Handout 6 (Parser Combinators)

This handout explains how *parser combinators* work and how they can be implemented in Scala. Their most distinguishing feature is that they are very easy to implement (admittedly it is only easy in a functional programming language). Another good point of parser combinators is that they can deal with any kind of input as long as this input is of "sequence-kind", for example a string or a list of tokens. The only two properties of the input we need is to be able to test when it is empty and "sequentially" take it apart. Strings and lists fit this bill. However, parser combinators also have their drawbacks. For example they 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 hold.

The general idea behind parser combinators is to transform the input into sets of pairs, like so

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

Given the extended effort we have spent implementing a lexer in order to generate lists of tokens, it might be surprising that in what follows we shall often use strings as input, rather than lists of tokens. This is for making the explanation more lucid and for quick examples. It does not make our previous work on lexers obsolete (remember they transform a string into a list of tokens). Lexers will still be needed for building a somewhat realistic compiler.

As mentioned above, parser combinators are relatively agnostic about what kind of input they process. 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 type `Set[(T, I)]`. Since the input needs to be of "sequence-kind", I actually have to often write `I <% Seq[_]` for the input type. This ensures the input is a subtype of Scala sequences. The first component of the generated pairs corresponds to what the parser combinator was able to parse from the input and the second is the unprocessed, or leftover, part of the input (therefore the type of this unprocessed part is the same as the input). A parser combinator might return more than one such pair; the idea is that there are potentially several ways of how to parse the input. As a concrete example, consider the string

`iffoo␣test␣bar`

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

$$\{(\texttt{if}, \texttt{foo␣test␣bar}), (\texttt{iffoo}, \texttt{␣test␣bar})\}$$

where the first pair means the parser could recognise `if` from the input and leaves the `foo_testbar` as unprocessed part; in the other case it could recognise `iffoo` and leaves `_testbar` as unprocessed. If the parser cannot recognise anything from the input at all, 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 output type `T` for the processed part can potentially be different from the input type `I` in the parser. In the example above is just happens to be the same. The reason for the difference is that in general we are interested in transforming 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, for instance (we shall do this later on).

The main driving force behind 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 a functional/object-oriented programming language, like Scala, we need to specify an abstract class for parser combinators. In the abstract class we specify that `I` is the *input type* of the parser combinator and that `T` is the *output type*. This implies 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(in: I) : Set[(T, I)]

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

It is the obligation in each instance of this class 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 is `String` and the output type is `Char`. This means `parse` returns `Set[(Char, String)]`. The code in Scala is as follows:

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

You can see `parse` tests whether the first character of the input string `in` is equal to `c`. If yes, then it splits the string into the recognised part `c` and the unprocessed part `in.tail`. In case `in` 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, for example, 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 it extracts also 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`, not `Token`. Such a conversion would be needed if we want to implement a simple calculator program, because string-numbers need to be transformed into `Int`-numbers in order to do the calculations.

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. There are three such parser combinators that can be implemented generically. 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 can implement alternative parser combinator as follows

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

The types of this parser combinator are again generic (we 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` which are given as arguments. Both parsers need to be able to process input of type `I` and return in `parse` the same output type `Set[(T, I)]`.[1] The alternative parser runs the input with the first parser `p` (producing a set of pairs) and then runs 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 allows us to construct a parser that parses either a character `a` or `b` using the `CharParser` shown above. For this we can write

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

Later on we will use 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 parsed part, and `cde` in the unprocessed part. Clearly this parser cannot parse anything with `ccde`, therefore the empty set is returned.

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. Mathematically we would write something like this for the set of pairs:

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

[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.

Notice that the p will first be run on the input, producing pairs of the form $(output_1, u_1)$ where the u_1 stands for the unprocessed, or leftover, parts of p. We want that q runs on all these unprocessed parts u_1. Therefore these unprocessed parts are fed into the second parser q. The overall result of the sequence parser combinator is pairs of the form $((output_1, output_2), u_2)$. This means the unprocessed part of the sequence parser combinator is the unprocessed part the second parser q leaves as leftover. The parsed parts of the component parsers are combined in a pair, namely $(output_1, output_2)$. The reason is we want to know what p and q were able to parse. 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(in: I) =
    for ((output1, u1) <- p.parse(in);
         (output2, u2) <- q.parse(u1))
            yield ((output1, output2), u2)
}
```

This parser takes again as arguments two parsers, `p` and `q`. It implements `parse` as follows: 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` (recall that there can be several such pairs). Let then `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`. Note that the input type of the sequence parser combinator is as usual `I`, but the output type is

```
(T, S)
```

Consequently, the function `parse` in the sequence parser combinator returns sets of type `Set[((T, S), I)]`. That means we have essentially two output types for the sequence parser combinator (packaged in a pair), because in general p and q might produce different things (for example we recognise a number with `p` and then with `q` a string corresponding to an operator). If any of the runs of p and q fail, that is produce the empty set, then `parse` will also produce the empty set.

With the shorthand notation we shall introduce later 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 some examples of how this parser combinator processes some 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 starts 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 expect. 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. The first succeeds but notice how the results nest with sequences: the parsed part is a nested pair of the form `((a, b), c)`. If we nest the sequence parser differently, say `"a" ~ ("b" ~ "c")`, then also our output pairs nest differently

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

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 again how the results nest 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	$\rightarrow$	$\{((((\texttt{a}, \texttt{a}), \texttt{a}), \texttt{a}), \texttt{""})\}$

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

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

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 well, because they represent ultimately-unsuccessful-parses. The main point is that the sequence parser combinator returns pairs that can nest according to the nesting of the component parsers.

Consider also carefully that constructing a parser such `"a" | ("a" ~ "b")` will result in a typing error. The intention with this parser is that we want to parse either an `a`, or an `a` followed by a `b`. However, 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 required to have both component parsers to have the same type—the reason is that we need to be able to build the union of two sets, which requires in Scala that the sets have the same type. Since they are not in this case, there is a typing error. We will see later how we can build this parser without the typing error.

The next parser combinator, called *semantic action*, does not actually combine two 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(in: I) =
    for ((head, tail) <- p.parse(in)) yield (f(head), tail)
}
```

This parser combinator takes a parser `p` (with input type `I` and output type `T`) as one argument but also a function `f` (with type `T => S`). The parser `p` produces sets of type `Set[(T, I)]`. The semantic action combinator then applies the function `f` to all the 'processed' 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 short `p ==> f` for it.

What are semantic actions good for? Well, they allow you to transform the parsed input into datastructures you can use for further processing. A simple (contrived) example would be to transform parsed characters into ASCII numbers. Suppose we define a function `f` (from characters to `Int`s) and use a `CharParser` for parsing the character `c`.

```
val f = (c: Char) => c.toInt
val c = new CharParser('c')
```

We then can run the following two parsers on the input `cbd`:

```
c.parse("cbd")
(c ==> f).parse("cbd")
```

In the first line we obtain the expected result `Set(('c', "bd"))`, whereas the second produces `Set((99, "bd"))`—the character has been transformed into an ASCII number.

A slightly less contrived example is about parsing numbers (recall `NumParser` above). However, we want to do this here for strings, not for tokens. For this assume we have the following (atomic) `RegexParser`.

```
import scala.util.matching.Regex

case class RegexParser(reg: Regex) extends Parser[String, String] {
  def parse(in: String) = reg.findPrefixMatchOf(in) match {
    case None => Set()
    case Some(m) => Set((m.matched, m.after.toString))
  }
}
```

This parser takes a regex as argument and splits up a string into a prefix and the rest according to this regex (`reg.findPrefixMatchOf` generates a match—in the successful case—and the corresponding strings can be extracted with `matched` and `after`). The input and output type for this parser is `String`. Using `RegexParser` we can define a `NumParser` for `Strings` to `Int` as follows:

```
val NumParser = RegexParser("[0-9]+".r)
```

This parser will recognise a number at the beginning of a string. For example

```
NumParser.parse("123abc")
```

produces `Set((123,abc))`. The problem is that `123` is still a string (the required double-quotes are not printed by Scala). We want to convert this string into the corresponding `Int`. We can do this as follows using a semantic action

```
(NumParser ==> (s => s.toInt)).parse("123abc")
```

The function in the semantic action converts a string into an `Int`. Now `parse` generates `Set((123,abc))`, but this time `123` is an `Int`. Let us come back to semantic actions when we are going to implement actual context-free grammars.

Shorthand notation for parser combinators

Before we proceed, let us just explain the shorthand notation for parser combinators. Like for regular expressions, the shorthand notation will make our life much easier when writing actual parsers. We can define some implicits which allow us to write

`p | q`	alternative parser
`p ~ q`	sequence parser
`p ==> f`	semantic action parser

as well as to use plain strings for specifying simple string parsers.

The idea is that this shorthand notation allows us to easily translate context-free grammars into code. For example recall our context-free grammar for palindromes:

$$\begin{array}{rcl} \textit{Pal} & ::= & a \cdot \textit{Pal} \cdot a \mid b \cdot \textit{Pal} \cdot b \mid a \mid b \\ & & \mid \epsilon \end{array}$$

Each alternative in this grammar translates into an alternative parser combinator. The $\cdot$ can be translated to a sequence parser combinator. The parsers for a, b and ϵ can be simply written as `"a"`, `"b"` and `""`.

How to build parsers using parser combinators?

The beauty of parser combinators is the ease with which they can be implemented and how easy it is to translate context-free grammars into code (though the grammars need to be non-left-recursive). To demonstrate this consider again the grammar for palindromes from above. The first idea would be to translate it into the following code

```
lazy val Pal : Parser[String, String] =
  (("a" ~ Pal ~ "a") | ("b" ~ Pal ~ "b") | "a" | "b" | "")
```

Unfortunately, this does not quite work yet as it produces a typing error. The reason is that the parsers `"a"`, `"b"` and `""` all produce strings as output type and therefore can be put into an alternative `...| "a" | "b" | ""`. But both sequence parsers `"a" ~ Pal ~ "a"` and `"b" ~ Pal ~ "b"` produce pairs of the form

(((a-part, Pal-part), a-part), unprocessed part)

That is how the sequence parser combinator nests results when `~` is used between two components. The solution is to use a semantic action that "flattens" these pairs and appends the corresponding strings, like

```
lazy val Pal : Parser[String, String] =
  (("a" ~ Pal ~ "a") ==> { case ((x, y), z) => x + y + z } |
   ("b" ~ Pal ~ "b") ==> { case ((x, y), z) => x + y + z } |
    "a" | "b" | "")
```

How does this work? Well, recall again what the pairs look like for the parser `"a" ~ Pal ~ "a"`. The pattern in the semantic action matches the nested pairs (the `x` with the `a`-part and so on). Unfortunately when we have such nested pairs, Scala requires us to define the function using the `case`-syntax

```
{ case ((x, y), z) => ... }
```

If we have more sequence parser combinators or have them differently nested, then the pattern in the semantic action needs to be adjusted accordingly. The action we implement above is to concatenate all three strings, which means after the semantic action is applied the output type of the parser is `String`, which means it fits with the alternative parsers `...| "a" | "b" | ""`.

If we run the parser above with `Pal.parse_all("abaaaba")` we obtain as result the `Set(abaaaba)`, which indicates that the string is a palindrome (an empty set would mean something is wrong). But also notice what the intermediate results are generated by `Pal.parse("abaaaba")`

```
Set((abaaaba,""),(aba,aaba), (a,baaaba), ("",abaaaba))
```

That there are more than one output might be slightly unexpected, but can be explained as follows: the pairs represent all possible (partial) parses of the string `"abaaaba"`. The first pair above corresponds to a complete parse (all output is consumed) and this is what `Pal.parse_all` returns. The second pair is a small "sub-palindrome" that can also be parsed, but the parse fails with the rest `aaba`, which is therefore left as unprocessed. The third one is an attempt to parse the whole string with the single-character parser `a`. That of course only partially succeeds, by leaving `"baaaba"` as the unprocessed part. Finally, since we allow the empty string to be a palindrome we also obtain the last pair, where actually nothing is consumed from the input string. While all this works as intended, we need to be careful with this (especially with including the `""` parser in our grammar): if during parsing the set of parsing attempts gets too big, then the parsing process can become very slow as the potential candidates for applying rules can snowball.

Important is also to note is that we must define the `Pal`-parser as a *lazy* value in Scala. Look again at the code: `Pal` occurs on the right-hand side of the definition. If we had just written

```
val Pal : Parser[String, String] =  ...rhs...
```

then Scala before making this assignment to `Pal` attempts to find out what the expression on the right-hand side evaluates to. This is straightforward in case of simple expressions `2 + 3`, but the expression above contains `Pal` in the right-hand side. Without `lazy` it would try to evaluate what `Pal` evaluates to and start a new recursion, which means it falls into an infinite loop. The definition of `Pal` is recursive and the `lazy` key-word prevents it from being fully evaluated. Therefore whenever we want to define a recursive parser we have to write

```
lazy val SomeParser : Parser[...,...] =  ...rhs...
```

That was not necessary for our atomic parsers, like `RegexParser` or `CharParser`, because they are not recursive. Note that this is also the reason why we had to write

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

class SeqParser[I, T, S]
       (p: => Parser[I, T],
        q: => Parser[I, S]) extends Parser[I, (T, S)] {...}
```

where the `=>` in front of the argument types for `p` and `q` prevent Scala from evaluating the arguments. Normally, Scala would first evaluate what kind of parsers `p` and `q` are, and only then generate the alternative parser combinator, respectively sequence parser combinator. Since the arguments can be recursive parsers, such as `Pal`, this would lead again to an infinite loop.

As a final example in this section, let us consider the grammar for well-nested parentheses:

$$\boldsymbol{P} ::= (\cdot \boldsymbol{P} \cdot) \cdot \boldsymbol{P} \mid \epsilon$$

Let us assume we want to not just recognise strings of well-nested parentheses but also transform round parentheses into curly braces. We can do this by using a semantic action:

```
lazy val P : Parser[String, String] =
  "(" ~ P ~ ")" ~ P ==> { case (((_,x),_),y) => "{" + x + "}" + y } | ""
```

Here we define a function where which ignores the parentheses in the pairs, but replaces them in the right places with curly braces when assembling the new string in the right-hand side. If we run `P.parse_all("((((()))))())")` we obtain `Set({{{{}{}}}{}})` as expected.

Implementing an Interpreter

The first step before implementing an interpreter for a full-blown language is to implement a simple calculator for arithmetic expressions. Suppose our arithmetic expressions are given by the grammar:

$$\begin{array}{rcl} \boldsymbol{E} & ::= & \boldsymbol{E} \cdot + \cdot \boldsymbol{E} \\ & | & \boldsymbol{E} \cdot - \cdot \boldsymbol{E} \\ & | & \boldsymbol{E} \cdot * \cdot \boldsymbol{E} \\ & | & (\cdot \boldsymbol{E} \cdot) \\ & | & \mathit{Number} \end{array}$$

Naturally we want to implement the grammar in such a way that we can calculate what the result of, for example, `4*2+3` is—we are interested in an `Int` rather than a string. This means every component parser needs to have as output type `Int` and when we assemble the intermediate results, strings like `"+"`, `"*"` and so on, need to be translated into the appropriate Scala operation of adding, multiplying and so on. Being inspired by the parser for well-nested parentheses above and ignoring the fact that we want $*$ to take precedence over $+$ and $-$, we might want to write something like

```
lazy val E: Parser[String, Int] =
  (E ~ "+" ~ E ==> { case ((x, y), z) => x + z} |
   E ~ "-" ~ E ==> { case ((x, y), z) => x - z} |
   E ~ "*" ~ E ==> { case ((x, y), z) => x * z} |
   "(" ~ E ~ ")" ==> { case ((x, y), z) => y} |
   NumParserInt)
```

Consider again carefully how the semantic actions pick out the correct arguments for the calculation. In case of plus, we need `x` and `z`, because they correspond to the results of the component parser `E`. We can just add `x + z` in order to obtain an `Int` because the output type of `E` is `Int`. Similarly with subtraction and multiplication. In contrast in the fourth clause we need to return `y`, because it is the result enclosed inside the parentheses. The information about parentheses, roughly speaking, we just throw away.

So far so good. The problem arises when we try to call `parse_all` with the expression `"1+2+3"`. Lets try it

```
E.parse_all("1+2+3")
```

…and we wait and wait and …still wait. What is the problem? Actually, the parser just fell into an infinite loop! The reason is that the above grammar is left-recursive and recall that our parser combinators cannot deal with such left-recursive grammars. Fortunately, every left-recursive context-free grammar can be transformed into a non-left-recursive grammars that still recognises the same strings. This allows us to design the following grammar

$$
\begin{array}{lcl}
\boldsymbol{E} & ::= & \boldsymbol{T} \cdot + \cdot \boldsymbol{E} \mid \boldsymbol{T} \cdot - \cdot \boldsymbol{E} \mid \boldsymbol{T} \\
\boldsymbol{T} & ::= & \boldsymbol{F} \cdot * \cdot \boldsymbol{T} \mid \boldsymbol{F} \\
\boldsymbol{F} & ::= & (\cdot \boldsymbol{E} \cdot) \mid \mathit{Number}
\end{array}
$$

Recall what left-recursive means from Handout 5 and make sure you see why this grammar is *non* left-recursive. This version of the grammar also deals with the fact that $*$ should have a higher precedence. This does not affect which strings this grammar can recognise, but in which order we are going to evaluate any arithmetic expression. We can translate this grammar into parsing combinators as follows:

```
lazy val E: Parser[String, Int] =
  (T ~ "+" ~ E) ==> { case ((x, y), z) => x + z } |
  (T ~ "-" ~ E) ==> { case ((x, y), z) => x - z } | T
lazy val T: Parser[String, Int] =
  (F ~ "*" ~ T) ==> { case ((x, y), z) => x * z } | F
lazy val F: Parser[String, Int] =
  ("(" ~ E ~ ")") ==> { case ((x, y), z) => y } | NumParserInt
```

Let us try out some examples:

input strings		output of `parse_all`
`1 + 2 + 3`	$\rightarrow$	`Set(6)`
`4 * 2 + 3`	$\rightarrow$	`Set(11)`
`4 * ( 2 + 3 )`	$\rightarrow$	`Set(20)`
`( 4 ) * ( ( 2 + 3 ) )`	$\rightarrow$	`Set(20)`
`4 / 2 + 3`	$\rightarrow$	`Set()`
`1 ␣ + ␣ 2 ␣ + ␣ 3`	$\rightarrow$	`Set()`

Note that we call `parse_all`, not `parse`. The examples should be quite self-explanatory. The last two example do not produce any integer result because our parser does not define what to do in case of division (could be easily added), but also has no idea what to do with whitespaces. To deal with them is the task of the lexer! Yes, we can deal with them inside the grammar, but that would render many grammars becoming unintelligible, including this one.[2]

[2] If you think an easy solution is to extend the notion of what a number should be, then think again—you still would have to deal with cases like `( ␣ ( ␣ 2 + 3 ) ␣ )`. Just think of the mess you would have in a grammar for a full-blown language where there are numerous such cases.