Combinator Parsing

– Draft –

Small languages abound on the internet. There are many situations where you need to whip up a processor for such a language. However, often you are stopped in your tracks by the problem of *parsing* sentences in the language you want to process. Essentially, you have two 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 to do this.

The alternative choice is to use a parser generator such Yacc, Bison, or AntLR, or JavaCC. You'll probably also need a scanner generator such as Lex to go with it. This might be the best solution, except for a couple of inconveniences: You need to learn a new tool, including its – sometimes obscure – error messages. You also need to figure out how to connect the output of this tool 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 stand-alone domain specific language of a parser generator you will use an *embedded DSL* for the same task. The embedded DSL consists of a library of *parser combinators*. These are functions and operators defined in Scala that are bullding blocks for parsers. The building blocks follow one by one the constructions of a context-free grammar, so they are very easy to understand.

1 Example: Arithmetic Expressions

Here's an example: Let's 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. For arithmetic expressions, this grammar reads as follows:

```
expr     ::=  term {'+' term | '-' term}.
term     ::=  factor {'*' factor | '/' factor}.
factor   ::=  numericLit | '(' expr ')'.
```

Here, "|" denotes alternative productions. `{...}` denotes repetition (zero or more times) whereas `[...]` denotes an optional occurrence.

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 follows:

```
import scala.util.parsing.combinator1.syntactical.StandardTokenParsers
class Arith extends StandardTokenParsers {
  lexical.delimiters ++= List("(", ")", "+", "-", "*", "/")
  def expr  : Parser[Any] = term ~ rep("+" ~ term | "-" ~ term)
  def term  : Parser[Any] = factor ~ rep("*" ~ factor | "/" ~ factor)
  def factor: Parser[Any] = "(" ~ expr ~ ")" | numericLit
}
```

The parser for arithmetic expressions is a class which inherits from

```
scala.util.parsing.combinator1.syntactical.StandardTokenParsers.
```

The latter is a class which provides the machinery to write a standard parser. The class builds on a standard lexer which recognizes Java-like tokens consisting of strings, integers, and identifiers. The lexer skips over whitespace and Java-like comments. Both multi-line comments `/* ... */` and single line comments `// ...` are supported. The lexer still needs to be configured with a set of *delimiters*. These are tokens consisting of special symbols that the lexer should recognize. In the case of arithmetic expressions, the delimiters are "`(`", "`)`", "`+`", "`-`", "`*`", and "`/`". They are installed by the line:

```
lexical.delimiters += ("(", ")", "+", "-", "*", "/")
```

Here, `lexical` refers to the lexer component inherited from class `StandardTokenParsers` and `delimiters` is a mutable set in this component. The "`+=`" operation enters the desired delimiters into the set.

The next three lines 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 "produces" sign `::=` to `: Parser[Any] =` . You'll find out below 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.
4. Repetition is expressed `rep(...)` instead of `{...}`. Analogously (not shown in the example), option is expressed `opt(...)` instead of `[...]`.
5. The point "." at the end of each production is omitted – you can however write a semicolon ";" if you like.

That's it! You have a parser for arithmetic expressions. As you can see, the combinator parsing framework gives you a fast path to construct your own parsers. In part this is due to the fact that a lot of functionality is "pre-canned" in the `StandardTokenParsers` class. But the parsing framework as a whole is also easy to adapt to other scenarious. For instance, it is

quite bossible to configure the framework to use a different lexer (including one you write). In fact, you will find out that the lexer itself can be written with the same combinator parsers that underlie the parser for arithmetic expressions.

2 Running Your Parser

You can test your parser with the following a small program:

```
object ArithTest extends Arith {
  def main(args: Array[String]) {
    val tokens = new lexical.Scanner(args(0))
    println("input: "+args(0))
    println(phrase(expr)(tokens))
  }
}
```

The `ArithTest` object defines a `main` method which parses the first command line argument that's passed to it. It first creates a `Scanner` object that reads the first input argument and converts it to a token sequence named `tokens`. It then prints the original input argument, and finally prints its parsed version. Parsing is done by the expression

```
phrase(expr)(tokens)
```

This expression applies the parser `phrase(expr)` to the token sequence `tokens`. The `phrase` method is a special parser. It takes another parser as argument, applies this parser to an input sequence, and at the same time makes sure that after parsing the input sequence is completely read. So `phrase(expr)` is like `expr`, except that `expr` is can parse parts of input sequences, whereas `phrase(expr)` succeeds only if the input sequence is parsed from beginning to end.

You can run the arithmetic parser with the following command:

```
scala ArithTest "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, or, otherwise put, all the input string was parsed. Ignore 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 which is not a legal expression. For instance, you could write one closing parenthesis too many:

```
java ArithTest "2 * (3 + 7))"
input: 2 * (3 + 7))
[1.12] failure: end of input expected

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

Here, the `expr` parser parsed everything until the final closing parenthesis, which does not form part of the arithmetic expression. The `phrase` parser then issued an error message which said that it expected the input to end at the point of the closing parenethesis.

3 Another Example: JSON

Let's try another example. JSON, the JavaScript Object Notation is a popular data interchange format. You'll now find out how to write a parser for it. Here is the syntax of JSON:

```
value = obj | arr | stringLit | numericLit | "null" | "true" | "false"
obj = "{" [members] "}"
arr = "[" [values] "]"
members = member {"," member}
member  = stringLit ":" value
values  = value {"," value}
```

A JSON value is an object, or an array, or a string, or a number, or one of the three reserved words `null`, `true`, or `false`. A JSON object is a (possible 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 brackets.

Here is an example of 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"
    ]
  }
}
```

Parsing JSON data is straightforward when using Scala's parser combinators. Here is the complete parser:

```
import scala.util.parsing.combinator1.syntactical.StandardTokenParsers
class JSON extends StandardTokenParsers {
  lexical.delimiters += ("{", "}", "[", "]", ":", ",")
  lexical.reserved += ("null", "true", "false")

  def value : Parser[Any] = obj | arr | stringLit | numericLit |
                            "null" | "true" | "false"
  def obj   : Parser[Any] = "{" ~ repsep(member, ",") ~ "}"
```

```
  def arr   : Parser[Any] = "[" ~ repsep(value, ",") ~ "]"
  def member: Parser[Any] = stringLit ~ ":" ~ value
}
```

This parser follows the same structure as the arithmetic expression parser. The delimiters of `JSON` are `"{", "}", "[", "]", ":", ","`. There are also some *reserved words*: `null`, `true`, `false`. Reserved words are tokens which follow the syntax of identifiers, but which are reserved. Reserved words are communicated to the lexer by entering them into its `reserved` table:

```
lexical.reserved += ("null", "true", "false")
```

The rest of the parser is made up of the productions of the JSON grammar. The productions use one shortcut which simplifies the grammar: The `repsep` combinator parses a (possibly empty) sequence of terms which are separated by a given separator string. For instance, in the example above, `repsep(member, ",")` parses a comma-separated sequence of `member` terms. Otherwise, the productions in the parser correspond exactly to the productions in the grammer, just like it was the case for the arithmetic expression parsers.

To test the JSON parsers, let's change the framework a bit, so that the parser operates on a file instead of on the command line:

```
import scala.util.parsing.input.StreamReader
object JSONTest extends JSON {
  def main(args: Array[String]) {
    val reader = StreamReader(new java.io.FileReader(args(0)))
    val tokens = new lexical.Scanner(reader)
    println(phrase(value)(tokens))
  }
}
```

The `main` method in this program first creates a `StreamReader` object. This object represents an input stream of characters with positions; for every character that's read one can query its line and column numbers (both lines and columns start at 1). It then creates a `Scanner` over this stream reader. Finally the tokens returned from the scanner are parsed; they need to conform to the `value` production of the JSON grammar. If you store the "address book" object above into a file named `address-book.json` and run the test program on it you should get:

```
java JSONTest address-book.json
[14.1] 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)) ~ ])))))
~ }))))) ~ })
```

4 Parser Output

The test run above succeeded; the JSON address book was succesfully parsed. 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 parser output is not very useful. It is certainly less readable for humans than the input, 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 indvidual 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. Each of the single-token parsers `stringLit`, `numericLit`, and `ident` also returns the parsed string itself.
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 which 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 compositon `P | Q` returns the result of either `P` and `Q` (whichever one succeeds).
5. A repetition `rep(P)` or `repsep(P, separator)` returns the results of all runs of `P` as elements of a list.
6. An option `opt(P)` returns an instance of Scala's `Option` type. It returns the `Some(R)` if `P` succeeds with result `R` and `None` if `P` fails.

With these rules you can now figure *why* the parser output was as shown in the example above. 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 representation which is natural 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, we need to make use of two more combination forms for parsers, "^^" and "^^^".

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)@.

The "^^^" operator *replaces* the result of a parser. Expressions using this operator have the form `P ^^^ v` where `P` is a parser and `v` is a value. But whenever `P` returns with some result `R`, the result of `P ^^ v` is v instead of `R`. So "^^^" is related to "^^" by the equality

```
P ^^^ v  =  P ^^ (x => v) .
```

As an example, here is the JSON parser that parses a numeric literal and converts it to a Scala integer:

```
numericLit ^^ (_.toInt)
```

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

```
"true" ^^^ true
```

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

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

Remember that the "~" operator produces as result an instance of a case class with the same name, "~". This is no coincidence. It is designed that way so that 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 the 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), "}") .
```

The purpose of the pattern in the code above was to "strip off" the braces so that one can get at the list of members resulting from the `repsep(member, ",")` parser.

In cases like these there is also an alternative, which avoids producing the unnecessary parser results which are then discarded by the pattern match. The alternative makes use of the "~>" and "<~" parser combinators. Both express sequential compositon just like "~", but "~>" keeps only the result of its right operand, whereas "<~" keeps only the result of its left operand. So a shorter way to express the JSON object parser would be this:

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

Here is a full JSON parser that returns meaningful results:

```
class JSON extends StandardTokenParsers {
  lexical.delimiters += ("{", "}", "[", "]", ":", ",")
  lexical.reserved += ("null", "true", "false")

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

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

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

  def value: Parser[Any] =
    obj | arr | stringLit | numericLit ^^ (_.toInt)
    "null" ^^^ null | "true" ^^^ true | "false" ^^^ false
}
```

If you run this parser on the `address-book.json` file, you get the following result (after adding some newlines and indentation):

```
java 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)
  )
)
```

Summary: Using Combinator Parsers

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

`ident`	identifier
`"if"`	keyword or special symbol
`numericLit`	integer number
`stringLit`	string literal
`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
`P ^^^ v`	constant result

5 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 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 DSL in general.

The core of Scala's combinator parsing framework is all contained in a 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 sub-sections all reside in this class.

The `StandardTokenParsers` class from which all previous example parsers inherited is itself a subclass of `Parsers`. `StandardTokenParsers` fixes some of the things that are left open in `Parsers`.

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

Here, the type of parser inputs is fixed by the definition

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

The class `Reader` comes from the package `scala.util.parsing.input`. It is similar to a `Stream` but it 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, the class `StandardTokenParsers` fixes `Elem` to be the Java-like word-tokens that we have encountered so far. So for every class inheriting from `StandardTokenParsers`, type `Elem` is an alias of class `Token` which itself is defined as an abstract class:

```
type Elem = Token
abstract class Token { def chars: String }
```

Here, the `chars` method returns the characters making up the token as a `String`. Class `Token` has four standard subclasses:

```
case class Keyword   (override val chars: String)
```
for keywords,

```
case class NumericLit(override val chars: String)
```
for numbers,

```
case class StringLit (override val chars: String)
```
for strings,

```
case class Identifier(override val chars: String)
```
for identifiers.

Parser Results

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

```
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 `Success`, `ParseResult`, and `Parser` are all parameterized with a type parameter `T` which 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 which the parser consumed. This field is needed for chaining parsers, so that one parser can operate after another one. 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 will analyze some part of the input stream, and return the remaining part as its result.

The other subclass of `ParseResult` is `Failure`. This class takes as parameter a message which describes why the parser has failed. Like `Success`, `Failure` also takes the remaining input stream as a second parameter. This is needed to position the error message at the correct place in the input stream.

The `Parser` class

In fact, the previous characterization of parsers as functions from inputs to parse result has oversimplified a bit. The examples above have shown that parsers also implement *methods* such as "$\sim$" for sequential composition of two parsers and "`|`" for their alternative composition. So `Parser` is in reality a class which inherits from the function type `Input => ParseResult` and which additionally defines these methods:

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

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

The apply method in class Parser itself is abstract. It is implemented in the individual parsers that inherit from Parser. These parsers will be discussed next.

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 Elems 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 succeeeds. 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.

The StandardTokenParsers class defines four single-token parsers for the four kinds of tokens that are supported. Each of these is defined in terms of elem:

```
/** A parser which matches a numeric literal */
def numericLit: Parser[String] =
  elem("number", _.isInstanceOf[NumericLit]) ^^ (_.chars)

/** A parser which matches a string literal */
def stringLit: Parser[String] =
  elem("string literal", _.isInstanceOf[StringLit]) ^^ (_.chars)

/** A parser which matches an identifier */
def ident: Parser[String] =
  elem("identifier", _.isInstanceOf[Identifier]) ^^ (_.chars)

/** A parser which matches a given reserved word or delimiter */
implicit def keyword(chars: String): Parser[String] =
  elem("'"+chars+"'", _ == Keyword(chars))
```

The numericLit parser succeeds if the first input token is a numeric literal of type NumericLit; if it succeeds it returns the characters making up the literal as a string.

Analogously, the ident and stringLit parsers accept identifiers and string literals.

The last of the four parsers is keyword. This one accepts a given reserved word or delimiter. For instance keyword("+") succeeds if the first token is a "+" and fails otherwise. Or keyword("true") succeeds if the first token is the reserved word true and fails otherswise.

One pecularity of the keyword method is that it carries an implicit modifier. This means that the keyword method is applied implicitly to an expression e whenever e is a string and the expected type of the expression is a Parser. In that case, the Scala compiler expands e

to `keyword(e)`. That's why we could simply write strings in place of parsers in the examples at the beginning of this chapter. For instance, the JSON parser term for an object member `stringLit ~ ":" ~ value` really means `stringLit ~ keyword(":") ~ value`.

Sequential Composition

All parsers implemented so far consume a single element. To parse more interesting phrases, we can string parsers together with the sequential composition operator "~". As you have seen before, `P ~ Q` is a parser which 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`. Here is its definition:

```
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
    }
  }
```

Let's analyze this method in detail. It 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-hand argument (or: receiver) of "~". Its right-hand argument 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 `x`, the result of `p`, and `y`, the result of `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`.

The other two sequential composition operators "<~" and "~>" can be defined just like "~", with some small adjustment how the result is computed. But a more elegant technique 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, if `P` fails, then `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 f1 @ Failure(_, _) => q(in)
    }
  }
```

Dealing with Recursion

Note that the `q` parameter in methods "∼" and "`|`" is specified to be call-by-name. This means that the actual parser argument will be evaluated only when `q` is needed; and that is the case onyly after `p` has run. This makes it possible to write recursive parsers like the following one which parses a number enclosed by arbitarily many parentheses:

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

If "`|`" and "∼" had taken call-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 two methods of class `Parser` convert a parser's result. The parser forms `P ^^ f` and `P ^^^ v` both succeed exactly when `P` succeeds but they change its result. `P ^^ f` transforms `P`'s result by applying the function `f` to it. Bu contrast, `P ^^^ v` replaces `P`'s result with the value `v`.

```
  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
    }
  }
  def ^^^ [U](v: U): Parser[U] = f ^^ (x => v)
} // end Parser
```

Parsers that don't read any input

There are also two parsers that do not consume any input. 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 which 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 ^^ (x => Some(x))  |  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())
```

Note that the `repsep` operator takes two parsers. The `P` parser for the "payload" and the `Q` parser for the separators. All applications of `repsep` so far have used a string as separator argument, as in `repsep(member, ",")`. These applications are compatible with the definition of `repsep` given here, because of the implicit `keyword` conversion for parsers. So `repsep(member, ",")` is implicitly expanded to `repsep(member, keyword(","))`.

Backtracking vs LL(1)

to do

Error reporting

to do

Summary: The combinator parser framework

These are all the essential elements of Scala's combinator parsing framework. It's surprisingly little code for something that's genuinly 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.

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. This has to do with two effects: 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) 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`, `numericLit`, "$\sim$", "`|`" and so on. They need not be aware of the fact 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 are two-fold. First, one can now factor out parser construction from input analysis. If one writes

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

and then applies `jsonParser` to several different inputs, the `jsonParser` is constructed only once, not everytime an input is read.

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

At present, such an optimizing parser generator has not yet been written. 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, there remain still reasons for also keeping the current parser combinator framework around because it is much easier to understand and to adapt than a parser generator. Furthermore, the difference in speed would often not matter in practice, unless you want to parse very large inputs.

6 Lexical Analysis

to do