LLAnalyzer—LL（1）分析器

BY TANGENTA

2019,1,1

1 介绍

LL（1）分析器能够对一组文法规则进行处理，模拟"自顶而下递归子程序解析方法"的程序执行路线，将函数转换为数据保存到表中。 对于新的句子，可以通过查表的方法完成解析。 另外，LL（1）分析器还具有检测和消除左递归、左公因子，判断是否满足LL（1）文法的功能。

LLAnalyzer使用scala和java结合编写而成。其中scala负责核心逻辑，java负责用户界面。

界面如下：

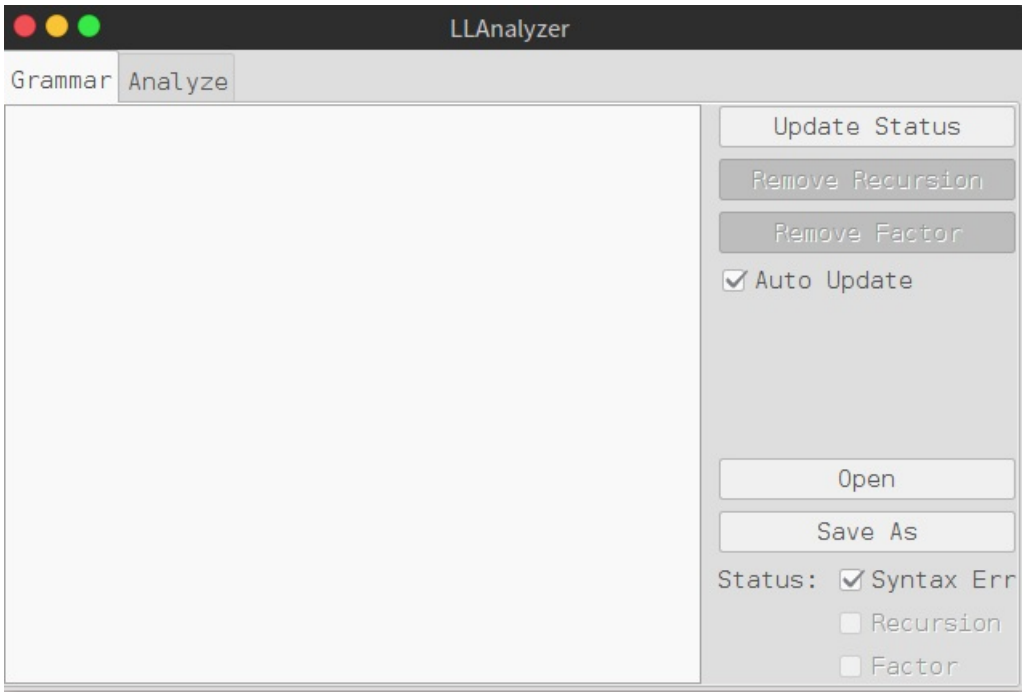

图 1. LLAnalyzer界面一

左边大的编辑框用于显示从键盘输入的文法规则。 当输入完成后，点击右上方的Update Status按钮或者等待2秒（需要勾上右边的Auto Update）即可看到解析文法规则的状态，在右下方显示。状态分别为：

- 该文法规则含有语法错误（即书写格式不正确）
- 该文法规则含有左递归
- 该文法规则含有左公因子

这些选项框反映了文法规则的解析状态，由程序自动完成填写，用户无法修改。 选项框打勾意味着该项成立，例如图1，文法规则含有语法错误。

另外几个按钮分别是消除左递归、消除左公因子（仅当文法规则正确且含有左递归或左公因子时可用）、打开和保存（用于左方的文法规则）。

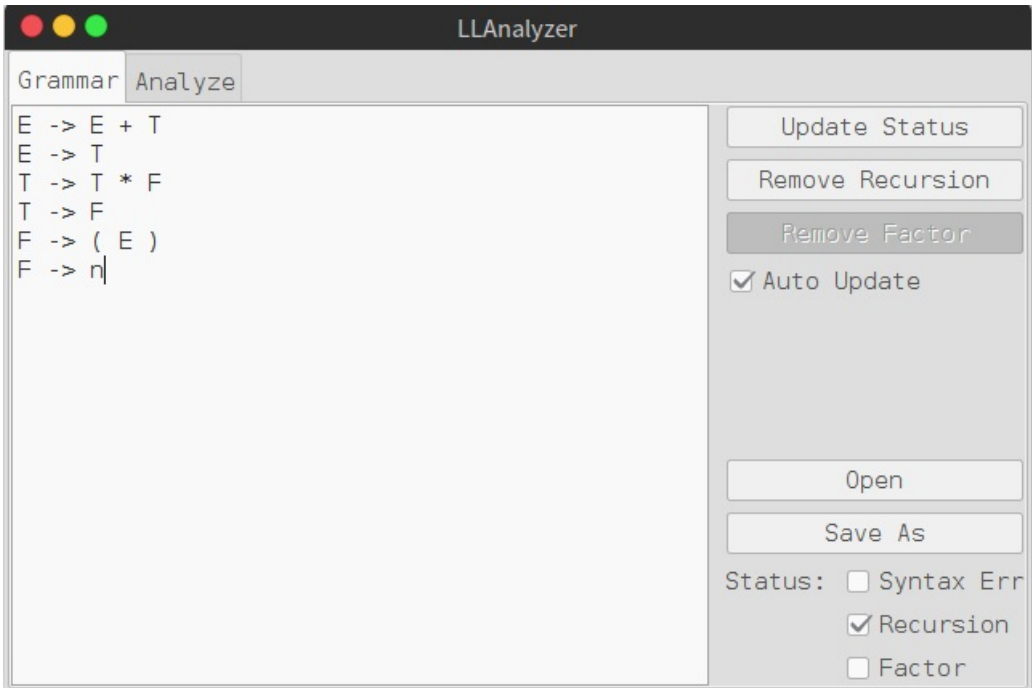

图 2. 输入一段文法规则

输入一段文法规则后，程序自动解析其合法性。 图2右下方的状态显示，该文法规则不含错误，但存在左递归。右上方的"消除左递归"按钮可用。

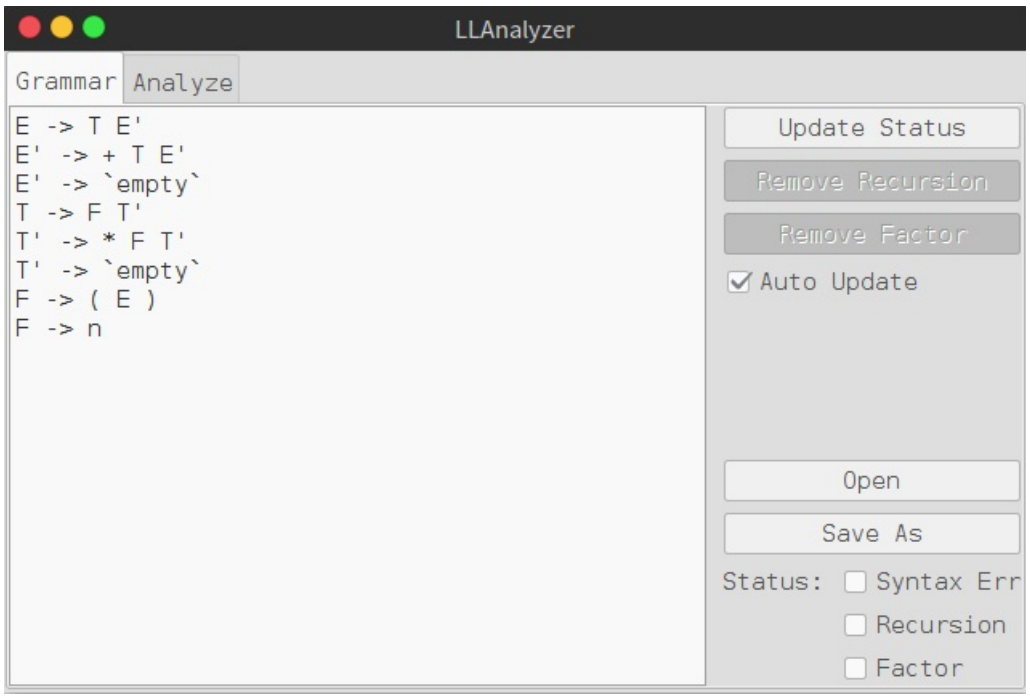

图 3. 点击Remove Recursion

消除左递归后，左方编辑框的内容变成了新的文法规则，它是旧文法规则消除左递归后的版本。

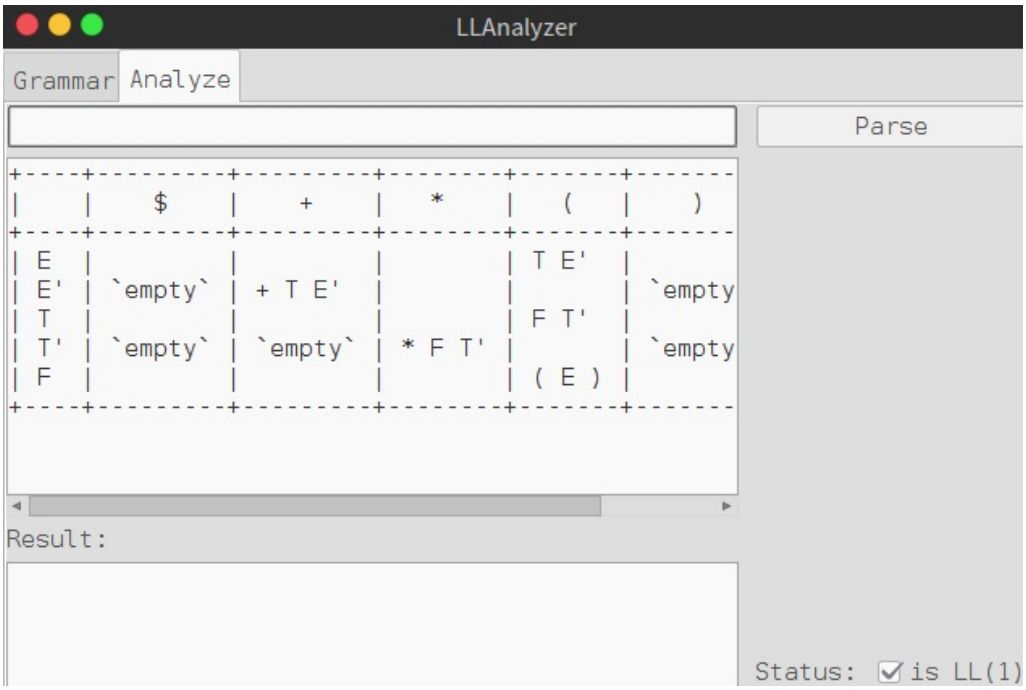

图 4. 查看分析选项卡

点击右边的Analyze选项卡，看到分析表已经显示在上方的文本区域。

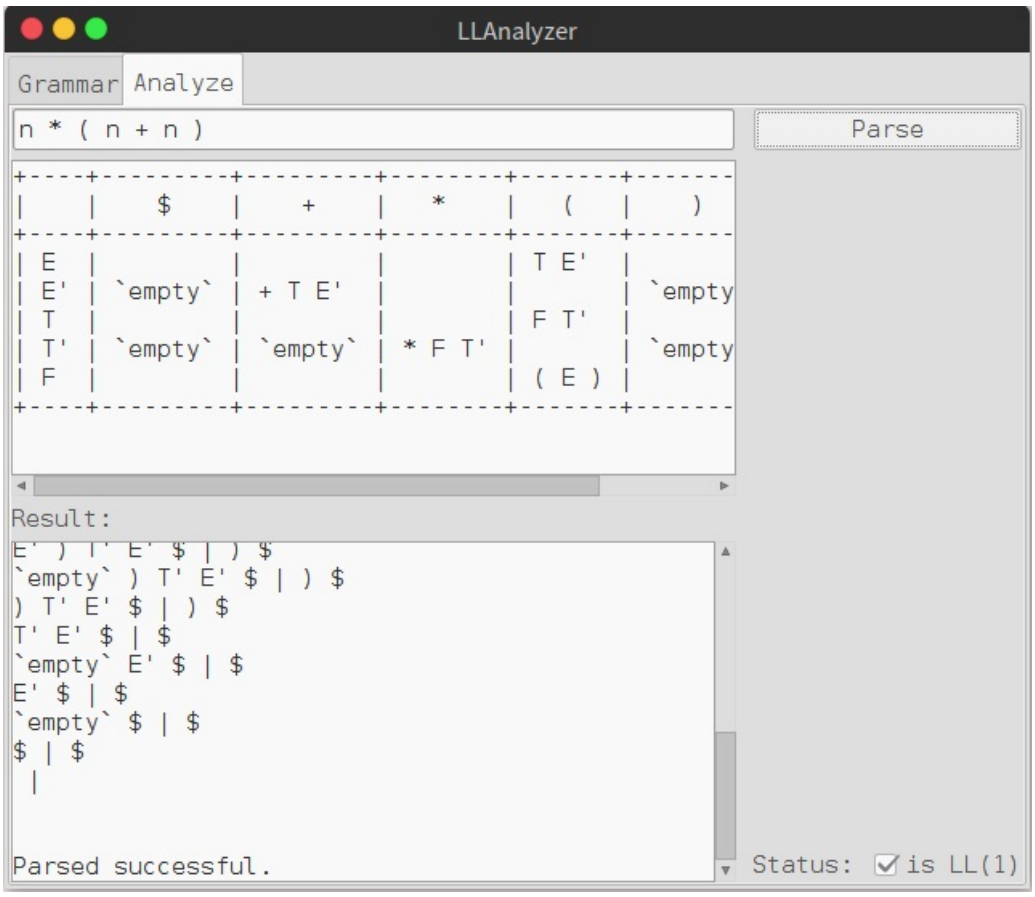

图 5. 解析句子

在上方输入以空格分割的句子，点击Parse，解析过程显示在下方的文本区域。

2 程序假设和约定

a) 程序假设文法规则列表的第一个规则的头部，是开始符号。

b) 程序约定输出用"`empty`"来表示空，书写文法规则时用"empty"表示空。

c) 程序约定输出用"$"标志句子的结束符号。用户书写句子时无需加"$"。

文法规则语法：

- 符号用连续的可显示的非空格文本字符串表示，例如Expr，Term，+等。
- "->"表示"推导"。
- 每条规则占一行。

3 程序结构

scala代码全部位于analyzer包中，共有七个文件，分别是：

- 基本数据结构（Basics.scala），定义了终结符、非终结符、规则等。
- 含有first、follow、分析表字符串化、解析句子等算法集合（Algorithm.scala）。
- LL（1）分析表数据结构（LLTable.scala）。
- 左公因子检测器，附带消除功能（LeftCommonFactorDetector.scala）。
- 左递归检测器，附带消除功能（LeftRecursiveDetector.scala）。
- 文法规则扫描器（GrammarScanner），用于将输入的文法规则字符串变为Basic.scala定义的规则集。
- scala与java之间的适配器， 用于跨语言函数调用。

用户界面定义在LLAnalyzer.java和LLANalyzer.form（窗体）中。

3.1 基本数据结构(Basics.scala)

终结符和非终结符都是对一个字符串的包装，并共同继承自符号类。 规则由一个非终结符作为头部，和一组符号作为主体组合而成。

```
trait Symbol {
  def str: String
}
case class Term(str: String) extends Symbol
case class NonTerm(str: String) extends Symbol
object Empty { val value = Term("`empty`") }
object EOS { val value = Term("$") }

case class Production(head: NonTerm, body: List[Symbol])
```

3.2 First、 Follow等算法集合(Algorithm.scala)

first方法用于寻找某个符号或某组符号第一个可能出现的所有终结符。思路如下：

- 对于单个符号S的First集合，如果是终结符或空，直接返回该符号。 否则，找到规则集以S为头部的规则，计算他们的尾部符号的First集合，再合并到一起作为最终结果。
- 对于一组符号的First集合a1 a2 ... an，计算First(a1)，如果其中包含空，则停止计算。 否则除去空元素，计算First(a2)并入结果，以此类推。 直到First(an)，如果仍然有空，将空加入到结果集合中， 返回结果集合。

follow方法用于查找某个非终结符A出现之后，第一个可能出现的所有终结符。 思路是定位到规则集合右边所有A的出现点，对它们后方的符号串求First集合，最后合并结果。 如果First集合包含空元素，或者后方没有符号，则计算该规则的头部的follow集合，并入到结果中。 基本情况（base case）是follow(S) = {"$"}，S为开始符号。

关于利用文法规则的分析表进行句子解析，放到2.3介绍。

3.3 LL(1)分析表(LLTable.scala)

由于分析表要求根据一个非终结符、 一个终结符即可定位到一条规则，这里在实现上使用映射套映射的方法（Map[NonTerm, Map[Term, Production]]）。 构建LLTable的过程是，对于每条规则，计算右边符号串的first集合（如果first集合包含空或右边符号串本身为空，则要在结果集中加入规则头部的follow集合），在结果中的每个终结符和规则头部非终结符共同定位的表项中填入该项规则即可。关键代码如下：

```
productions.foreach { prod =>
  val firstSet = first(productions, prod.body)
  val terms =
    if (firstSet.contains(Empty.value))
      firstSet - Empty.value ++ follow(productions, prod.head, startingSymbol)
    else firstSet
  val tmap = table.getOrElseUpdate(prod.head, mutable.Map.empty)
  terms.foreach { term =>
    if (tmap.contains(term)) isLL1Grammar = false
    tmap.update(term, prod)
  }
}
```

这里，在填入规则的时候顺便进行了一项检查：该表格是否已经被其他规则占领。 如果一个表项被两条或多条规则填入，该分析表在实际用于分析句子的时候就不能确定使用哪一条规则推导。这时，我们说这类文法不是LL(1)文法。

句子的解析： 句子的解析方法定义在算法集(Algorithm.scala)中，接受一个分析表和一个句子，以及解析监听器，共三个参数，返回布尔值，显示解析是否成功。下面介绍解析方法：

- 每次对一组符号 [S] 和一组终结符 [T] 进行解析。
- 当T长度为0时，表示已经解析完所有终结符，查看S是否为空，返回它的布尔值作为解析结果。
- 如果S中的第一项是空(ε)，忽略该项，继续解析。
- 如果S中第一项（S_1）是终结符，则判断S_1和T_1是否一致：
    - 若一致，则继续解析各自从第二项(即S_2和T_2)开始的符号串。
    - 否则， 返回false值， 表示解析错误。
- 如果S中第一项是非终结符，则在分析表中以[S第一项，T第一项]定位到一条规则P$\rightarrow\boldsymbol{\beta}$，用P的推导式替换S的第一项， 即 $S_1S_2\cdots S_n$ 替换为 $\boldsymbol{\beta}S_2S_3\cdots S_n$ ， 继续解析。

解析初始状态的符号串和句子分别是，

{开始符号，句子输入结束符号}

和

要解析的完整句子++{句子输入结束符号}。

```
  def parse(table: LLTable, terms: List[Term],
              parsingListener: ParsingListener = new ParsingListener): Boolean = {
    def helper(symbols: List[Symbol], restTerms: List[Term]): Boolean = {
      parsingListener.onStepping(symbols, restTerms)
      if (restTerms.isEmpty)
        symbols.isEmpty
      else {
        symbols.head match {
          case empty if empty == Empty.value =>
            helper(symbols.tail, restTerms)
          case t: Term =>
            if (t != restTerms.head) false
            else {
              parsingListener.onMatching(t)
              helper(symbols.tail, restTerms.tail)
            }
          case nt: NonTerm =>
            val optionProd = table.get(nt, restTerms.head)
            optionProd.exists { p =>
              parsingListener.onDeriving(p)
              helper(p.body ::: symbols.tail, restTerms)
            }
        }
      }
    }

    helper(List(table.startingSymbol, EOS.value), terms  ++ List(EOS.value))
  }
```

其中解析监听器用来向外部发送当前的解析状态（包括符号串和句子），包含几个时机：

- onStepping，每一步解析都会触发。
- onMatching，当句子首符号匹配到符号串第一个符号时触发。
- onDeriving， 当符号串第一个符号被替换为推导式时触发。

本程序的句子解析结果部分，就是通过注册解析监听器，并覆盖onStepping来获得当前解析状态的。

3.4 左公因子检测器(LeftCommonFactorDetector.scala)

要验证一组规则中是否存在左公因子很简单：对规则按照头部分组，验证每组内右边第一个符号是否有重复值即可。

要消除左公因子稍复杂一些。首先，左公因子是针对头部相同的两条规则而言的，因此第一步是将规则集合按头部分组，不断提取尾部共有的因子（前缀），直到同一分组的任意两条规则都不含公共前缀即可。提取的步骤如下：

1. 设该组规则的模式为A→*，找到该组内最长拥有共同前缀p的两条或以上的规则，对于这些规则尾部不相同的部分Xi，为它们每一条都新增一条规则A' → Xi，同时原规则改为A → pA'。
2. 重复第一步， 直到没有左公因子。

为了找到组内最长拥有共同前缀的两条或以上的规则，可以先对规则尾部按词法排序（这样可以保证前缀相同的元素相邻），维护最大前缀长度maxLen和前缀内容，遍历时取相邻两个元素，将他们的共同前缀长度与maxLen相比：

- 如果大于maxLen，更新maxLen，记录这两个规则。
- 如果等于maxLen，比较已有记录的前缀内容和当前的内容，如果相同，则并入新的规则；如果不同，则忽略。 一个特殊情况是maxLen = 0且当前两个元素没有共同前缀，这时也要忽略。
- 如果小于maxLen， 说明当前两个元素一定不满足"最长"要求， 忽略即可。

```
protected [analyzer] def longestCommonPrefixProductions(prods: List[Production])
    : (List[Symbol], Set[Production]) = {
  val sortedProds = prods.sortBy(p => p.body.map(_.str).mkString)
  sortedProds.zip(sortedProds.tail).foldLeft(
    (List.empty[Symbol], Set.empty[Production])) {
    case ((ml, res), (l, r)) =>
      val len = l.body.zip(r.body).takeWhile(z => z._1 == z._2).size
      if (len > ml.size)
        (l.body.take(len), Set.empty + l + r)
      else if (len == ml.size && l.body.take(len) == ml && len != 0)
        (ml, res + l + r)
      else (ml, res)
  }
}
```

3.5 左递归检测器(LeftRecursiveDetector.scala)

检测左递归的方法是：计算每个非终结符A的first集合，如果在计算first(A)的过程中又需要计算first(A)，则可以说该规则中A是左递归的。只要存在这样的非终结符，该规则就存在左递归。

实现方法是，在first方法的入口处添加一个容器，用来保存已访问的非终结符。 每次first方法被调用时，检测容器中是否已存在入参符号，就可以判断是否为第二次调用。 由于first算法以两个方法蹦床调用的方式实现，需要覆盖其中一个来修改入口逻辑。

在消除左递归方面，首先是消除直接左递归（下称DR）的方法：在头部相同的一组规则内，设他们为

$$A \to A\beta_1 \mid A\beta_2 \mid \cdots A\beta_m \mid \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$$

消除DR后的规则包括：

$$A \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_n A'$$

$$A' \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_m A'$$

$$A' \to \varepsilon$$

```
private def eliminate(prods: List[Production]): List[Production] = {
  val isRecursive = (p: Production) => p.head == p.body.head
  val partition = prods.partition(isRecursive)
  if (partition._1.isEmpty) partition._2
  else {
    val newNonTerm = NonTerm(prods.head.head.str + '\'')
     partition._2.map(p =>
      Production(p.head, p.body ++ List(newNonTerm))
    ) ++ partition._1.map(p =>
       Production(newNonTerm, p.body.tail ++ List(newNonTerm))
     ) ++ List(
      Production(newNonTerm, List(Empty.value))
    )
  }
}
```

但是，这并没有消除**间接**的左递归（下称IDR）。例如 $A \to B|\varepsilon, B \to Ac$ 两条规则中各自都不存在DR，但计算first(A)的过程中又需要计算first(A)，在自顶而下的递归子程序解析方法中会导致栈溢出。

为了系统地消除左递归，包括DR和IDR，首先要按规则头部排序，对每一组头部相同的规则按排序的先后顺序逐个进行DR的消除。对于某组规则$S \to \alpha$，执行消除DR操作之前，保证排序**先于**S的符号不出现在α的第一个符号即可（目的是破除"计算first过程中"的环状路径，使其无法递归）。这就要用到替换的方法：如果α的第一个符号A排序在S之前，说明A已经被消除过DR，这时可以用A所有的单步推导替换它。替换完后如果第一个符号B排序仍然在S之前，用B所有的单步推导再替换B，依次类推，直到第一个符号排序**后于**S，或者就是S本身，这时正式开始DR的消除。

替换方法代码：

```
private def substitute(source: Set[Production], dest: Set[Production])
  : List[Production] = {
  dest.flatMap { prod =>
    source.map(srcProd =>
      if (prod.body.head == srcProd.head)
        Production(prod.head, srcProd.body ++ prod.body.tail)
      else prod
    )
  }.toList
}
```

系统消除左递归代码：

```
def eliminateLeftRecursion(prods: List[Production]): List[Production] = {
  type ProdGroup = List[Production]
  type PGroupList = List[ProdGroup]
  def helper(result: PGroupList, rest: PGroupList): (PGroupList, PGroupList) = {
    if (rest.isEmpty) (result, rest)
    else {
      val tmp = eliminate(rest.head)
      helper(tmp :: result, rest.tail.map(rest => substitute(tmp.toSet, rest.toSet)))
  // fixme: need optimize
    }
  }
  helper(List.empty, orderedGrouping(prods))._1.reverse.flatten
}
```

这里的算法和上面描述的稍微有些不同。对于某条规则$S \to \alpha$，这里选择先消除DR，再用这条已消除DR的规则去替换后面所有的尾部以S开头的规则。以替换两条规则作为基本运算，二者的时间复杂度都是$O(n^2)$。

还有值得一提的地方是，消除左递归前后，在规则首部符号的相对顺序应保持一致，因为需要维持一个重要的假设：规则列表中第一项规则的首部是文法规则的开始符号。另外，保持一致的相对顺序提高了文法可读性。

3.6 文法规则扫描器(GrammarScanner.scala)

将字符串转换为Production类表示的文法规则。实现方法是按行读取，每行以"->"作为规则头部和尾部的分割，尾部的符号之间再用空格分割。对于不合法的模式，抛出相应的异常。

3.7 适配器(Adapter.scala)

由于"函数"（或"方法"）在java和scala中的地位不一样，scala中的特质 (trait)在java中也没有对应的语法，因此需要一个中转站来描述调用关系。

3.8 用户界面

用户界面根据程序当前的状态，分为"文法规则书写错误"、"文法规则书写正确，但存在左递归或左公因子"、"文法规则书写正确，不存在左递归、左公因子，但不是LL1文法"、"文法规则书写正确，不存在左递归、左公因子，是LL1文法"，在这些状态下，都启用或禁用了部分按钮，只有最后一个状态是能够解析句子的。具体代码见LLAnalyzer.java。