Handout 6 (Parser Combinators)

This handout explains how *parser combinators* work and how they can be implemented in Scala. Their distinguishing feature is that they are very easy to implement (admittedly it is only easy in a functional programming language). However, parser combinators require that the grammar to be parsed is *not* left-recursive and they are efficient only when the grammar is unambiguous. It is the responsibility of the grammar designer to ensure these two properties.

Another good point of parser combinators is that they can deal with any kind of input as long as this input of "sequence-kind", for example a string or a list of tokens. The general idea behind parser combinators is to transform the input into sets of pairs, like so

$$\underbrace{\text{list of tokens}}_{\text{input}} \Rightarrow \underbrace{\text{set of (parsed input, unparsed input)}}_{\text{output}}$$

As said, the input can be anything as long as it is a "sequence". The only property of the input we need is to be able to test when it is empty. Obviously we can do this for strings and lists. For more lucidity we shall below often use strings as input in order to illustrate matters. However, this does not make our previous work on lexers obsolete (remember they transform a string into a list of tokens). Lexers will still be needed to build a somewhat realistic compiler.

In my Scala code I use the following polymorphic types for parser combinators:

input: `I` output: `T`

that is they take as input something of type `I` and return a set of pairs of `Set[(T, I)]`. Since the input needs to be of "sequence-kind" I actually have to often write `I <% Seq[_]` for the input type in order to indicate it is a subtype of Scala sequences. The first component of the generated pairs corresponds to what the parser combinator was able to process from the input and the second is the unprocessed part of the input (therefore the type of this unprocessed part is the same as the input). As we shall see shortly, a parser combinator might return more than one such pair; the idea being that there are potentially several ways of how to parse the input. As a concrete example, consider the string

`if foo testbar`

We might have a parser combinator which tries to interpret this string as a keyword (`if`) or as an identifier (`iffoo`). Then the output will be the set

$$\{(\texttt{if}, \texttt{foo testbar}), (\texttt{iffoo}, \texttt{testbar})\}$$

where the first pair means the parser could recognise `if` from the input and leaves the rest as 'unprocessed' as the second component of the pair; in the other case it could recognise `iffoo` and leaves `testbar` as unprocessed. If the

parser cannot recognise anything from the input, then parser combinators just return the empty set $\{\}$. This will indicate something "went wrong"…or more precisely, nothing could be parsed.

Also important to note is that the type `T` for the processed part is different from the input type. The reason is that in general we are interested in transform our input into something "different"…for example into a tree, or if we implement the grammar for arithmetic expressions we might be interested in the actual integer number the arithmetic expression, say `1 + 2 * 3`, stands for. In this way we can use parser combinators to implement relatively easily a calculator.

The main idea of parser combinators is that we can easily build parser combinators out of smaller components following very closely the structure of a grammar. In order to implement this in an object-oriented programming language, like Scala, we need to specify an abstract class for parser combinators. This abstract class states that the function `parse` takes an argument of type `I` and returns a set of type `Set[(T, I)]`.

```
abstract class Parser[I, T] {
  def parse(ts: I): Set[(T, I)]

  def parse_all(ts: I): Set[T] =
    for ((head, tail) <- parse(ts); if (tail.isEmpty))
      yield head
}
```

It is the obligation in each instance (parser combinator) to supply an implementation for `parse`. From this function we can then "centrally" derive the function `parse_all`, which just filters out all pairs whose second component is not empty (that is has still some unprocessed part). The reason is that at the end of the parsing we are only interested in the results where all the input has been consumed and no unprocessed part is left over.

One of the simplest parser combinators recognises just a single character, say c, from the beginning of strings. Its behaviour can be described as follows:

- If the head of the input string starts with a c, then return the set

$$\{(c, \textit{tail of } s)\}$$

  where *tail of s* is the unprocessed part of the input string.

- Otherwise return the empty set $\{\}$.

The input type of this simple parser combinator for characters is `String` and the output type `Set[(Char, String)]`. The code in Scala is as follows:

```
case class CharParser(c: Char) extends Parser[String, Char] {
  def parse(sb: String) =
    if (sb.head == c) Set((c, sb.tail)) else Set()
}
```

You can see the `parse` function tests whether the first character of the input string `sb` is equal to `c`. If yes, then it splits the string into the recognised part `c` and the unprocessed part `sb.tail`. In case `sb` does not start with `c` then the parser returns the empty set (in Scala `Set()`). Since this parser recognises characters and just returns characters as the processed part, the output type of the parser is `Char`.

If we want to parse a list of tokens and interested in recognising a number token, we could write something like this

```
case object NumParser extends Parser[List[Token], Int] {
  def parse(ts: List[Token]) = ts match {
    case Num_token(s)::ts => Set((s.toInt, ts))
    case _ => Set ()
  }
}
```

In this parser the input is of type `List[Token]`. The function parse looks at the input `ts` and checks whether the first token is a `Num_token`. Let us assume our lexer generated these tokens for numbers. But this parser does not just return this token (and the rest of the list), like the `CharParser` above, rather extracts the string `s` from the token and converts it into an integer. The hope is that the lexer did its work well and this conversion always succeeds. The consequence of this is that the output type for this parser is `Int`. Such a conversion would be needed if we want to implement a simple calculator program.

These simple parsers that just look at the input and do a simple transformation are often called *atomic* parser combinators. More interesting are the parser combinators that build larger parsers out of smaller component parsers. For example the *alternative parser combinator* is as follows: given two parsers, say, p and q, we apply both parsers to the input (remember parsers are functions) and combine the output (remember they are sets of pairs)

$$p(\text{input}) \cup q(\text{input})$$

In Scala we would implement alternative parser combinator as follows

```
class AltParser[I, T]
       (p: => Parser[I, T],
        q: => Parser[I, T]) extends Parser[I, T] {
  def parse(sb: I) = p.parse(sb) ++ q.parse(sb)
}
```

The types of this parser combinator are again generic (we just have `I` for the input type, and `T` for the output type). The alternative parser builds a new parser out of two existing parsers `p` and `q`. Both need to be able to process

input of type `I` and return the same output type `Set[(T, I)]`.[1] Therefore the output type of this parser is `T`. The alternative parser should run the input with the first parser `p` (producing a set of pairs) and then run the same input with `q` (producing another set of pairs). The result should be then just the union of both sets, which is the operation `++` in Scala.

The alternative parser combinator already allows us to construct a parser that parses either a character `a` or `b`, as

```
new AltParser(CharParser('a'), CharParser('b'))
```

Later on we will use again Scala mechanism for introducing some more readable shorthand notation for this, like `"a" || "b"`. Let us look in detail at what this parser combinator produces with some sample strings

input strings		output
a c d e	$\rightarrow$	$\{($a, c d e$)\}$
b c d e	$\rightarrow$	$\{($b, c d e$)\}$
c c d e	$\rightarrow$	$\{\}$

We receive in the first two cases a successful output (that is a non-empty set). In each case, either `a` or `b` is in the processed part, and `cde` in the unprocessed part. Clearly this parser cannot parse anything in the string `ccde`, therefore the empty set.

A bit more interesting is the *sequence parser combinator*. Given two parsers, say again, p and q, we want to apply first the input to p producing a set of pairs; then apply q to all the unparsed parts in the pairs; and then combine the results like

$$\{((output_1, output_2), u_2) \quad | \quad (output_1, u_1) \in p(\text{input}) \wedge (output_2, u_2) \in q(u_1)\}$$

Notice that the p wil first be run on the input, producing pairs of the form $(output_1, u_1)$ where the u_1 stands for the unprocessed, or left-over, parts. We want that q runs on all these unprocessed parts u_1. This again will produce some processed part , p and q, we apply both parsers to the input (remember parsers are functions) and combine the output (remember they are sets of pairs)

$$p(\text{input}) \cup q(\text{input})$$

In Scala we would implement alternative parser combinator as follows

[1] There is an interesting detail of Scala, namely the `=>` in front of the types of `p` and `q`. They will prevent the evaluation of the arguments before they are used. This is often called *lazy evaluation* of the arguments. We will explain this later.

```
class AltParser[I, T]
       (p: => Parser[I, T],
        q: => Parser[I, T]) extends Parser[I, T] {
  def parse(sb: I) = p.parse(sb) ++ q.parse(sb)
}
```

The types of this parser combinator are again generic (we just have `I` for the input type, and `T` for the output type). The alternative parser builds a new parser out of two existing parsers `p` and `q`. Both need to be able to process input of type `I` and return the same output type `Set[(T, I)]`.[2] Therefore the output type of this parser is `T`. The alternative parser should run the input with the first parser `p` (producing a set of pairs) and then run the same input with `q` (producing another set of pairs). The result should be then just the union of both sets, which is the operation `++` in Scala.

The alternative parser combinator already allows us to construct a parser that parses either a character `a` or `b`, as

```
new AltParser(CharParser('a'), CharParser('b'))
```

Later on we will use again Scala mechanism for introducing some more readable shorthand notation for this, like `"a" || "b"`. Let us look in detail at what this parser combinator produces with some sample strings

input strings		output
a c d e	$\rightarrow$	$\{($a, c d e$)\}$
b c d e	$\rightarrow$	$\{($b, c d e$)\}$
c c d e	$\rightarrow$	$\{\}$

We receive in the first two cases a successful output (that is a non-empty set). In each case, either `a` or `b` is in the processed part, and `cde` in the unprocessed part. Clearly this parser cannot parse anything in the string `ccde`, therefore the empty set.

A bit more interesting is the *sequence parser combinator*. Given two parsers, say again, p and q, we want to apply first the input to p producing a set of pairs; then apply q to all the unparsed parts in the pairs; and then combine the results like

$$\{((output_1, output_2), u_2) \quad | \quad (output_1, u_1) \in p(\text{input}) \wedge (output_2, u_2) \in q(u_1)\}$$

Notice that the p wil first be run on the input, producing pairs of the form $output_1$ and unprocessed part u_1. The overall result of the sequence parser combinator is pairs of the form $((output_1, output_2), u_2)$. This means the unprocessed

[2] There is an interesting detail of Scala, namely the `=>` in front of the types of `p` and `q`. They will prevent the evaluation of the arguments before they are used. This is often called *lazy evaluation* of the arguments. We will explain this later.

parts of both parsers is the unprocessed parts the second parser q produces as left-over. The processed parts of both parsers is just the pair of the outputs $(output_1, output_2)$. This behaviour can be implemented in Scala as follows:

```
class SeqParser[I, T, S]
       (p: => Parser[I, T],
        q: => Parser[I, S]) extends Parser[I, (T, S)] {
  def parse(sb: I) =
    for ((output1, u1) <- p.parse(sb);
         (output2, u2) <- q.parse(u1))
            yield ((output1, output2), u2)
}
```

This parser takes again as input two parsers, `p` and `q`. It implements `parse` as follows: let first run the parser `p` on the input producing a set of pairs (`output1`, `u1`). The `u1` stands for the unprocessed parts left over by `p`. Let `q` run on these unprocessed parts producing again a set of pairs. The output of the sequence parser combinator is then a set containing pairs where the first components are again pairs, namely what the first parser could parse together with what the second parser could parse; the second component is the unprocessed part left over after running the second parser `q`. Therefore the input type of the sequence parser combinator is as usual `I`, but the output type is

```
Set[((T, S), I)]
```

If any of the runs of p and q fail, that is produce the empty set, then `parse` will also produce the empty set. Notice that we have now two output types for the sequence parser combinator, because in general p and q might produce different things (for example first we recognise a number and then a string corresponding to an operator).

We have not yet looked at this in detail, but Scala allows us to provide some shorthand notation for the sequence parser combinator. We can write for example `"a" ~ "b"`, which is the parser combinator that first recognises the character `a` from a string and then `b`. Let us look again at three examples of how this parser combinator processes strings:

input strings		output
a b c d e	$\rightarrow$	{((a, b), c d e)}
b a c d e	$\rightarrow$	{}
c c c d e	$\rightarrow$	{}

In the first line we have a successful parse, because the string started with `ab`, which is the prefix we are looking for. But since the parsing combinator is constructed as sequence of the two simple (atomic) parsers for `a` and `b`, the result is a nested pair of the form `((a, b), cde)`. It is *not* a simple pair `(ab, cde)` as

one might erroneously expects. The parser returns the empty set in the other examples, because they do not fit with what the parser is supposed to parse.

A slightly more complicated parser is `("a" || "b") ~ "c"` which parses as first character either an `a` or `b` followed by a `c`. This parser produces the following outputs.

input strings		output
`a c d e`	$\rightarrow$	$\{((a, c), d\ e)\}$
`b c d e`	$\rightarrow$	$\{((b, c), d\ e)\}$
`a b d e`	$\rightarrow$	$\{\}$

Now consider the parser `("a" ~ "b") ~ "c"` which parses `a`, `b`, `c` in sequence. This parser produces the following outputs.

input strings		output
`a b c d e`	$\rightarrow$	$\{(((a, b), c), d\ e)\}$
`a b d e`	$\rightarrow$	$\{\}$
`b c d e`	$\rightarrow$	$\{\}$

The second and third example fail, because something is "missing" in the sequence we are looking for. Also notice how the results nest with sequences: the parsed part is a nested pair of the form `((a, b), c)`. Two more examples: first consider the parser `("a" ~ "a") ~ "a"` and the input `aaaa`:

input string		output
`a a a a`	$\rightarrow$	$\{(((a, a), a), a)\}$

Notice how the results nests deeper and deeper as pairs (the last `a` is in the unprocessed part). To consume everything of this string we can use the parser `(("a" ~ "a") ~ "a") ~ "a"`. Then the output is as follows:

input string		output
`a a a a`	$\rightarrow$	$\{((((a, a), a), a), "")\}$

This is an instance where the parser consumed completely the input, meaning the unprocessed part is just the empty string. So if we called `parse_all` instead of `parse` we would get back the result

$$\{(((a, a), a), a)\}$$

where the unprocessed (empty) parts have been stripped away from the pairs; everything where the second part was not empty has been thrown away as ultimately-unsuccessful-parses.

Note carefully that constructing a parser such `'a' || ('a' ~ 'b')` will result in a typing error. The first parser has as output type a single character (recall the type of `CharParser`), but the second parser produces a pair of characters

as output. The alternative parser is however required to have both component parsers to have the same type. We will see later how we can build this parser without the typing error.

The next parser combinator does not actually combine smaller parsers, but applies a function to the result of a parser. It is implemented in Scala as follows

```
class FunParser[I, T, S]
         (p: => Parser[I, T],
          f: T => S) extends Parser[I, S] {
  def parse(sb: I) =
    for ((head, tail) <- p.parse(sb)) yield (f(head), tail)
}
```

This parser combinator takes a parser `p` with output type `T` as one argument as well as a function `f` with type `T => S`. The parser `p` produces sets of type `(T, I)`. The `FunParser` combinator then applies the function `f` to all the parser outputs. Since this function is of type `T => S`, we obtain a parser with output type `S`. Again Scala lets us introduce some shorthand notation for this parser combinator. Therefore we will write `p ==> f` for it.

How to build parsers using parser combinators?

Implementing an Interpreter