Handout 6 (Parser Combinators)

In what follows we explain *parser combinators*, because they are very easy to implement. However, they only work when 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.

Parser combinators can deal with any kind of input as long as this input is a kind of sequence, for example a string or a list of tokens. If the input are lists of tokens, then parser combinators transform them into sets of pairs, like so

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

In Scala parser combinators will therefore be functions of type

```
I ⇒ Set[(T, I)]
```

that is they take as input something of type `I` and return a set of pairs. The first component of these pairs corresponds to what the parser combinator was able to process from the input and the second is the unprocessed part of the input. As we shall see shortly, a parser combinator might return more than one such pair, with the idea that there are potentially several ways how to interpret the input. To simplify matters we will first look at the case where the input to the parser combinators is just strings. As a concrete example, consider the case where the input is the string

`i f f o o ␣ t e s t b a r`

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

$$\{(\texttt{i f}, \texttt{f o o ␣ t e s t b a r}), (\texttt{i f f o o}, \texttt{␣ t e s t b a r})\}$$

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.

The main attraction 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 requires the implementation of the function `parse` taking 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
}
```

From the function `parse` 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 parsing we are only interested in the results where all the input has been consumed and no unprocessed part is left.

One of the simplest parser combinators recognises just a 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, it returns the set $\{(c, \textit{tail of } s)\}$, where *tail of* s is the unprocessed part of the input string
- otherwise it returns 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()
}
```

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()`).

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, then we apply both parsers to the input (remember parsers are functions) and combine the input

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

In Scala we would implement alternative parsers 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 polymorphic (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 parser combinator `p` and `q`. Both need to be able to process input of type `I` and return the same output type `Set[(T, I)]`. (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.) The alternative parser should run the input with the first parser `p` (producing a set of outputs) and then run the same input with `q`. The result should be then just the union of both sets, which is the operation `++` in Scala.

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

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

Scala allows us to introduce some more readable shorthand notation for this, like `'a' || 'b'`. We can call this parser combinator with the strings

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

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

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

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

This 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 ((head1, tail1) <- p.parse(sb);
         (head2, tail2) <- q.parse(tail1))
            yield ((head1, head2), tail2)
}
```

This parser takes 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 (`head1`, `tail1`). The `tail1` 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)]
```

Scala allows us to provide some shorthand notation for the sequence parser combinator. So we can write for example `'a'` $\sim$ `'b'`, which is the parser combinator that first consumes the character `a` from a string and then `b`. Three examples of this parser combinator are as follows:

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

A slightly more complicated parser is `('a' || 'b')` $\sim$ `'b'` which parses as first character either an `a` or `b` followed by a `b`. This parser produces the following results.

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

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 nest deeper as pairs (the last `a` is in the unprocessed part). To consume everything 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.

Note carefully that constructing a parser such `'a' || ('a' ~ 'b')` will result in a tying 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 the 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 input 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 parer 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.