` *in the* `main` *method?*

2.3 Factoring out the plumbing

We will now improve our first implementation using standard techniques from functional programming. Our combinators for alternation and sequencing worked correctly, but were somewhat tricky to get right. More specifically, we had to pay attention to "threading" the input correctly when combining the parsers. In this section, we will encapsulate this.

Listing 2.7: Logging

```
def log[T](p: ⇒ Parser[T])(name: String) = new Parser[T]{
  def apply(in: Input) : Result[T] = {
    println("trying␣"+ name +"␣at␣\'"+ in + "\'")
    val r = p(in)
    println(name +"␣-→␣"+ r)
    r
  }
}
```

Listing 2.8: Improved Results

```
trait SimpleResults {
  type Input

  trait Result[+T] {
    def next: Input

    def map[U](f: T ⇒ U): Result[U]
    def flatMapWithNext[U](f: T ⇒ Input ⇒ Result[U]): Result[U]
    def append[U >: T](alt: ⇒ Result[U]): Result[U]
  }

  case class Success[+T](result: T, next: Input) extends Result[T] {
    def map[U](f: T ⇒ U)
      = Success(f(result), next)
    def flatMapWithNext[U](f: T ⇒ Input ⇒ Result[U])
      = f(result)(next)
    def append[U >: T](alt: ⇒ Result[U])
      = this
  }

  case class Failure(msg: String, next: Input) extends Result[Nothing] {
    def map[U](f: Nothing ⇒ U)
      = this
    def flatMapWithNext[U](f: Nothing ⇒ Input ⇒ Result[U])
      = this
    def append[U](alt: ⇒ Result[U])
      = alt
  }
}
```

Listing 2.9: Parsing

```
trait SimpleParsers extends SimpleResults {
  abstract class Parser[+T] extends (Input ⇒ Result[T]) {
    def apply(in: Input): Result[T]

    def flatMap[U](f: T ⇒ Parser[U]): Parser[U]
      = new Parser[U]{def apply(in: Input)
                        = Parser.this(in) flatMapWithNext(f)}

    def map[U](f: T ⇒ U): Parser[U]
      = new Parser[U]{def apply(in: Input)
                        = Parser.this(in) map(f)}

    def | [U >: T](p: ⇒ Parser[U]): Parser[U]
      = new Parser[U]{def apply(in: Input)
                        = Parser.this(in) append p(in)}

    def ~ [U](p: ⇒ Parser[U]): Parser[Pair[T, U]]
      = for(a ← this; b ← p) yield (a,b)
  }
}
```

The new `Result`, as defined in listing 2.8, provides three simple methods. When called on a `Success`, `map` produces a new `Success` that contains the transformed result value. The `map` method is useful when transforming the result of a combinator. The function passed to `flatMapWithNext` produces a new `Result` based on the result value and `next`. We will use this method for chaining combinators. Finally, the only method whose implementation does not follow directly from its type signature, is `append`. Our simple model of results does not allow for multiple successful results, so appending a result to a success is a no-op. However, for more sophisticated systems that allow multiple results, `append` would add `alt` to a collection. Here, it simply returns the current `Result`.

For a `Failure`, the methods behave dually.

These methods may seem a bit arbitrary, but with them, we can re-implement `Parser` as shown in listing 2.9. `flatMap`, `map`, and `|` simply create `Parser`'s that call the corresponding methods on the results they produce for a given input.

Finally, `~` can now be implemented very naturally using Scala's for-comprehension syntax. Its implementation can be read as: "perform parser **`this`** and *bind* its result to `a` in the *next* computation, which performs `p` and maps its result `b` to the pair `(a, b)`". Note that we do not have to do *any* bookkeeping on which input to pass to which parser! (This interpretation of for-comprehensions explains why `flatMap` is sometimes also called "`bind`".)

For-comprehensions [8, Sec 6.19] are syntactic sugar for nested calls to `flatMap`, `map`, and `filter` (we do not yet use the latter). More concretely, **`for`**`(a ← this; b ← p)` **`yield`** `(a,b)` is shorthand for **`this`**`.flatMap{a ⇒ p.map{b ⇒ (a, b)}}`.

Our existing `OXOParser` can be used as-is with this new version of `SimpleParsers`.

Exercise 2.6 *Implement* **`def`** `flatMap[U](f: T ⇒ Result[U]): Result[U]` *in the appropriate classes. Experiment with for-comprehensions over* `Result`*'s. Can you think of other applications besides parsing?*

Listing 2.10: Reducing SimpleParsers

```
trait SimpleParsers extends SimpleResults {
  def Parser[T](f: Input ⇒ Result[T])
    = new Parser[T]{ def apply(in: Input) = f(in) }

  abstract class Parser[+T] extends (Input ⇒ Result[T]) {
    def flatMap[U](f: T ⇒ Parser[U]): Parser[U]
      = Parser{in ⇒ Parser.this(in) flatMapWithNext(f)}

    def map[U](f: T ⇒ U): Parser[U]
      = Parser{in ⇒ Parser.this(in) map(f)}

    def | [U >: T](p: ⇒ Parser[U]): Parser[U]
      = Parser{in ⇒ Parser.this(in) append p(in)}

    def ~ [U](p: ⇒ Parser[U]): Parser[Pair[T, U]]
      = for(a ← this; b ← p) yield (a,b)
  }
}
```

Exercise 2.7 *Convince yourself that the two implementations of* `~` *and* `|` *are indeed equivalent, without actually executing anything. Inline the method calls made by the new version until you arrive at our first implementation. (See p. 25 for the solution to this exercise.)*

Exercise 2.8 *Improve* `OXOParser` *so that* `oxos` *produces a parser that returns a list of strings (where each string equals* `"oxo"`*). (Hint: use* `map`*.)*

2.4 More Polish and Advanced Features

2.4.1 Encapsulating Parser Instantiation

Listing 2.10 shows how we can get rid of the repetitive `new Parser[U]{def apply(in: Input) = ... }` fragment. We simply define a method `Parser` that makes a new instance of `Parser` given a function that fully defines the parser's logic. (We've also omitted the abstract `apply` method – as said before, it is inherited from `Function1` anyway.)

2.4.2 Improving `accept`

As it stands, our oxo-grammar is already pretty close to BNF notation:

```
  def oxo = accept('o') ~ accept('x') ~ accept('o')
  def oxos: Parser[Any] =
    ( oxo ~ accept(' ') ~ oxos
    | oxo
    )
```

We will now see how we can reduce this to:

```
  def oxo = 'o' ~ 'x' ~ 'o'
  def oxos: Parser[Any] =
    ( oxo ~ ' ' ~ oxos
    | oxo
    )
```

Without further intervention, this code will not compile, as `Char` does not have a `~` method. We can use implicit conversions to remedy this.

Simply adding the **`implicit`** keyword to the signature of `accept` does the trick:

```
implicit def accept(expected: Char): Parser[Char] = ... // as before
```

Now, whenever the compiler encounters a value of type `Char` whereas its expected type is `Parser[Char]`, it will automatically insert a call to the `accept` method!

In the following sections we will see how we can express the oxo-grammar even more succinctly using more advanced combinators. First, we will refactor `accept` and add initial machinery to improve error-reporting.

2.4.3 Filtering

It is now time to change the `accept` we defined in listing 2.4, so that it is less tightly coupled to the kind of input we are examining.

`accept` generates a parser that accepts a given element of input. To do this, it suffices that it can retrieve one element of input as well as the input that follows this element. We introduce a new abstract type `Elem` to represent an element of input, which can be retrieved using **`def`** `first(in: Input): Elem`. The rest of the input is returned by **`def`** `rest(in: Input): Input`. Given these abstractions, we can implement `accept` once and for all.

For reference, listing 2.11 shows the simplified `StringParsers`, which now only contains the essential methods that deal with the specific kind of input that is supported by this component. (Note that `first` and `rest` correspond to the typical `head` and `tail` functions that are used to access lists in functional programming.)

Listing 2.11: StringParsers with first and rest

```
trait StringParsers extends SimpleParsers {
  type Input = String
  type Elem = Char
  private val EOI = 0.toChar

  def first(in: Input): Elem = if(in == "") EOI else in(0)
  def rest(in: Input): Input = if(in == "") in else in.substring(1)

  def eoi = accept(EOI) // accept is now defined in SimpleParsers
}
```

To further deconstruct `accept`'s functionality, we define a simple parser that accepts any given piece of input and passes the *rest* of the input on to the next parser. We will introduce another method that decides whether this piece of input is acceptable.

```
def consumeFirst: Parser[Elem] = Parser{in ⇒
  Success(first(in), rest(in))
}
```

We need one more standard method in `Parser`: `filter`. This method wraps an existing parser so that it accepts only results that meet a certain predicate (which is modelled as a function `T ⇒ Boolean`).

```
  def filter(f: T ⇒ Boolean): Parser[T]
    = Parser{in ⇒ this(in) filter(f)}
```

Finally, we recover `accept` as the parser that filters the parser that consumes any input, by checking that the produced result is equal to the expected element.

```
def acceptIf(p: Elem ⇒ Boolean): Parser[Elem]
  = consumeFirst filter(p)
implicit def accept(e: Elem): Parser[Elem] = acceptIf(_ == e)
```

Exercise 2.9 *Implement the corresponding* `filter` *method in* `Result` *and its subclasses. Note that this is not entirely trivial! Try out your implementation. Compare the resulting* `accept` *method to our original one – where did it go wrong? (See p. 26 for the solution and an explanation.)*

2.4.4 More Combinators

To make it easier to define more advanced combinators, we add three more methods to `Parser`:

```
  def ~> [U](p: ⇒ Parser[U]): Parser[U] = for(a ← this; b ← p) yield b
  def <~ [U](p: ⇒ Parser[U]): Parser[T] = for(a ← this; b ← p) yield a

  def ^^ [U](f: T ⇒ U): Parser[U] = map(f)
```

These methods allow sequencing parsers when we only care about the result of either the right or the left one. Because the following combinators heavily rely on `map`, we define a shorthand for it: `^^`.

The combinators in listing 2.12 implement optionality, repetition (zero or more, or one or more), repetition with a separator, and chaining (to deal with left-recursion). These combinators were inspired by Hutton and Meijer's excellent introduction, in which they explain them in more detail [6]. We will not discuss their implementation, but we will use them later in this paper.

We can leverage these combinators to further shorten our oxo-parser as follows (additionally, we improve the output):

```
object OXOParser extends StringParsers with MoreCombinators {
  def oxo  = acceptSeq("oxo") ^^ {x ⇒ x.mkString("")}
  def oxos = rep1sep(oxo, ' ')

  def main(args: Array[String]) = println((oxos <~ eoi)(args(0)))
}
```

Listing 2.12: More Advanced Combinators

```
trait MoreCombinators extends SimpleParsers {
  def success[T](v: T): Parser[T]
    = Parser{in ⇒ Success(v, in)(Failure("unknown␣failure", in))}

  def opt[T](p: ⇒ Parser[T]): Parser[Option[T]]
    = ( p ^^ {x: T ⇒ Some(x)}
      | success(None)
      )

  def rep[T](p: ⇒ Parser[T]): Parser[List[T]]
    = rep1(p) | success(List())

  def rep1[T](p: ⇒ Parser[T]): Parser[List[T]]
    = rep1(p, p)

  def rep1[T](first: ⇒ Parser[T], p: ⇒ Parser[T]): Parser[List[T]]
    = first ~ rep(p) ^^ mkList

  def repsep[T, S](p: ⇒ Parser[T], q: ⇒ Parser[S]): Parser[List[T]]
    = rep1sep(p, q) | success(List())

  def rep1sep[T, S](p: ⇒ Parser[T], q: ⇒ Parser[S]): Parser[List[T]]
    = rep1sep(p, p, q)

  def rep1sep[T, S](first: ⇒ Parser[T], p: ⇒ Parser[T], q: ⇒ Parser[S])
   : Parser[List[T]]
    = first ~ rep(q ~> p) ^^ mkList

  def chainl1[T](p: ⇒ Parser[T], q: ⇒ Parser[(T, T) ⇒ T]): Parser[T]
    = chainl1(p, p, q)

  def chainl1[T, U](first: ⇒ Parser[T], p: ⇒ Parser[U], q: ⇒ Parser[(T,
    U) ⇒ T]): Parser[T]
    = first ~ rep(q ~ p) ^^ {
        case (x, xs) ⇒ xs.foldLeft(x){(_, _) match {case (a, (f, b)) ⇒ f
   (a, b)}}      }

  def acceptSeq[ES <% Iterable[Elem]](es: ES): Parser[List[Elem]] = {
     def acceptRec(x: Elem, pxs: Parser[List[Elem]]) = (accept(x) ~ pxs)
   ^^ mkList
     es.foldRight[Parser[List[Elem]]](success(Nil))(acceptRec _ _)
  }

  private def mkList[T] = (_ : Pair[T, List[T]]) match {case (x, xs) ⇒ x
   :: xs }
}
```

2.4.5 Controlling Backtracking

To manage the way in which a parser performs backtracking, we introduce a different type of unsuccessful result, `Error`, whose `append` method does not try the alternative. This effectively disables backtracking. In order to trigger this behaviour, we add the `dontBacktrack` method to `Parser`, which turns failures into errors. The implementation of `dontBacktrack` is more subtle than simply turning `Failure`s into `Error`s, as a `Success` must also carry with it that subsequent `Failure`s must be turned into `Error`s.

We explain the usage of the `~!` combinator in section 2.4.6.

Listing 2.13: Error

```
case class Error(msg: String, next: Input) extends Result[Nothing] {
  def map[U](f: Nothing ⇒ U)                          = this
  def flatMap[U](f: Nothing ⇒ Result[U])              = this
  def flatMapWithNext[U](f: Nothing ⇒ Input ⇒ Result[U]) = this
  def filter(f: Nothing ⇒ Boolean): Result[Nothing] = this
  def append[U](alt: ⇒ Result[U])                     = this

  def explain(ei: String) = Error(ei, next)
}
```

Listing 2.14: Disabling Backtracking

```
    def dontBacktrack: Parser[T] = ... /* a Parser whose Failures become
   Errors.
    This behaviour propagates through all other parsers that follow this
   one. */

    def ~! [U](p: ⇒ Parser[U]): Parser[Pair[T, U]]
      = dontBacktrack ~ p
```

2.4.6 Error Reporting

Better Messages

Until now, the user of our library could not easily influence the error message in case a parser failed. To solve this, we add three more methods to `Parser`:

```
  def explainWith(msg: Input ⇒ String): Parser[T] = Parser{in ⇒
    this(in) explain msg(in)
  }

  def explain(msg: String): Parser[T] = Parser{in ⇒
    Parser.this(in) explain msg
  }

  def expected(kind: String): Parser[T]
```

```
 = explainWith{in ⇒ ""+ kind +"␣expected,␣but␣\'"+ first(in) +"\'␣
found."}
```

As usual, this requires a modest change to `Result`. This is the implementation for `Failure`:

```
def explain(ei: String) = Failure(ei, next)
```

A parser can now be customised with an appropriate error message by calling one of these new methods. For example, here is an improved version of `accept`:

```
implicit def accept(e: Elem): Parser[Elem] = acceptIf(_ == e).expected(e
 .toString)
```

Failing Better

Besides better error messages, we can also improve the location where parsing breaks down. Generally, the further down the input, the more informative the input will be. By disabling backtracking when we know it will not help, the parser is prevented from going back to an earlier point in the input, thus improving the error message.

For example, consider a simplified parser for a single (Scala-style) member declaration:

```
def member = ("val" ~! ident | "def" ~! ident)
```

As soon as we have encountered the **`val`** keyword, but fail to parse an identifier, there is no point in going back to look for the **`def`** keyword. Thus, if a branch of the ordered choice signals an error, the subsequent alternatives are pre-empted.

Chapter 3

Scala's Parser Combinators by Example

The library of parser combinators in the standard Scala distribution is very similar to the library that we developed in the previous chapter. For detailed documentation, please consult the API-documentation included in the distribution. In this chapter, we discuss a number of interesting examples.

3.1 Parsing and evaluating simple arithmetic expressions

An introduction to parsing would not be complete without a parser for arithmetic expressions. Except for the import statements, listing 3.1 is a complete implementation of such a parser. Let's dissect this example line by line.

The first line declares an application object `ArithmeticParser`, which is suitable as a main class (running it will evaluate the expressions in its body). More importantly, `ArithmeticParser` is a `StdTokenParsers`, which means it contains parsers that operate on a stream of tokens. `StdTokenParsers` earned its 'Std' prefix by providing a couple of commonly-used tokens such as keywords, identifiers and literals (strings and numbers). If these defaults don't suit you, simply go over its head and use its super class, `TokenParsers`.

A token parser abstracts from the type of tokens it parses. This abstraction is made concrete in line 2: we use `StdLexical` for our lexical analysis. It's important to note that lexical analysis is done using the same parsers that we use for syntax analysis. The only difference is that lexical parsers operate on streams of characters to produce tokens, whereas syntactical parsers consume streams of tokens and produce yet a more complex type of structured data. To conclude the lexical aspects of our example, line 3 specifies which characters should be recognised as delimiters (and returned as keyword tokens).

Now we get to the actual grammar. `expr` returns a parser that parses a list of `term`'s, separated by either a `"+"` or a `"-"` and returns an integer, which, unsurprisingly, corresponds to the evaluation of the expression. An implicit conversion (`keyword` in `StdTokenParsers`) automatically lifts a string to the `UnitParser` that matches that string and returns nothing. `UnitParser`'s are parsers whose results are discarded.

In general, `p*` means repeat parser `p` zero or more times and collect the results of `p` in a list. `p*(q)` generalises this to repeating `p` alternated with `q`. If `q` returns a result (i.e., it's not a `UnitParser`), its result must be a function that combines the result of `p` when called right before it and that of `p` when called right after it. In our case, `q` is `"+" ^^ {(x: int, y: int) => x + y} | "-" ^^ {(x: int, y: int) => x - y}`, which, when it sees a `"+"`, returns the function that sums two integers, and similarly when it encounters a `"-"`. `p*(q)` uses this function to "fold" the list of results into a single result. Again, in our case, this

Listing 3.1: Parsing 1+1 etc.

```
object ArithmeticParser extends StdTokenParsers with Application {
  type Tokens = StdLexical ; val lexical = new StdLexical
  lexical.delimiters ++= List('(', ')', '+', '-', '*', '/')

  def expr = term*("+" ^^ {(x: int, y: int) ⇒ x + y}
                 | "-" ^^ {(x: int, y: int) ⇒ x - y})
  def term = factor*("*" ^^ {(x: int, y: int) ⇒ x * y}
                   | "/" ^^ {(x: int, y: int) ⇒ x / y})
  def factor: Parser[int] = "(" ~ expr ~ ")"
                          | numericLit ^^ (.toInt)

  Console.println(expr(new lexical.Scanner("1+2*3*7-1")))
}
```

Listing 3.2: Desugared version of listing 3.1.

```
def expr = chainl1(term, (keyword("+").^^{(x: int, y: int) ⇒ x + y}).|(
   keyword("-").^^{(x: int, y: int) ⇒ x - y}))
def term = chainl1(factor, (keyword("*").^^{(x: int, y: int) ⇒ x * y}).|(
   keyword("/").^^{(x: int, y: int) ⇒ x / y}))
def factor: Parser[int] = keyword("(").~(expr.~(keyword(")"))).|(
   numericLit.^^(x ⇒ x.toInt))
```

is the sum (or subtraction) of the constituent terms. When `q` is a `UnitParser`, it's freed from returning such a function and the combinator just collects `p`'s results in a list.

Let's pick `"+" ^^ {(x: int, y: int) ⇒ x + y} | "-" ^^ {(x: int, y: int) ⇒ x - y}` apart a bit further. Because of Scala's operator precedence, `|` is the first combinator to be applied. `p | q` is the parser that first tries `p` on the input, and if successful, just returns it result. If its result was `Failure` (and not `Error`), the second parser, `q` is used. Note that this combinator is sensitive to the ordering of its constituents. It does not try to produce all possible parses – it stops as soon as it encounters the first successful result.

The `^^` combinator takes a parser and a function and returns a new parser whose result is the function applied to the original parser's result. In the case of a `UnitParser`, the function can only be a constant function (i.e., a value).

Finally, we create a scanner that does lexical analysis on a string and returns a stream of tokens, which is then passed on to the expression parser. The latter's result is then printed on the console.

To illustrate what all this syntactical sugar really boils down to, listing 3.2 shows what we'd have to write if Scala's syntax wasn't as liberal. In line with the stoic approach, the code also doesn't use implicit conversions.

3.2 Context sensitivity in parsing XML

Listing 3.3, which was inspired by [7], shows how to make context-sensitive parsers. It is a self-contained parser for an extremely minimal subset of XML. Thus, it also demonstrates

Listing 3.3: A sketch for a context-sensitive parser for XML

```
import scala.util.parsing.combinator._

object XMLParser extends Parsers with Application {
  type Elem = Char

  trait Node
  case class ContainerNode(name: String, content: List[Node]) extends Node
  case class TextNode(content: String) extends Node

  def str1(what: String, pred: Char ⇒ Boolean) = rep1(elem(what, pred))
   ^^ (_.mkString(""))

  def openTag: Parser[String] = '<' ~> str1("tag␣name", _.isLetter) <~ '>'
  def endTag(name: String) = ('<' ~ '/')  ~> accept(name.toList) <~ '>'
  def xmlText: Parser[Node] = str1("xml␣text", {c ⇒ !(c == '<' || c ==
   '>')}) ^^ TextNode

  def xml: Parser[Node] = (
        (openTag into {name ⇒ rep(xml) <~ endTag(name) ^^ (ContainerNode(
   name, _))})
      | xmlText )


  import scala.util.parsing.input.CharArrayReader

  def phrase[T](p: Parser[T]): Parser[T] = p <~ accept(CharArrayReader.
   EofCh)

  println(phrase(xml)(new CharArrayReader("<b>bold</b>".toArray)))
}
```

how combinator parsers can be used for scanner-less parsing. Although it is typically more convenient to separate lexical and syntactical parsing, they can also be performed by the same parser.

The essential part of the example is:

```
openTag into {name ⇒ rep(xml) <~ endTag(name) ^^ (ContainerNode(name, _))
   }
```

This constructs a parser that first tries the `openTag` parser, and if successful, feeds its result into the next part:

```
rep(xml) <~ endTag(name) ^^ (ContainerNode(name, _))
```

This parser, where `name` is bound to the tag-name that was parsed by `openTag`, accepts any number of nested constructs, and a closing tag with the right name. Thus, the end result is that this parser recognises when tags are not properly closed.

Listing 3.4: Parser for the Lambda calculus that tracks variable binding

```
trait LambdaParser extends Syntax with Lexical with ContextualParsers {
  // Context tracks which variables are bound,
  // the generic functionality is implemented in ContextualParsers
  // Here, we only specify how to create our Context class
  object Context extends ContextCompanion {
    def apply(f: String ⇒ Option[Name]): Context
      = new Context { def apply(n: String) = f(n) }
  }

  // Since parsing a term depends on the variable bindings that we have
  // previously added to the context, we put these parsers in a Context,
  // which provides the `bind`, `in`, and `bound` combinators.
  trait Context extends ContextCore {
    def term = chainl1(termSingle, ws ^^^ (App(_: Term, _: Term)))
    def termSingle: Parser[Term] =
      ( '(' ~> term <~ ')'
      | '\\' ~> bind(wss(ident)) >> in{ctx ⇒ '.' ~> wss(ctx.term)} ^^ Abs
      | bound(ident)                                               ^^ Var
      )
  }

  import scala.util.parsing.input.Reader
  def parse(input: Reader[Char]) = (wss(Context.empty.term) <~ eoi)(input)
}
```

3.3 Tracking variable binding

Listing 3.4 implements a parser for the lambda calculus that also enforces the scoping rules for variables. Essentially, we nest our productions in a class `Context`, which models the current scope. Then, in a given scope, a term is a sequence of applications, where a single term in an application may be a parenthesized term, an abstraction, or a bound variable. An abstraction binds an identifier[1], and brings it into scope by calling the `term` production on the context that is produced by the `bind` combinator.

The combinators that maintain which variables are bound are show in Listing 3.5. Basically, a context is a function that takes a variable name (as a `String`), and returns a (unique) `Name` that represents the variable, if it is bound. The `bind` combinator is parameterised in a parser that yields a string-representation `name`, which is then turned into a fresh name `n`. The result of this combinator is a pair that consists of a new context that binds `name` to `n`, and `n`. This redundancy is needed for the implementation of `in`, which wraps a parser that parses a construct (of type `T`) in which a variable is bound, in a `\\[T]`.

The `bound` combinator succeeds if the given name (as a `String`) is in scope, and returns the corresponding `Name`.

Finally, Listing 3.6 shows the essence of the case classes that model the abstract syntax tree.

The implementation of the `Binding` trait is out of scope for this paper. It should be clear that these combinators are amenable to most common approaches to dealing with variable binding.

Listing 3.5: Infrastructure for dealing with variable binding

```
trait Binding {
  // represents a variable name
  type Name
  // creates a fresh name
  def Name(n: String): Name

  // something that contains Name's
  type HasBinder[T]

  // some construct of type T in which the Name n is bound
  type \\[T]
  def \\[T](n: Name, scope: HasBinder[T]): \\[T]
}

trait ContextualParsers extends Parsers with Binding {
  type Context <: ContextCore

  val Context: ContextCompanion
  trait ContextCompanion {
    val empty: Context = apply{name ⇒ None}
    def apply(f: String ⇒ Option[Name]): Context
  }

  trait ContextCore extends (String ⇒ Option[Name]) {
    def bind(nameParser: Parser[String]): Parser[(Context, Name)]
      = (for(name ← nameParser) yield {
           val n=Name(name)
           (this(name) = n, n)
         })

    def bound(nameParser: Parser[String]): Parser[Name]
      = (for(name   ← nameParser;
             binder ← lookup(name)) yield binder)

    def lookup(name: String): Parser[Name] = this(name) match {
      case None ⇒ failure explain("unbound␣name:␣"+name)
      case Some(b) ⇒ success(b)
    }

    def in[T](p: Context ⇒ Parser[HasBinder[T]]): Pair[Context, Name] ⇒
   Parser[\\[T]] = {case (ctx, n) ⇒ p(ctx) ^^ (\\(n, _))}

    def update(rawName: String, binder: Name): Context = Context{name ⇒
   if(name == rawName) Some(binder) else this(name) }
  }
}
```

Listing 3.6: Abstract syntax (details omitted)

```
  trait Term extends HasBinder[Term]
  case class Var(name: Name) extends Term
  case class Abs(abs: \\[Term]) extends Term
  case class App(fun: Term, arg: Term) extends Term
```

Listing 3.7: Lexical parsing

```
trait Lexical extends Parsers {
  type Elem = Char
  import scala.util.parsing.input.CharArrayReader

  // these combinators do the lexical analysis, which we have not
    separated explicitly from the syntactical analysis
  def letter = acceptIf(_.isLetter) expected("letter")
  def digit = acceptIf(_.isDigit) expected("digit")
  def ws = rep1(accept(' ')) expected("whitespace")
  // hint: only wrap wss(...) around the parsers that really need it
  def wss[T](p: Parser[T]): Parser[T] = opt(ws) ~> p <~ opt(ws)
  def ident = rep1(letter, letter | digit) ^^ {_.mkString("")} expected("
    identifier")
  def eoi = accept(CharArrayReader.EofCh)
}
```

[1]The identifier may be surrounded by whitespace – the lexical parsers are shown in Listing 3.7.

Acknowledgments

The authors would like to thank Eric Willigers for his feedback on earlier drafts.

Appendix A

Solutions to Selected Exercises

Solution A.1 (Of exercise 2.7) *Listing A.1 details the steps to relate the two implementations of ~.*

Listing A.1: Expansion

```
// given:
def flatMap[U](f: T ⇒ Parser[U]): Parser[U]
  = new Parser[U]{def apply(in: Input) = Parser.this(in) flatMapWithNext(f
    )}

def map[U](f: T ⇒ U): Parser[U]
  = new Parser[U]{def apply(in: Input) = Parser.this(in) map(f)}

def flatMapWithNext[U](f: T ⇒ Input ⇒ Result[U]): Result[U] = this match
     {
  case Success(x, i) ⇒ f(x)(i)
  case Failure(e, n) ⇒ Failure(e, n)
}
def map[U](f: T ⇒ U): Result[U] = this match {
  case Success(x, i) ⇒ Success(f(x), i)
  case Failure(e, n) ⇒ Failure(e, n)
}

// expansion:
def ~ [U](p: ⇒ Parser[U]): Parser[Pair[T, U]]
= for(a ← this; b ← p) yield (a,b)
= this.flatMap{a ⇒ p.map{b ⇒ (a, b)}}
= this.flatMap{a ⇒ new Parser[Pair[T, U]]{def apply(in: Input) = p(in)
   map{b ⇒ (a, b)}}}
= new Parser{def apply(in: Input) =
    Parser.this(in) flatMapWithNext{a ⇒
      new Parser[Pair[T, U]]{def apply(in: Input) =
        p(in) map{b ⇒ (a, b)}
      }
    }
  }
= new Parser{def apply(in: Input) =
    Parser.this(in) match {
      case Success(x, i) ⇒ {a ⇒
```

```
        new Parser[Pair[T, U]]{def apply(in: Input) =
          p(in) map{b ⇒ (a, b)}
        }
      }(x)(i)
      case Failure(e, n) ⇒ Failure(e, n)
    }
  }
= new Parser{def apply(in: Input) =
    Parser.this(in) match {
      case Success(x, i) ⇒
        new Parser[Pair[T, U]]{def apply(in: Input) =
          p(in) map{b ⇒ (x, b)}
        }(i)
      case Failure(e, n) ⇒ Failure(e, n)
    }
  }
= new Parser{def apply(in: Input) =
    Parser.this(in) match {
      case Success(x, i) ⇒ p(i) map{b ⇒ (x, b)}
      case Failure(e, n) ⇒ Failure(e, n)
    }
 }
= new Parser{def apply(in: Input) =
    Parser.this(in) match {
      case Success(x, i) ⇒ p(i) match {
        case Success(x2, i) ⇒ Success({b ⇒ (x, b)}(x2), i)
        case Failure(e, n) ⇒ Failure(e, n)
      }
      case Failure(e, n) ⇒ Failure(e, n)
    }
  }
= new Parser[Pair[T, U]]{ def apply(in: Input) =
    Parser.this(in) match {
      case Success(x, next) ⇒ p(next) match {
        case Success(x2, next2) ⇒ Success((x, x2), next2)
        case Failure(e, n) ⇒ Failure(e, n)
      }
      case Failure(e, n) ⇒ Failure(e, n)
    }
  }
```

Solution A.2 (Of exercise 2.9) *Listing A.2 shows the implementation of* `filter` *in* `Success` *(in* `Failure`*, it simply returns* **`this`***), as well as the correct version of* `consumeFirst`*.*

A correct implementation of `filter` requires each successful result to know what should happen in case it is filtered out: we cannot simply "invent" the failure that occurs when `filter` 's predicate is not met! Therefore, we add a `zero` member to `Success`, which determines how `filter` on a `Success` fails.

This way, `consumeFirst` can produce successes that "un-consume" their input if they are refuted by `filter`. The next parser will then be applied to the input that `consumeFirst` would have consumed, had it been successful.

Listing A.2: Filtering with zero

```
case class Success[+T](result: T, next: Input)(val zero: Failure)
 extends Result[T] {
  def map[U](f: T ⇒ U)
    = Success(f(result), next)(zero)
  def flatMap[U](f: T ⇒ Result[U])
    = f(result)
  def flatMapWithNext[U](f: T ⇒ Input ⇒ Result[U])
    = f(result)(next)
  def filter(f: T ⇒ Boolean): Result[T]
    = if(f(result)) this else zero
  def append[U >: T](alt: ⇒ Result[U])
    = this
}

def consumeFirst: Parser[Elem] = Parser{in ⇒
  Success(first(in), rest(in))(Failure("unknown␣failure", in))
}
```

Bibliography

[1] G. Bracha. Executable grammars in newspeak. *Electr. Notes Theor. Comput. Sci.*, 193:3–18, 2007.

[2] J. Fokker. Functional parsers. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 1995.

[3] B. Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *ICFP*, pages 36–47, 2002.

[4] R. Harper. *Practical Foundations for Programming Languages*. 2008. Working Draft, http://www.cs.cmu.edu/~rwh/plbook/book.pdf.

[5] G. Hutton. Higher-order functions for parsing. *J. Funct. Program.*, 2(3):323–343, 1992.

[6] G. Hutton and E. Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[7] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[8] M. Odersky. *The Scala Language Specification, Version 2.6*. EPFL, Nov. 2007. http://www.scala-lang.org/docu/files/ScalaReference.pdf.

[9] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, L. Spoon, E. Stenman, and M. Zenger. An Overview of the Scala Programming Language (2. edition). Technical report, 2006.

[10] S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 184–207. Springer, 1996.

[11] P. Wadler. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *FPCA*, pages 113–128, 1985.

[12] A. Warth, J. R. Douglass, and T. Millstein. Packrat parsers can support left recursion. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 103–110, New York, NY, USA, 2008. ACM.