1 Introduction

In projects `1-hello` to `9-final` we gradually built up a framework for parsing strings using grammars. We started with easy arithmetic expressions (`product of sum of 2 and 5 and 3`), which are easily parseable by a recursive descent parser, since they're non-ambiguous. A recursive descent parser basically tries to apply every rule until it finds a valid chain of applications.
Then we continued on to more difficult expressions, removing `product of` and `sum of` by defining a left recursive grammar; rules like $A \to A\alpha$ are called left-recursive. These lead to an endless recursion when using a recursive descent parser. Therefore we had to modify the grammar used by hand to still be able to parse our expressions. Doing this automatically is the topic of my final project.
Later on we concerned ourselves with precedence, associativity and pretty printing as well. My project is able to handle associativity; precedence can be handled by carefully crafting the grammar.

2 Transforming left-recursive grammars

Let's say we have a rule like this:

$$\begin{aligned}\langle A\rangle \ ::= \ & \langle A\rangle\ \alpha_1 \mid \langle A\rangle\ \alpha_2 \mid \dots \\ \mid \ & \beta_1 \mid \beta_2 \mid \dots\end{aligned}$$

If we were to form a word using the rule, we'd eventually have to replace $\langle A\rangle$ by one of the β_is. This would then be followed by any amount of α_js (even none). Therefore we can form the following, equivalent rules:

$$\langle A\rangle \ ::= \ \beta_1\langle A\text{'}\rangle \mid \beta_2\langle A\text{'}\rangle \mid \dots$$

$$\langle A\text{'}\rangle ::= \ \alpha_1\langle A\text{'}\rangle \mid \alpha_2\langle A\text{'}\rangle \mid \dots \mid \epsilon$$

Where $\langle A\text{'}\rangle$ is a nonterminal not yet used in the grammar, and ϵ is the empty word.
By applying this method to every rule, we can transform a directly left-recursive grammar to a non left-recursive grammar producing the same words. Not covered are loops of length 2 or more; however this is not very complicated in principle and follows the same general rule as eliminating direct left recursion.

3 Implementation

For this example I'll be using the following simple grammar to parse arithmetic expressions:

```
val lrG: Grammar =
    Grammar(
      start = add,
      rules = Map(
        add -> ((add ~ #+ ~ add) || mul),
        mul -> ((mul ~ #* ~ mul) || num)
      ),
      associativity = Map(),
      precedence = Map()
    )
```

Where `~` is the concatenation of primitives and `||` is the union. You may notice the fields `associativity` and `precedence`; these are only used for declaring associativity of operators, I'll come to that later.
To eliminate left recursion of a rule, I declared the following method:

```
def transformLRRule(lhs: Nonterminal, rhses: List[RuleRHS], join: List[RuleRHS] =>
    RuleRHS): Map[Nonterminal, RuleRHS] = {
    var alphas = ListBuffer[RuleRHS]()
    var betas = ListBuffer[RuleRHS]()
```

```
    for(rhs <- rhses) {
      rhs match {
        case t:Terminal => betas += t
        case c:Comment => betas += c
        case nt:Nonterminal if(nt != lhs) => betas += nt
        case s:Sequence => {
          val collapsed = collapseSequence(s)
          println(collapsed.head)
          if(collapsed.head == lhs){
            alphas += joinToSequence(collapsed.tail)
          }
          else{
            betas += s
          }
        }
      }
    }
    var secondNT = nextNonterminal(lhs)
    var lhsTo = join(betas.toList.map {
      r => (r ~~ secondNT)
    })
    var sNTTo = join(alphas.toList.map {
      r => (r ~~ secondNT)
    } :+ eps)
    Map(
      lhs -> lhsTo,
      secondNT -> sNTTo
    )
  }
```

It takes as arguments the left hand side symbol, a list of alternative right sides and a function which rebuilds the right sides into single rules. Those are returned as a Map for easier handling. It builds a list of αs and βs as seen in 2 and "guesses" a not used nonterminal by prepending a '. Then it builds the rules using the special operator ~~, which when parsing indicates that this concatenation has been automatically built. The second rule ($\langle A' \rangle ::= \alpha_1 \langle A' \rangle \mid \dots$) has epsilon appended. Since the recursive descent parser tries the rules in order, it would succeed on the epsilon rule every time and stop parsing afterwards falsely if the epsilon would be in any other place.
By applying this method to every rule of an input grammar, we receive a non-left recursive equivalent grammar. When parsing however, we receive a bloated syntax tree, since most nonterminals will be duplicated. Therefore we introduce the method `simplify`:

```
def simplify(g: Grammar)(syntaxTree: Tree): Tree = syntaxTree match {
    case Branch(symbol, list) => {
      val list2 = list.filter(x => x match{case Branch(sym, List()) => false; case _ =>
          true})
      var rule = g.lookup(Nonterminal(symbol))
      rule match {
        case Select(_, _) | AutoSequence(_, _) => {
          if(list2.length == 1){
            simplify(g)(list2.head)
          }
          else
            Branch(symbol, list2.map(simplify(g)))
        }
        case t => Branch(symbol, list2.map(simplify(g)))
      }
    }
    case t => t
  }
```

`simplify` takes a grammar and a corresponding syntax tree and tries to simplify the tree. It does this recursively by removing empty lists generated by the epsilon, trimming double symbols by looking for sequences generated by `~~` and removing unneeded symbols like `+`. The latter is specified by the grammar by declaring them as `Comment` instead of `Terminal`. Likewise symbols with a list containing only one element are trimmed if allowed by declaring them as `Select` instead of `Choice`.
Lastly associativity is handled by rotating the syntax tree. Per default all operators are left associative. If you want right associative operators, you have to specify `associativity` and `precedence` when declaring your grammar and later run `transformAssoc` on your syntax tree.
`associativity` should contain a Map which maps all nonterminals to a boolean indicating whether they are left (`false`) or right(`true`) associative. `precedence` should contain a Map which maps all nonterminals to their precedence. Operators with a different precedence will not be affected by each other when doing associativity. The other way around is also true: if you declare `add` and `sub` with the same precedence, a syntax tree of $2 + 3 - 4$ will represent $(2 + 3) - 4$ after applying `transformAssoc`.