Chapter 31

Combinator Parsing

[1]

Ocassionally, you may need to process a small, special-purpose language. For example, you may need to read configuration files for your software, and you want to make them easier to modify by hand than XML. Alternatively, maybe you want to support an input language in your program, such as search terms with boolean operators (computer, find me a movie "with 'space ships' and without 'love stories"'). Whatever the reason, you are going to need a *parser*. You need a way to convert the input language into some data structure your software can process.

Essentially, you have only a few choices. One choice is to roll your own parser (and lexical analyzer). If you are not an expert this is hard. If you are an expert, it is still time consuming.

An alternative choice is to use a parser generator. There exist quite a few of these generators. Some of the better known are Yacc and Bison for parsers written in C and ANTLR for parsers written in Java. You'll probably also need a scanner generator such as Lex, Flex, or JFlex to go with it. This might be the best solution, except for a couple of inconveniences. You need to learn new tools, including their—sometimes obscure—error messages. You also need to figure out how to connect the output of these tools to your program. This might limit the choice of your programming language, and complicate your tool chain.

This chapter presents a third alternative. Instead of using the standalone domain specific language of a parser generator, you will use an *internal do-*

[1] From: Martin Odersky, Lex Spoon, Bill Venners: Programming in Scala, Artima Press, 2008

main specific language, or internal DSL for short. The internal DSL will consist of a library of *parser combinators*—functions and operators defined in Scala that will serve as building blocks for parsers. These building blocks will map one to one to the constructions of a context-free grammar, to make them easy to understand.

This chapter introduces only one language feature that was not explained before: `this` aliasing, in Section 31.6. The chapter does, however, heavily use several other features that were explained in previous chapters. Among others, parameterized types, abstract types, functions as objects, operator overloading, by-name parameters, and implicit conversions all play important roles. The chapter shows how these language elements can be combined in the design of a very high-level library.

The concepts explained in this chapter tend to be a bit more advanced than previous chapters. If you have a good grounding in compiler construction, you'll profit from it reading this chapter, because it will help you put things better in perspective. However, the only prerequisite for understanding this chapter is that you know about regular and context-free grammars. If you don't, the material in this chapter can also safely be skipped.

31.1 Example: Arithmetic expressions

We'll start with an example. Say you want to construct a parser for arithmetic expressions consisting of integer numbers, parentheses, and the binary operators `+`, `-`, `*`, and `/`. The first step is always to write down a grammar for the language to be parsed. Here's the grammar for arithmetic expressions:

$$
\begin{array}{rcl}
\mathit{expr} & ::= & \mathit{term}\ \{\texttt{"+"}\ \mathit{term} \mid \texttt{"-"}\ \mathit{term}\}. \\
\mathit{term} & ::= & \mathit{factor}\ \{\texttt{"*"}\ \mathit{factor} \mid \texttt{"/"}\ \mathit{factor}\}. \\
\mathit{factor} & ::= & \mathit{floatingPointNumber} \mid \texttt{"("}\ \mathit{expr}\ \texttt{")"}.
\end{array}
$$

Here, | denotes alternative productions, and { ... } denotes repetition (zero or more times). And although there's no use of it in this example, [...] denotes an optional occurrence.

This context-free grammar defines formally a language of arithmetic expressions. Every expression (represented by *expr*) is a *term*, which can be followed by a sequence of `+` or `-` operators and further *term*s. A *term* is a *factor*, possibly followed by a sequence of `*` or `/` operators and further *factor*s. A *factor* is either a numeric literal or an expression in parentheses.

Note that the grammar already encodes the relative precedence of operators. For instance, * binds more tightly than +, because a * operation gives a *term*, whereas a + operation gives an *expr*, and *expr*s can contain *term*s but a *term* can contain an *expr* only when the latter is enclosed in parentheses.

Now that you have defined the grammar, what's next? If you use Scala's combinator parsers, you are basically done! You only need to perform some systematic text replacements and wrap the parser in a class, as shown in Listing 31.1:

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

class Arith extends JavaTokenParsers {
  def expr: Parser[Any] = term~rep("+"~term | "-"~term)
  def term: Parser[Any] = factor~rep("*"~factor | "/"~factor)
  def factor: Parser[Any] = floatingPointNumber | "("~expr~")"
}
```

Listing 31.1 · An arithmetic expression parser.

The parsers for arithmetic expressions are contained in a class that inherits from the class `JavaTokenParsers`. This class provides the basic machinery for writing a parser and also provides some primitive parsers that recognize some word classes: identifiers, string literals and numbers. In the example in Listing 31.1 you need only the primitive `floatingPointNumber` parser, which is inherited from this class.

The three definitions in class `Arith` represent the productions for arithmetic expressions. As you can see, they follow very closely the productions of the context-free grammar. In fact, you could generate this part automatically from the context-free grammar, by performing a number of simple text replacements:

1. Every production becomes a method, so you need to prefix it with `def`.

2. The result type of each method is `Parser[Any]`, so you need to change the ::= symbol to "`: Parser[Any] =`". You'll find out later in this chapter what the type `Parser[Any]` signifies, and also how to make it more precise.

3. In the grammar, sequential composition was implicit, but in the program it is expressed by an explicit operator: ~. So you need to insert

a ~ between every two consecutive symbols of a production. In the example in Listing 31.1 we chose not to write any spaces around the ~ operator. That way, the parser code keeps closely to the visual appearance of the grammar—it just replaces spaces by ~ characters.

4. Repetition is expressed `rep( ... )` instead of { ... }. Analogously (though not shown in the example), option is expressed `opt( ... )` instead of [...].

5. The period (.) at the end of each production is omitted—you can, however, write a semicolon (`;`) if you prefer.

That's all there is to it. The resulting class `Arith` defines three parsers, `expr`, `term` and `factor`, which can be used to parse arithmetic expressions and their parts.

31.2 Running your parser

You can exercise your parser with the following small program:

```
object ParseExpr extends Arith {
  def main(args: Array[String]) {
    println("input : "+ args(0))
    println(parseAll(expr, args(0)))
  }
}
```

The `ParseExpr` object defines a `main` method that parses the first command-line argument passed to it. It prints the original input argument, and then prints its parsed version. Parsing is done by the expression

```
parseAll(expr, input)
```

This expression applies the parser, `expr`, to the given `input`. It expects that all of the input matches, *i.e.*, that there are no characters trailing a parsed expression. There's also a method `parse` which allows to parse an input prefix, leaving some remainder unread.

You can run the arithmetic parser with the following command:

```
$ scala ParseExpr "2 * (3 + 7)"
input: 2 * (3 + 7)
[1.12] parsed: ((2~List((*~(((~((3~List())~List((+
~(7~List())))))~)))))~List())
```

The output tells you that the parser successfully analyzed the input string up to position [1.12]. That means the first line and the twelfth column—in other words, the whole input string—was parsed. Disregard for the moment the result after "`parsed:`". It is not very useful, and you will find out later how to get more specific parser results.

You can also try to introduce some input string that is not a legal expression. For instance, you could write one closing parenthesis too many:

```
$ scala ParseExpr "2 * (3 + 7))"
input: 2 * (3 + 7))
[1.12] failure: `-' expected but `)' found

2 * (3 + 7))
           ^
```

Here, the `expr` parser parsed everything until the final closing parenthesis, which does not form part of the arithmetic expression. The `parseAll` method then issued an error message, which said that it expected a `-` operator at the point of the closing parenthesis. You'll find out later in this chapter why it produced this particular error message, and how you can improve it.

31.3 Basic regular expression parsers

The parser for arithmetic expressions made use of another parser, named `floatingPointNumber`. This parser, which was inherited from `Arith`'s superclass, `JavaTokenParsers`, recognizes a floating point number in the format of Java. But what do you do if you need to parse numbers in a format that's a bit different from Java's? In this situation, you can use a *regular expression parser*.

The idea is that you can use any regular expression as a parser. The regular expression parses all strings that it can match. Its result is the parsed string. For instance, the regular expression parser shown in Listing 31.2 describes Java's identifiers:

```
object MyParsers extends RegexParsers {
  val ident: Parser[String] = """[a-zA-Z_]\w*""".r
}
```

Listing 31.2 · A regular expression parser for Java identifiers.

The `MyParsers` object of Listing 31.2 inherits from class `RegexParsers`, whereas `Arith` inherited from `JavaTokenParsers`. Scala's parsing combinators are arranged in a hierarchy of classes, which are all contained in package `scala.util.parsing.combinator`. The top-level class is `Parsers`, which defines a very general parsing framework for all sorts of input. One level below is class `RegexParsers`, which requires that the input is a sequence of characters and provides for regular expression parsing. Even more specialized is class `JavaTokenParsers`, which implements parsers for basic classes of words (or *tokens*) as they are defined in Java.

31.4 Another example: JSON

JSON, the JavaScript Object Notation, is a popular data interchange format. In this section, we'll show you how to write a parser for it. Here's a grammar that describes the syntax of JSON:

value	::=	*obj* | *arr* | *stringLiteral* |
		floatingPointNumber |
		`"null"` | `"true"` | `"false"`.
obj	::=	`"{"` [*members*] `"}"`.
arr	::=	`"["` [*values*] `"]"`.
members	::=	*member* {`","` *member*}.
member	::=	*stringLiteral* `":"` *value*.
values	::=	*value* {`","` *value*}.

A JSON value is an object, array, string, number, or one of the three reserved words `null`, `true`, or `false`. A JSON object is a (possibly empty) sequence of members separated by commas and enclosed in braces. Each member is a string/value pair where the string and the value are separated by a colon. Finally, a JSON array is a sequence of values separated by commas and enclosed in square brackets. As an example, Listing 31.3 contains an address-book formatted as a JSON object.

```
{
  "address book": {
    "name": "John Smith",
    "address": {
      "street": "10 Market Street",
      "city"  : "San Francisco, CA",
      "zip"   : 94111
    },
    "phone numbers": [
      "408 338-4238",
      "408 111-6892"
    ]
  }
}
```

Listing 31.3 · Data in JSON format.

Parsing such data is straightforward when using Scala's parser combinators. The complete parser is shown in Listing 31.4. This parser follows the same structure as the arithmetic expression parser. It is again a straightforward mapping of the productions of the JSON grammar. The productions use one shortcut that simplifies the grammar: The `repsep` combinator parses a (possibly empty) sequence of terms that are separated by a given separator string. For instance, in the example in Listing 31.4, `repsep(member, ",")` parses a comma-separated sequence of `member` terms. Otherwise, the productions in the parser correspond exactly to the productions in the grammar, as was the case for the arithmetic expression parsers.

To try out the JSON parsers, we'll change the framework a bit, so that the parser operates on a file instead of on the command line:

```
import java.io.FileReader

object ParseJSON extends JSON {
  def main(args: Array[String]) {
    val reader = new FileReader(args(0))
    println(parseAll(value, reader))
  }
}
```

The `main` method in this program first creates a `FileReader` object. It then parses the characters returned by that reader according to the `value` production of the JSON grammar. Note that `parseAll` and `parse` exist in overloaded variants: both can take a character sequence or alternatively an input reader as second argument.

If you store the "address book" object shown in Listing 31.3 into a file named `address-book.json` and run the `ParseJSON` program on it, you should get:

```
$ scala ParseJSON address-book.json
[13.4] parsed: (({~List((("address book"~:)~(({~List(((
"name"~:)~"John Smith"), (("address"~:)~(({~List(((
"street"~:)~"10 Market Street"), (("city"~:)~"San Francisco
,CA"), (("zip"~:)~94111)))~})), (("phone numbers"~:)~(([~
List("408 338-4238", "408 111-6892"))~])))~})))~})
```

31.5 Parser output

The `ParseJSON` program successfully parsed the JSON address book. However, the parser output looks strange. It seems to be a sequence composed of bits and pieces of the input glued together with lists and ~ combinations. This output is not very useful. It is less readable for humans than the input,

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

class JSON extends JavaTokenParsers {
  def value : Parser[Any] = obj | arr |
                            stringLiteral |
                            floatingPointNumber |
                            "null" | "true" | "false"
  def obj   : Parser[Any] = "{"~repsep(member, ","~")"}"
  def arr   : Parser[Any] = "["~repsep(value, ","~"]"
  def member: Parser[Any] = stringLiteral~":"~value
}
```

Listing 31.4 · A simple JSON parser.

but it is also too disorganized to be easily analyzable by a computer. It's time to do something about this.

To figure out what to do, you need to know first what the individual parsers in the combinator frameworks return as a result (provided they succeed in parsing the input). Here are the rules:

1. Each parser written as a string (such as: `"{"` or `":"` or `"null"`) returns the parsed string itself.

2. Regular expression parsers such as `"""[a-zA-Z_]\w*""".r` also return the parsed string itself. The same holds for regular expression parsers such as `stringLiteral` or `floatingPointNumber`, which are inherited from class `JavaTokenParsers`.

3. A sequential composition `P~Q` returns the results of both `P` and of `Q`. These results are returned in an instance of a case class that is also written `~`. So if `P` returns `"true"` and `Q` returns `"?"`, then the sequential composition `P~Q` returns `~("true", "?")`, which prints as `(true~?)`.

4. An alternative composition `P | Q` returns the result of either `P` or `Q`, whichever one succeeds.

5. A repetition `rep(P)` or `repsep(P, separator)` returns a list of the results of all runs of `P`.

6. An option `opt(P)` returns an instance of Scala's `Option` type. It returns `Some(R)` if `P` succeeds with result `R` and `None` if `P` fails.

With these rules you can now deduce *why* the parser output appeared as it did in the previous examples. However, the output is still not very convenient. It would be much better to map a JSON object into an internal Scala representation that represents the meaning of the JSON value. A more natural representation would be as follows:

- A JSON object is represented as a Scala map of type `Map[String, Any]`. Every member is represented as a key/value binding in the map.
- A JSON array is represented as a Scala list of type `List[Any]`.
- A JSON string is represented as a Scala `String`.
- A JSON numeric literal is represented as a Scala `Int`.

- The values `true`, `false`, and `null` are represented in as the Scala values with the same names.

To produce to this representation, you need to make use of one more combination form for parsers: `^^`.

The `^^` operator *transforms* the result of a parser. Expressions using this operator have the form `P ^^ f` where `P` is a parser and `f` is a function. `P ^^ f` parses the same sentences as just `P`. Whenever `P` returns with some result `R`, the result of `P ^^ f` is `f(R)`.

As an example, here is a parser that parses a floating point number and converts it to a Scala value of type `Double`.

```
floatingPointNumber ^^ (_.toDouble)
```

And here is a parser that parses the string `"true"` and returns Scala's boolean `true` value:

```
"true" ^^ (x => true)
```

Now for more advanced transformations. Here's a new version of a parser for JSON objects that returns a Scala `Map`:

```
def obj: Parser[Map[String, Any]] = // Can be improved
  "{"~repsep(member, ",")~"}" ^^
    { case "{"~ms~"}" => Map() ++ ms }
```

Remember that the `~` operator produces as its result an instance of a case class with the same name: `~`. Here's a definition of that class—it's an inner class of class `Parsers`:

```
case class ~[+A, +B](x: A, y: B) {
  override def toString = "("+ x +"~"+ y +")"
}
```

The name of the class is intentionally the same as the name of the sequence combinator method, `~`. That way, you can match parser results with patterns that follow the same structure as the parsers themselves. For instance, the pattern `"{"~ms~"}"` matches a result string `"{"` followed by a result variable `ms`, which is followed in turn by a result string `"}"`. This pattern corresponds exactly to what is returned by the parser on the left of the `^^`. In its desugared versions where the `~` operator comes first, the same pattern reads `~(~("{", ms), "}")`, but this is much less legible.

The purpose of the `"{"~ms~"}"` pattern is to strip off the braces so that you can get at the list of members resulting from the `repsep(member, ",")` parser. In cases like these there is also an alternative that avoids producing unnecessary parser results that are immediately discarded by the pattern match. The alternative makes use of the ~> and <~ parser combinators. Both express sequential composition like ~, but ~> keeps only the result of its right operand, whereas <~ keeps only the result of its left operand. Using these combinators, the JSON object parser can be expressed more succinctly:

```
def obj: Parser[Map[String, Any]] =
  "{"~> repsep(member, ",") <~"}" ^^ (Map() ++ _)
```

Listing 31.5 shows a full JSON parser that returns meaningful results. If you run this parser on the `address-book.json` file, you will get the following result (after adding some newlines and indentation):

```
$ scala JSON1Test address-book.json
[14.1] parsed: Map(
  address book -> Map(
    name -> John Smith,
    address -> Map(
      street -> 10 Market Street,
      city -> San Francisco, CA,
      zip -> 94111),
    phone numbers -> List(408 338-4238, 408 111-6892)
  )
)
```

This is all you need to know in order to get started writing your own parsers. As an aide to memory, Table 31.1 lists the parser combinators that were discussed so far.

Symbolic versus alphanumeric names

Many of the parser combinators in Table 31.1 use symbolic names. This has both advantages and disadvantages. On the minus side, symbolic names take time to learn. Users who are unfamiliar with Scala's combinator parsing libraries are probably mystified what ~, ~>, or ^^ mean. On the plus side, symbolic names are short, and can be chosen to have the "right" precedences and associativities. For instance, the parser combinators ~, ^^, and | are

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

class JSON1 extends JavaTokenParsers {

  def obj: Parser[Map[String, Any]] =
    "{"~> repsep(member, ",") <~"}" ^^ (Map() ++ _)

  def arr: Parser[List[Any]] =
    "["~> repsep(value, ",") <~"]"

  def member: Parser[(String, Any)] =
    stringLiteral~":"~value ^^
      { case name~":"~value => (name, value) }

  def value: Parser[Any] = (
    obj
  | arr
  | stringLiteral
  | floatingPointNumber ^^ (_.toInt)
  | "null"  ^^ (x => null)
  | "true"  ^^ (x => true)
  | "false" ^^ (x => false)
  )
}
```

Listing 31.5 · A full JSON parser that returns meaningful results.

Table 31.1 · Summary of parser combinators

`"..."`	literal
`"...".r`	regular expression
`P~Q`	sequential composition
`P <~ Q`, `P ~> Q`	sequential composition; keep left/right only
`P | Q`	alternative
`opt(P)`	option
`rep(P)`	repetition
`repsep(P, Q)`	interleaved repetition
`P ^^ f`	result conversion

Turning off semicolon inference

Note that the body of the `value` parser in Listing 31.5 is enclosed in parentheses. This is a little trick to disable semicolon inference in parser expressions. You saw in Section 4.2 that Scala assumes there's a semicolon between any two lines that can be separate statements syntactically, unless the first line ends in an infix operator, or the two lines are enclosed in parentheses or square brackets. Now, you could have written the `|` operator at the end of the each alternative instead of at the beginning of the following one, like this:

```
def value: Parser[Any] =
  obj |
  arr |
  stringLiteral |
  ...
```

In that case, no parentheses around the body of the `value` parser would have been required. However, some people prefer to see the `|` operator at the beginning of the second alternative rather than at the end of the first. Normally, this would lead to an unwanted semicolon between the two lines, like this:

```
  obj;  // semicolon implicitly inserted
| arr
```

The semicolon changes the structure of the code, causing it to fail compilation. Putting the whole expression in parentheses avoids the semicolon and makes the code compile correctly.

chosen intentionally in decreasing order of precedence. A typical grammar production is composed of alternatives that have a parsing part and a transformation part. The parsing part usually contains several sequential items separated by `~` operators. With the chosen precedences of `~`, `^^`, and `|` you can write such a grammar production without needing any parentheses.

Furthermore, symbolic operators take less visual real estate than alphabetic ones. That's important for a parser because it lets you concentrate on

the grammar at hand, instead of the combinators themselves. To see the difference, imagine for a moment that sequential composition (~) was called andThen and alternative (|) was called orElse. The arithmetic expression parsers in Listing 31.1 on page 615 would look as follows:

```
class ArithHypothetical extends JavaTokenParsers {
  def expr: Parser[Any]   =
    term andThen rep(("+" andThen term) orElse
                     ("-" andThen term))
  def term: Parser[Any]   =
    factor andThen rep(("*" andThen factor) orElse
                       ("/" andthen factor))
  def factor: Parser[Any] =
    floatingPointNumber orElse
    ("(" andThen expr andThen ")")
}
```

You notice that the code becomes much longer, and that it's hard to "see" the grammar among all those operators and parentheses. On the other hand, somebody new to combinator parsing could probably figure out better what the code is supposed to do.

31.6 Implementing combinator parsers

The previous sections have shown that Scala's combinator parsers provide a convenient means for constructing your own parsers. Since they are nothing more than a Scala library, they fit seamlessly into your Scala programs. So it's very easy to combine a parser with some code that processes the results it delivers, or to rig a parser so that it takes its input from some specific source (say, a file, a string, or a character array).

How is this achieved? In the rest of this chapter you'll take a look "under the hood" of the combinator parser library. You'll see what a parser is, and how the primitive parsers and parser combinators encountered in previous sections are implemented. You can safely skip these parts if all you want to do is write some simple combinator parsers. On the other hand, reading the rest of this chapter should give you a deeper understanding of combinator parsers in particular, and of the design principles of a combinator domain-specific language in general.

Choosing between symbolic and alphabetic names

As guidelines for choosing between symbolic and alphabetic names we recommend the following:

- Use symbolic names in cases where they already have a universally established meaning. For instance, nobody would recommend to write `add` instead of `+` for numeric addition.
- Otherwise, give preference to alphabetic names if you want your code to be understandable to casual readers.
- You can still choose symbolic names for domain-specific libraries, if this gives clear advantages in legibility and you do not expect anyway that a casual reader without a firm grounding in the domain would be able understand the code immediately.

In the case of parser combinators we are looking at a highly domain-specific language, which casual readers may have trouble understanding even with alphabetic names. Furthermore, symbolic names give clear advantages in legibility for the expert. So we believe their use is warranted in this application.

The core of Scala's combinator parsing framework is contained in the class `scala.util.parsing.combinator.Parsers`. This class defines the `Parser` type as well as all fundamental combinators. Except where stated explicitly otherwise, the definitions explained in the following two subsections all reside in this class. That is they are assumed to be contained in a class definition that starts as follows:

```
package scala.util.parsing.combinator
class Parsers {
  ... // code goes here unless otherwise stated
}
```

A `Parser` is in essence just a function from some input type to a parse result. As a first approximation, the type could be written as follows:

```
type Parser[T] = Input => ParseResult[T]
```

Parser input

Sometimes, a parser reads a stream of tokens instead of a raw sequence of characters. A separate lexical analyzer is then used to convert a stream of raw characters into a stream of tokens. The type of parser inputs is defined as follows:

```
type Input = Reader[Elem]
```

The class `Reader` comes from the package `scala.util.parsing.input`. It is similar to a `Stream`, but also keeps track of the positions of all the elements it reads. The type `Elem` represents individual input elements. It is an abstract type member of the `Parsers` class:

```
type Elem
```

This means that subclasses of `Parsers` need to instantiate class `Elem` to the type of input elements that are being parsed. For instance, `RegexParsers` and `JavaTokenParsers` fix `Elem` to be equal to `Char`. But it would also be possible to set `Elem` to some other type, such as the type of tokens returned from a separate lexer.

Parser results

A parser might either succeed or fail on some given input. Consequently class `ParseResult` has two subclasses for representing success and failure:

```
sealed abstract class ParseResult[+T]
case class Success[T](result: T, in: Input)
  extends ParseResult[T]
case class Failure(msg: String, in: Input)
  extends ParseResult[Nothing]
```

The `Success` case carries the result returned from the parser in its `result` parameter. The type of parser results is arbitrary; that's why `ParseResult`, `Success`, and `Parser` are all parameterized with a type parameter `T`. The type parameter represents the kinds of results returned by a given parser. `Success` also takes a second parameter, `in`, which refers to the input immediately following the part that the parser consumed. This field is needed for chaining parsers, so that one parser can operate after another. Note that this is a purely functional approach to parsing. Input is not read as a side effect,

but it is kept in a stream. A parser analyzes some part of the input stream, and then returns the remaining part in its result.

The other subclass of `ParseResult` is `Failure`. This class takes as a parameter a message that describes why the parser failed. Like `Success`, `Failure` also takes the remaining input stream as a second parameter. This is needed not for chaining (the parser won't continue after a failure), but to position the error message at the correct place in the input stream.

Note that parse results are defined to be covariant in the type parameter `T`. That is, a parser returning `String`s as result, say, is compatible with a parser returning `AnyRef`s.

The `Parser` class

The previous characterization of parsers as functions from inputs to parse results was a bit oversimplified. The previous examples showed that parsers also implement *methods* such as `~` for sequential composition of two parsers and `|` for their alternative composition. So `Parser` is in reality a class that inherits from the function type `Input => ParseResult[T]` and additionally defines these methods:

```
abstract class Parser[+T] extends (Input => ParseResult[T])
{ p =>
  // An unspecified method that defines
  // the behavior of this parser.
  def apply(in: Input): ParseResult[T]

  def ~ ...
  def | ...
  ...
}
```

Since parsers are (*i.e.*, inherit from) functions, they need to define an `apply` method. You see an abstract `apply` method in class `Parser`, but this is just for documentation, as the same method is in any case inherited from the parent type `Input => ParseResult[T]` (recall that this type is an abbreviation for `scala.Function1[Input, ParseResult[T]]`). The `apply` method still needs to be implemented in the individual parsers that inherit from the abstract `Parser` class. These parsers will be discussed after the following section on `this` aliasing.

Aliasing `this`

The body of the `Parser` class starts with a curious expression:

```
abstract class Parser[+T] extends ... { p =>
```

A clause such as `id =>` immediately after the opening brace of a class template defines the identifier `id` as an alias for `this` in the class. It's as if you had written:

```
val id = this
```

in the class body, except that the Scala compiler knows that `id` is an alias for `this`. For instance, you could access an object-private member `m` of the class using either `id.m` or `this.m`; the two are completely equivalent. The first expression would not compile if `id` were just defined as a `val` with `this` as its right hand side, because in that case the Scala compiler would treat `id` as a normal identifier.

You saw syntax like this in Section 27.4, where it was used to give a self type to a trait. Aliasing can also be a good abbreviation when you need to access the `this` of an outer class. Here's an example:

```
class Outer { outer =>
  class Inner {
    println(Outer.this eq outer) // prints: true
  }
}
```

The example defines two nested classes, `Outer` and `Inner`. Inside `Inner` the `this` value of the `Outer` class is referred to twice, using different expressions. The first expression shows the Java way of doing things: You can prefix the reserved word `this` with the name of an outer class and a period; such an expression then refers to the `this` of the outer class. The second expression shows the alternative that Scala gives you. By introducing an alias named `outer` for `this` in class `Outer`, you can refer to this alias directly also in inner classes. The Scala way is more concise, and can also improve clarity, if you choose the name of the alias well. You'll see examples of this in pages 631 and 632.

Single-token parsers

Class `Parsers` defines a generic parser `elem` that can be used to parse any single token:

```
def elem(kind: String, p: Elem => Boolean) =
  new Parser[Elem] {
    def apply(in: Input) =
      if (p(in.first)) Success(in.first, in.rest)
      else Failure(kind +" expected", in)
  }
```

This parser takes two parameters: a `kind` string describing what kind of token should be parsed and a predicate `p` on `Elem`s, which indicates whether an element fits the class of tokens to be parsed.

When applying the parser `elem(kind, p)` to some input `in`, the first element of the input stream is tested with predicate `p`. If `p` returns `true`, the parser succeeds. Its result is the element itself, and its remaining input is the input stream starting just after the element that was parsed. On the other hand, if `p` returns `false`, the parser fails with an error message that indicates what kind of token was expected.

Sequential composition

The `elem` parser only consumes a single element. To parse more interesting phrases, you can string parsers together with the sequential composition operator `~`. As you have seen before, `P~Q` is a parser that applies first the `P` parser to a given input string. Then, if `P` succeeds, the `Q` parser is applied to the input that's left after `P` has done its job.

The `~` combinator is implemented as a method in class `Parser`. Its definition is shown in Listing 31.6. The method is a member of the `Parser` class. Inside this class, `p` is specified by the "`p =>`" part as an alias of `this`, so `p` designates the left operand (or: receiver) of `~`. Its right operand is represented by parameter `q`. Now, if `p~q` is run on some input `in`, first `p` is run on `in` and the result is analyzed in a pattern match. If `p` succeeds, `q` is run on the remaining input `in1`. If `q` also succeeds, the parser as a whole succeeds. Its result is a `~` object containing both the result of `p` (*i.e.*, `x`) and the result of `q` (*i.e.*, `y`). On the other hand, if either `p` or `q` fails the result of `p~q` is the `Failure` object returned by `p` or `q`.

```
abstract class Parser[+T] ... { p =>
  ...
  def ~ [U](q: => Parser[U]) = new Parser[T~U] {
    def apply(in: Input) = p(in) match {
      case Success(x, in1) =>
        q(in1) match {
          case Success(y, in2) => Success(new ~(x, y), in2)
          case failure => failure
        }
      case failure => failure
    }
  }
```

Listing 31.6 · The ~ combinator method.

The result type of ~ is a parser that returns an instance of the case class ~ with elements of types T and U. The type expression T~U is just a more legible shorthand for the parameterized type ~[T, U]. Generally, Scala always interprets a binary type operation such as A op B, as the parameterized type op[A, B]. This is analogous to the situation for patterns, where an binary pattern P op Q is also interpreted as an application, *i.e.*, op(P, Q).

The other two sequential composition operators, <~ and ~>, could be defined just like ~, only with some small adjustment in how the result is computed. A more elegant technique, though, is to define them in terms of ~ as follows:

```
def <~ [U](q: => Parser[U]): Parser[T] =
  (p~q) ^^ { case x~y => x }
def ~> [U](q: => Parser[U]): Parser[U] =
  (p~q) ^^ { case x~y => y }
```

Alternative composition

An alternative composition P | Q applies either P or Q to a given input. It first tries P. If P succeeds, the whole parser succeeds with the result of P. Otherwise, Q is tried *on the same input* as P. The result of Q is then the result of the whole parser.

Here is a definition of | as a method of class Parser:

```
def | (q: => Parser[T]) = new Parser[T] {
  def apply(in: Input) = p(in) match {
    case s1 @ Success(_, _) => s1
    case failure => q(in)
  }
}
```

Note that if `P` and `Q` both fail, then the failure message is determined by `Q`. This subtle choice is discussed later, in Section 31.9.

Dealing with recursion

Note that the `q` parameter in methods `~` and `|` is by-name—its type is preceded by `=>`. This means that the actual parser argument will be evaluated only when `q` is needed, which should only be the case after `p` has run. This makes it possible to write recursive parsers like the following one which parses a number enclosed by arbitrarily many parentheses:

```
def parens = floatingPointNumber | "("~parens~")"
```

If `|` and `~` took by-value parameters, this definition would immediately cause a stack overflow without reading anything, because the value of `parens` occurs in the middle of its right-hand side.

Result conversion

The last method of class `Parser` converts a parser's result. The parser `P ^^ f` succeeds exactly when `P` succeeds. In that case it returns `P`'s result converted using the function `f`. Here is the implementation of this method:

```
  def ^^ [U](f: T => U): Parser[U] = new Parser[U] {
    def apply(in: Input) = p(in) match {
      case Success(x, in1) => Success(f(x), in1)
      case failure => failure
    }
  }
} // end Parser
```

Parsers that don't read any input

There are also two parsers that do not consume any input: `success` and `failure`. The parser `success(result)` always succeeds with the given `result`. The parser `failure(msg)` always fails with error message `msg`. Both are implemented as methods in class `Parsers`, the outer class that also contains class `Parser`:

```
def success[T](v: T) = new Parser[T] {
  def apply(in: Input) = Success(v, in)
}
def failure(msg: String) = new Parser[Nothing] {
  def apply(in: Input) = Failure(msg, in)
}
```

Option and repetition

Also defined in class `Parsers` are the option and repetition combinators `opt`, `rep`, and `repsep`. They are all implemented in terms of sequential composition, alternative, and result conversion:

```
  def opt[T](p: => Parser[T]): Parser[Option[T]] = (
    p ^^ Some(_)
  | success(None)
  )

  def rep[T](p: Parser[T]): Parser[List[T]] = (
    p~rep(p) ^^ { case x~xs => x :: xs }
  | success(List())
  )

  def repsep[T, U](p: Parser[T],
      q: Parser[U]): Parser[List[T]] = (
    p~rep(q~> p) ^^ { case r~rs => r :: rs }
  | success(List())
  )
} // end Parsers
```

31.7 String literals and regular expressions

The parsers you saw so far made use of string literals and regular expressions to parse single words. The support for these comes from `RegexParsers`, a subclass of `Parsers`:

```
class RegexParsers extends Parsers {
```

This class is more specialized than class `Parsers` in that it only works for inputs that are sequences of characters:

```
type Elem = Char
```

It defines two methods, `literal` and `regex`, with the following signatures:

```
implicit def literal(s: String): Parser[String] = ...
implicit def regex(r: Regex): Parser[String] = ...
```

Note that both methods have an `implicit` modifier, so they are automatically applied whenever a `String` or `Regex` is given but a `Parser` is expected. That's why you can write string literals and regular expressions directly in a grammar, without having to wrap them with one of these methods. For instance, the parser `"("~expr~")"` will be automatically expanded to `literal("(")~expr~literal(")")`.

The `RegexParsers` class also takes care of handling white space between symbols. To do this, it calls a method named `handleWhiteSpace` before running a `literal` or `regex` parser. The `handleWhiteSpace` method skips the longest input sequence that conforms to the `whiteSpace` regular expression, which is defined by default as follows:

```
  protected val whiteSpace = """\s+""".r
} // end RegexParsers
```

If you prefer a different treatment of white space, you can override the `whiteSpace` `val`. For instance, if you want white space not to be skipped at all, you can override `whiteSpace` with the empty regular expression:

```
object MyParsers extends RegexParsers {
  override val whiteSpace = "".r
  ...
}
```

31.8 Lexing and parsing

The task of syntax analysis is often split into two phases. The *lexer* phase recognizes individual words in the input and classifies them into some *token* classes. This phase is also called *lexical analysis*. This is followed by a *syntactical analysis* phase that analyzes sequences of tokens. Syntactical analysis is also sometimes just called parsing, even though this is slightly imprecise, as lexical analysis can also be regarded as a parsing problem.

The `Parsers` class as described in the previous section can be used for either phase, because its input elements are of the abstract type `Elem`. For lexical analysis, `Elem` would be instantiated to `Char`, meaning the individual characters that make up a word are being parsed. The syntactical analyzer would in turn instantiate `Elem` to the type of token returned by the lexer.

Scala's parsing combinators provide several utility classes for lexical and syntactic analysis. These are contained in two sub-packages, one for each kind of analysis:

```
scala.util.parsing.combinator.lexical
scala.util.parsing.combinator.syntactical
```

If you want to split your parser into a separate lexer and syntactical analyzer, you should consult the ScalaDocs for these packages. But for simple parsers, the regular expression based approach shown in previously this chapter is usually sufficient.

31.9 Error reporting

There's one final topic that was not covered yet: how does the parser issue an error message? Error reporting for parsers is somewhat of a black art. One problem is that when a parser rejects some input, it generally has encountered many different failures. Each alternative parse must have failed, and recursively so at each choice point. Which of the usually numerous failures should be emitted as error message to the user?

Scala's parsing library implements a simple heuristic: among all failures, the one that occurred at the latest position in the input is chosen. In other words, the parser picks the longest prefix that is still valid and issues an error message that describes why parsing the prefix could not be continued further. If there are several failure points at that latest position, the one that was visited last is chosen.

For instance, consider running the JSON parser on a faulty address book which starts with the line:

```
{ "name": John,
```

The longest legal prefix of this phrase is `{ "name": `. So the JSON parser will flag the word `John` as an error. The JSON parser expects a value at this point, but `John` is an identifier, which does not count as a value (presumably, the author of the document had forgotten to enclose the name in quotation marks). The error message issued by the parser for this document is:

```
[1.13] failure: "false" expected but identifier John found

  { "name": John,
            ^
```

The part that "false" was expected comes from the fact that `"false"` is the last alternative of the production for `value` in the JSON grammar. So this was the last failure at this point. Users who know the JSON grammar in detail can reconstruct the error message, but for non-experts this error message is probably surprising and can also be quite misleading.

A better error message can be engineered by adding a "catch all" failure point as last alternative of a `value` production:

```
def value: Parser[Any] =
  obj | arr | stringLit | floatingPointNumber | "null" | "true" |
  "false" | failure("illegal start of value")
```

This addition does not change the set of inputs that are accepted as valid documents. What it does is improve the error messages, because now it will be the explicitly added `failure` that comes as last alternative and therefore gets reported:

```
[1.13] failure: illegal start of value

  { "name": John,
            ^
```

The implementation of the "latest possible" scheme of error reporting uses a field named `lastFailure`: in class `Parsers` to mark the failure that occurred at the latest position in the input.

```
var lastFailure: Option[Failure] = None
```

The field is initialized to None. It is updated in the constructor of the Failure class:

```
case class Failure(msg: String, in: Input)
extends ParseResult[Nothing] {
  if (lastFailure.isDefined &&
      lastFailure.get.in.pos <= in.pos)
    lastFailure = Some(this)
}
```

The field is read by the phrase method, which emits the final error message if the parser failed. Here is the implementation of phrase in class Parsers:

```
def phrase[T](p: Parser[T]) = new Parser[T] {
  lastFailure = None
  def apply(in: Input) = p(in) match {
    case s @ Success(out, in1) =>
      if (in1.atEnd) s
      else Failure("end of input expected", in1)
    case f : Failure =>
      lastFailure
  }
}
```

The phrase method runs its argument parser p. If p succeeds with a completely consumed input, the success result of p is returned. If p succeeds but the input is not read completely, a failure with message "end of input expected" is returned. If p fails, the failure or error stored in lastFailure is returned. Note that the treatment of lastFailure is non-functional; it is updated as a side effect by the constructor of Failure and by the phrase method itself. A functional version of the same scheme would be possible, but it would require threading the lastFailure value though every parser result, no matter whether this result is a Success or a Failure.

Putting it all together

The last two methods in class Parsers run a given parser on an input. They are implemented as follows:

```
def parse[T](p: Parser[T], in: Input): ParseResult[T] =
```

```
    p(in)

  def parseAll[T](p: Parser[T], in: Input): ParseResult[T] =
    parse(phrase(p), in)
```

The `parse` method parses some prefix of the given input `in` with the given parser `p`. To do that, it simply applies the parser to the input. The `parseAll` method parses all of the given input `in` with the given parser `p`. To do that it runs `phrase(p)` on the input.

31.10 Backtracking versus LL(1)

The parser combinators employ *backtracking* to choose between different parsers in an alternative. In an expression `P | Q`, if `P` fails, then `Q` is run on the same input as `P`. This happens even if `P` has parsed some tokens before failing. In this case the same tokens will be parsed again by `Q`.

Backtracking imposes only a few restrictions on how to formulate a grammar so that it can be parsed. Essentially, you just need to avoid left-recursive productions. A production such as:

$$expr \quad ::= \quad expr \ \texttt{"+"} \ term \ \mid \ term.$$

will always fail because `expr` immediately calls itself and thus never progresses any further.[2] On the other hand, backtracking is potentially costly because the same input can be parsed several times. Consider for instance the production:

$$expr \quad ::= \quad term \ \texttt{"+"} \ expr \ \mid \ term.$$

What happens if the `expr` parser is applied to an input such as `(1 + 2) * 3` which constitutes a legal term? The first alternative would be tried, and would fail when matching the "+" sign. Then the second alternative would be tried on the same term and this would succeed. In the end the term ended up being parsed twice.

It is often possible to modify the grammar so that backtracking can be avoided. For instance, in the case of arithmetic expressions, either one of the following productions would work:

[2] There are ways to avoid stack overflows even in the presence of left-recursion, but this requires a more refined parsing combinator framework, which to date has not been implemented.

$$\begin{array}{lcl} expr & ::= & term\ [\texttt{"+"}\ expr]. \\ expr & ::= & term\ \{\texttt{"+"}\ term\}. \end{array}$$

Many languages admit so-called "LL(1)" grammars.[3] When a combinator parser is formed from such a grammar, it will never backtrack, *i.e.*, the input position will never be reset to an earlier value. For instance, the grammars for arithmetic expressions and JSON terms earlier in this chapter are both LL(1), so the backtracking capabilities of the parser combinator framework are never exercised for inputs from these languages.

The combinator parsing framework allows you to express the expectation that a grammar is LL(1) explicitly, using a new operator `~!`. This operator is like sequential composition `~` but it will never backtrack to "un-read" input elements that have already been parsed. Using this operator, the productions in the arithmetic expression parser could alternatively be written as follows:

```
def expr  : Parser[Any] =
  term ~! rep("+" ~! term | "-" ~! term)
def term  : Parser[Any] =
  factor ~! rep("*" ~! factor | "/" ~! factor)
def factor: Parser[Any] =
  "(" ~! expr ~! ")" | floatingPointNumber
```

One advantage of an LL(1) parser is that it can use a simpler input technique. Input can be read sequentially, and input elements can be discarded once they are read. That's another reason why LL(1) parsers are usually more efficient than backtracking parsers.

31.11 Conclusion

You have now seen all the essential elements of Scala's combinator parsing framework. It's surprisingly little code for something that's genuinely useful. With the framework you can construct parsers for a large class of context-free grammars. The framework lets you get started quickly, but it is also customizable to new kinds of grammars and input methods. Being a Scala library, it integrates seamlessly with the rest of the language. So it's easy to integrate a combinator parser in a larger Scala program.

[3] Aho, *et. al.*, *Compilers: Principles, Techniques, and Tools*. [**?**]

One downside of combinator parsers is that they are not very efficient, at least not when compared with parsers generated from special purpose tools such as Yacc or Bison. There are two reasons for this. First, the backtracking method used by combinator parsing is itself not very efficient. Depending on the grammar and the parse input, it might yield an exponential slow-down due to repeated backtracking. This can be fixed by making the grammar LL(1) and by using the committed sequential composition operator, `~!`.

The second problem affecting the performance of combinator parsers is that they mix parser construction and input analysis in the same set of operations. In effect, a parser is generated anew for each input that's parsed.

This problem can be overcome, but it requires a different implementation of the parser combinator framework. In an optimizing framework, a parser would no longer be represented as a function from inputs to parse results. Instead, it would be represented as a tree, where every construction step was represented as a case class. For instance, sequential composition could be represented by a case class `Seq`, alternative by `Alt`, and so on. The "outermost" parser method, `phrase`, could then take this symbolic representation of a parser and convert it to highly efficient parsing tables, using standard parser generator algorithms.

What's nice about all this is that from a user perspective nothing changes compared to plain combinator parsers. Users still write parsers in terms of `ident`, `floatingPointNumber`, `~`, `|`, and so on. They need not be aware that these methods generate a symbolic representation of a parser instead of a parser function. Since the `phrase` combinator converts these representations into real parsers, everything works as before.

The advantage of this scheme with respect to performance is two-fold. First, you can now factor out parser construction from input analysis. If you were to write:

```
val jsonParser = phrase(value)
```

and then apply `jsonParser` to several different inputs, the `jsonParser` would be constructed only once, not every time an input is read.

Second, the parser generation can use efficient parsing algorithms such as LALR(1).[4] These algorithms usually lead to much faster parsers than parsers that operate with backtracking.

[4] Aho, *et. al.*, *Compilers: Principles, Techniques, and Tools*. [**?**]

At present, such an optimizing parser generator has not yet been written for Scala. But it would be perfectly possible to do so. If someone contributes such a generator, it will be easy to integrate into the standard Scala library. Even postulating that such a generator will exist at some point in the future, however, there are reasons for keeping the current parser combinator framework around. It is much easier to understand and to adapt than a parser generator, and the difference in speed would often not matter in practice, unless you want to parse very large inputs.