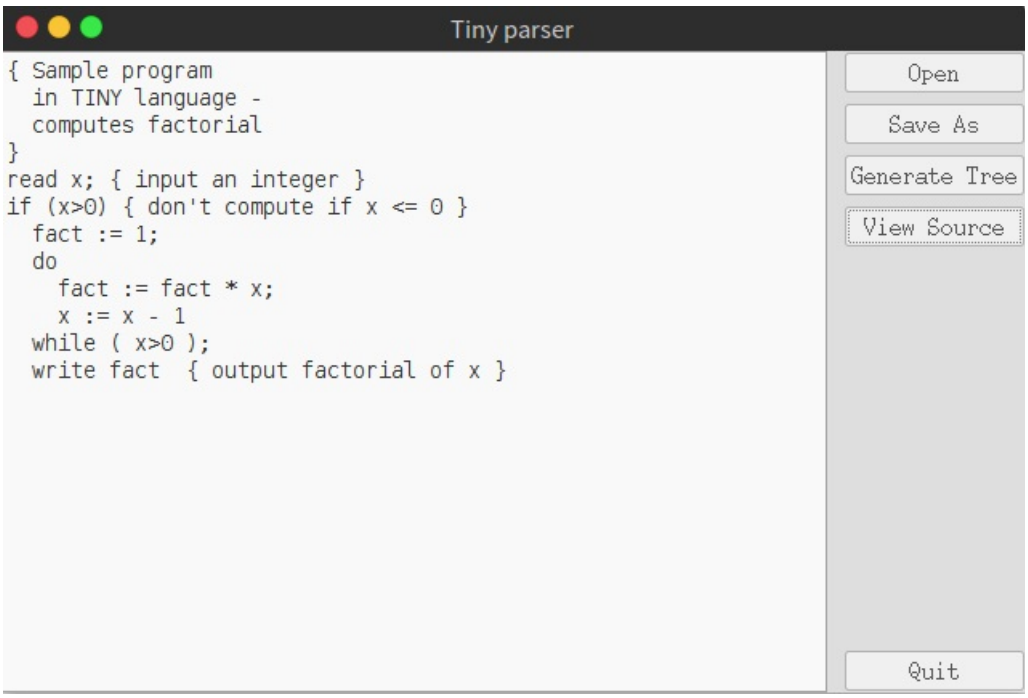
TinyParser

BY TANGENTA

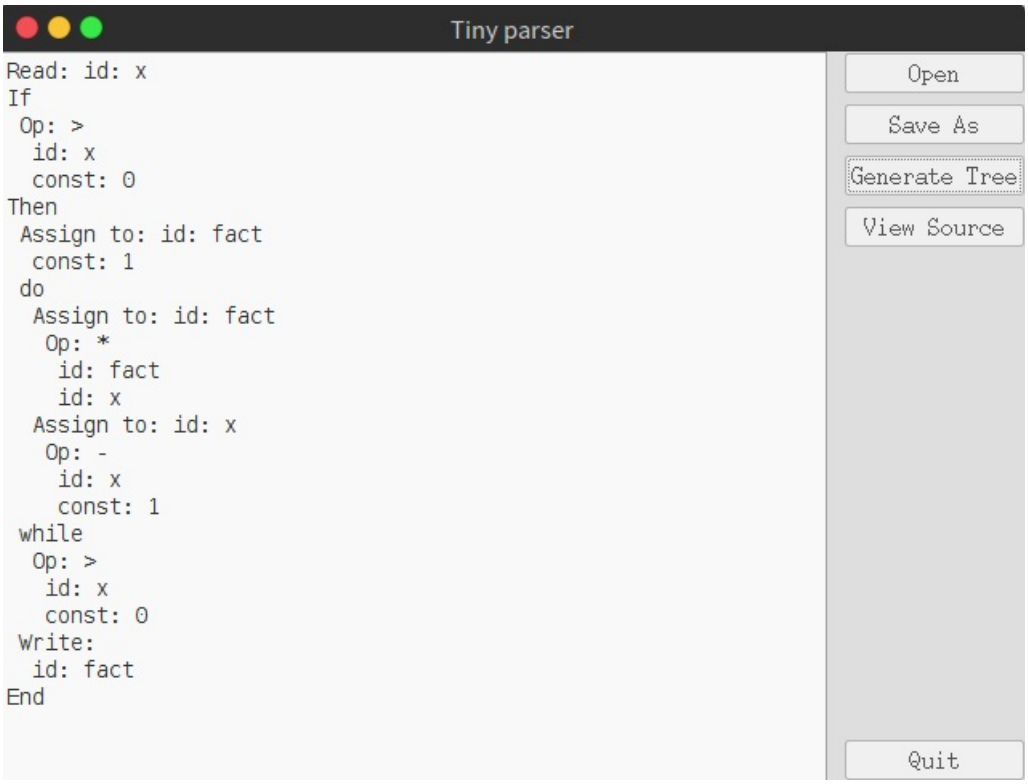
2018,12,2

1 概述

本程序的功能是将Tiny语言代码转换为满足Tiny语法规则的语法树，使用scala和java结合编写而成。其中scala部分负责核心业务逻辑，java部分负责用户界面。



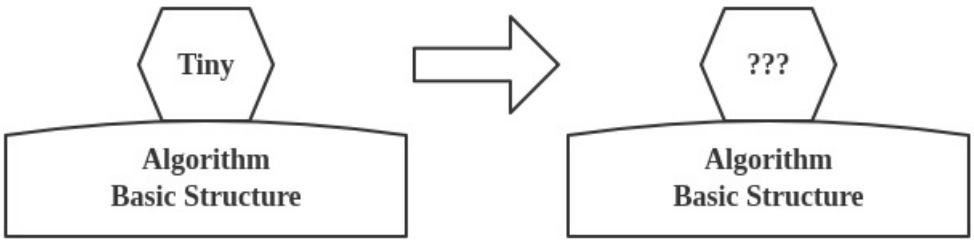
输入Tiny源程序后，点击Generate Tree可以得到解析后的语法树结构。



右方除了有文件的打开、保存功能以外，View Source可以显示上一次解析的源程序。

2 程序结构

本程序分为两个部分，一个是BNF语言通用（language independent）的包grammar，其中定义了基本的数据结构和算法。 另一个是和Tiny语言相关的tiny包，定义了语法规则、扫描器和解释器。



设计的初衷是打算实现一个可拔插的解释器：仅仅通过某个语言的文法规则及其源程序代码片段，构造出相应的语法树，再加上语言特定的语法树输出布局，最终达到输出语法树的效果。这样做的好处是，如果需要修改文法规则，只需要在字面上"修改文法规则"，再增加一些树的输出布局即可。

实现思路是首先构造出LL(1)语法分析表，结合扫描器提供的代码片段建立语法树，最后根据语言规定的语法树布局，将语法树转换为字符串。

2.1 通用数据结构

通用数据结构位于grammar包的Basic.scala文件中，分几个部分：

i. 终结符和非终结符用Terminal和NonTerminal表示，它们只是简单地封装了字符串，并共同继承于抽象基类Symbol（符号）。语法规则用Production表示，包含一个头部（非终结符）和一组符号。

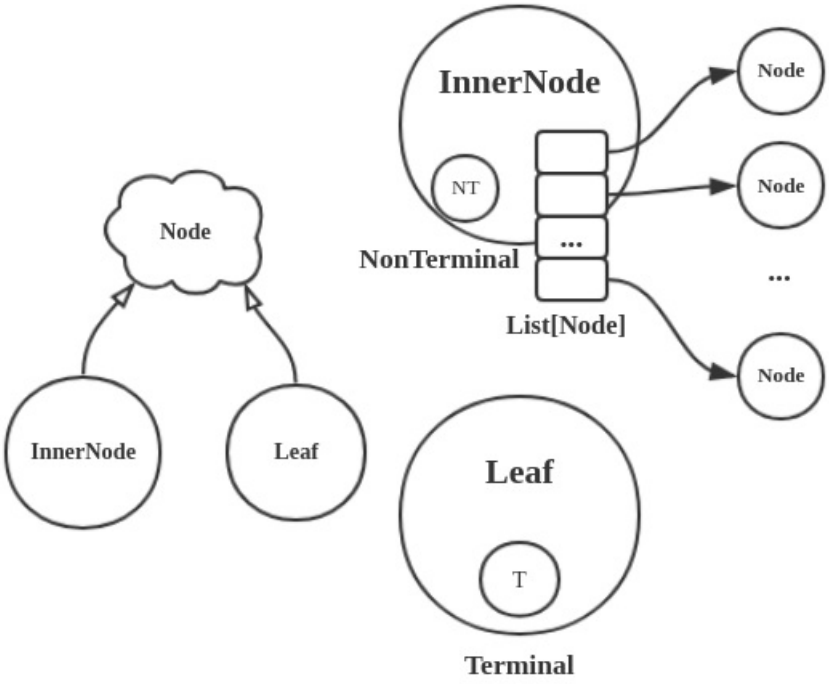
```
case class Production(head: NonTerminal, body: List[Symbol])
```

ii. Token，用来包装终结符和必要的信息，在语言扫描器中可以用到。以tiny语言为例，扫描到单词"factor"的时候，可以包装成Token("identifier","factor")。

iii. LL(1)分析表ParsingTable，它接受一组文法规则作为构造参数，完成构造后可以通过一个非终结符和一个终结符进行存取操作。

iv. 通用语法树结构。有内部节点和叶节点，内部节点可以有多个子节点。由于List可以有1个或多个元素，非常灵活。

```
trait Node
case class InnerNode(symbol: NonTerminal, children: List[Node]) extends Node
case class Leaf(token: Token) extends Node
```



2.2 通用算法

通用算法定义在grammar.Algorithm类中，其中包含了的函数有，用于建立LL(1)分析表的buildParsingTable，用于建立语法树的buildParseTree以及辅助函数first、follow等。

2.2.1 First

First的作用是，计算某个符号中第一个可能出现的终结符。算法如下：

a) 在First(X)中，如果X是终结符，则First(X) = {X}

b) 找到文法规则中所有的头部为X的规则，形式为$X \rightarrow A_1A_2\cdots A_n$，从k = 1开始，逐个计算$A_k$，直到$A_k$不包含空为止，将计算结果做并集，剔除空元素并返回。如果直到A_n仍然包含空，则将所有结果做并集，直接返回。

具体的代码如下：

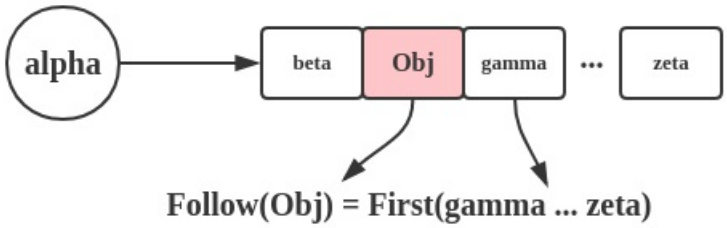
```
def first(productions: List[Production], symbol: Symbol): Set[Terminal] =
```

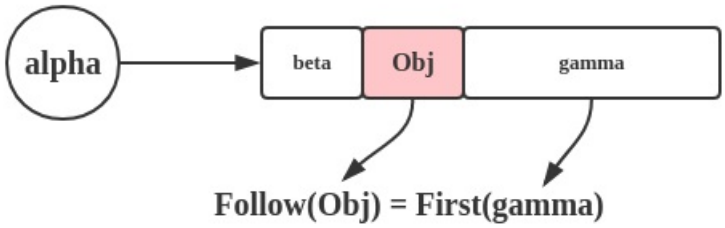
```
    symbol match {
      case term: Terminal => Set(term)
      case nonTerm: NonTerminal =>
        val relatedProductions = productions.toSet.filter(_.head == nonTerm)
        for {
          prod <- relatedProductions
          index = prod.body.indexWhere(!first(productions, _).contains(Terminal("")))
          firstSymbols = if (index == -1) prod.body else prod.body.take(index + 1)
          firstSet = firstSymbols.flatMap(first(productions, _)).toSet
          i <- if (index != -1) firstSet - Terminal("") else firstSet
        } yield i
    }
```

同理可以计算一组符号中的First集合。

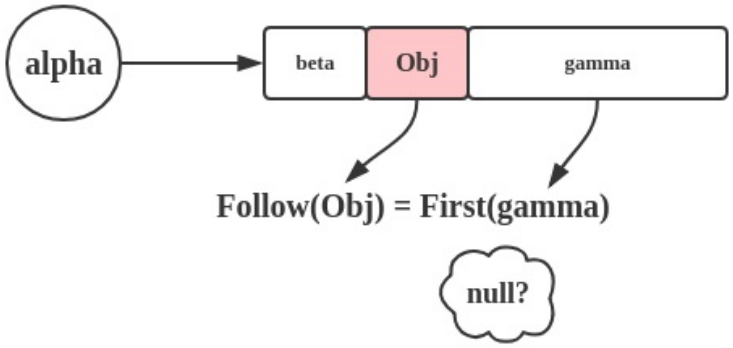
2.2.2 Follow

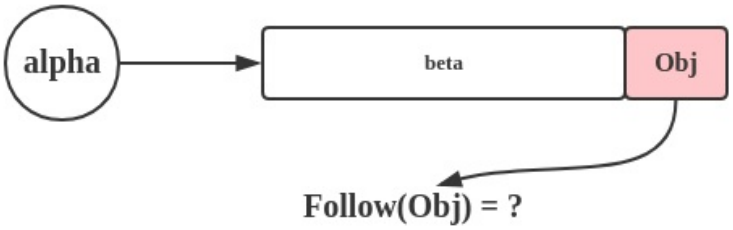
follow(X)的含义是符号X在特定规则中下一个可能出现的终结符的集合。



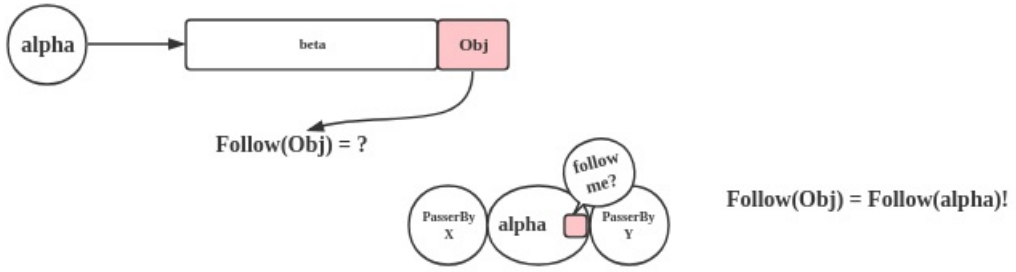


如上图所示，一般情况下Follow计算的是同一条规则中右边所有符号的First集合。 但是当这个First集合为空时，即出现下图所示的情况时:





则需要将该条规则头部的follow集合添加进结果中：




```
def follow(productions: List[Production], symbol: Symbol): Set[Terminal] = {
  def helper(symbo: Symbol, searchedList: Set[Symbol]): Set[Terminal] = {
    if (searchedList.contains(symbo)) Set()
    else {
      val result = for {
        prod <- productions
        index = prod.body.indexOf(symbo)
        if index != -1
        i <- if (index == prod.body.size - 1) helper(prod.head, searchedList + symbo)
        else {
          val firstSet = first(productions, prod.body(index + 1))
          if (firstSet.contains(Terminal("")))
            firstSet - Terminal("") ++ helper(prod.head, searchedList + symbo)
          else firstSet
        }
      } yield i
      if (symbo == productions.head.head) result.toSet + Terminal("$")
      else result.toSet
    }
  }
  helper(symbol, Set())
}
```

注意到在代码实现中，需要添加一个searchedList来记录已经搜索过的非终结符，以免环状调用导致栈溢出。

2.2.3 建立LL(1)分析表

构建分析表的方法很简单，对于每条规则，只要在生成式部分(body)的first集合中的每个终结符nonTerm和规则头部(head)共同定位的分析表元素中填入该规则即可：其含义是，如果在解析的时候遇到了某个终结符，应该使用这条规则来进行推导(derive)。除此之外，如果规则可以直接推导(derive)出空，那该规则头部的follow集合的每个终结符和规则头部共同定位的分析表元素中要填入"空"。

注意，如果分析表中的某个位置被填入了两条（或以上）规则，则说明该语法不是LL(1)语法（向前查看的一个记号不能确定使用哪条规则）。

2.2.4 建立语法分析树

观察语法树的结构，发现每一个符号（symbol）都对应了一个节点，因此可以定义辅助函数来接受符号，返回节点。但是要构造出一个节点，必须先知道它的子节点有哪些，而子节点列表由LL(1)分析表决定，进而需要知道用来定位的终结符。所以辅助函数的第二个参数类型应为剩余的记号输入流。处理完后，将处理过的记号输入流作为第二个返回值，传递出去。于是有了下面辅助函数的函数原型：

```
def helper(symbol: Symbol, rest: List[Token]): (Node, List[Token])
```

其中rest是记号输入流，它的第一个元素，就是这个函数要处理的下一个记号。

在辅助函数中，遇到symbol可能为终结符或非终结符，分情况讨论：

- symbol为终结符时，直接和下一个记号比较：如果相同，说明已找到匹配，构造出Leaf。如果不同，报错。 一种特殊情况是symbol可能为空，这时不处理输入记号，构造出空的Leaf。

- symbol为非终结符时，在分析表中定位到生成式，这个生成式的一组符号就是构建当前节点所需要的所有子节点的信息来源。 这时递归调用辅助函数将这些符号转换为节点即可。注意在转换时要按照从左到右的顺序，记号输入流才能正确反应出"剩余待处理记号"的含义。

综上，写出来的代码如下所示：

```
def helper(symbol: Symbol, rest: List[Token]): (Node, List[Token]) = symbol match {
  case terminal: Terminal =>
    if (terminal == Terminal("")) (Leaf(Token(terminal)), rest)
    else if (terminal == rest.head.terminal) (Leaf(rest.head), rest.tail)
    else throwRTException("unmatched token", terminal, rest.head.terminal)

  case nonTerminal: NonTerminal =>
    val optionalBody = parsingTable.getProd(nonTerminal)(rest.head.terminal)
    if (optionalBody.isEmpty)
      throwRTException("empty table", nonTerminal, rest.head.terminal)
    val body = optionalBody.get

    val (children, restTkn) =
      body.foldLeft((List[Node](), rest)) { case ((nodeList, tokenList), subSymbol) =>
        val tmpResult = helper(subSymbol, tokenList)
        (tmpResult._1 :: nodeList, tmpResult._2)
      }
    (InnerNode(nonTerminal, children.reverse), restTkn)
}
Try(helper(parsingTable.productions.head.head, tokens)._1)
```

通用的算法和数据结构到这里结束。 下面介绍"插件"部分 ——Tiny特有的扫描器、文法规则以及如何利用通用算法和数据结构写一个Tiny解释器。

2.3 Tiny Scanner（扫描器）

tiny包中的Scanner类负责将源程序字符串切割成一个个的记号（Token），其中用到的方法和实验一的词法分析器类似，即逐个读入字符，判断属于哪个类别，转到对应的处理函数。 关键字、标识符和数字的处理方法可参考其正则表达式生成的有限自动状态机。

2.4 Tiny Productions（文法规则）

tiny包的Productions类定义了Tiny语言的文法规则和语法树输出格式，略为臃肿：

```
val tinyProductions: List[Production] = Algorithm.construct(List(
  "program" -> List("stmt-sequence"),
  "stmt-sequence" -> List("statement", ";statement"),
  ";statement" -> List(";", "statement", ";statement"),
  ";statement" -> List(""),
  // ...
  "factor" -> List("number"),
  "factor" -> List("identifier"),
))

// layout code
// ...
```

```
case "while-stmt'' => blank + "While\n'' +
  helper(subtree(1), indent + 1) + "\n'' +
  blank + "do\n'' +
  helper(subtree(3), indent + 1) + "\n'' +
  blank + "endwhile"
case "dowhile-stmt'' => blank + "do\n'' +
  helper(subtree(1), indent + 1) + "\n'' +
  blank + "while\n'' +
  helper(subtree(4), indent + 1) + "\n"
case "for-stmt'' => blank + "for\n'' +
  helper(subtree(1), indent + 1) + "\n'' +
  helper(subtree(2), indent)
// ...
```

对于文法规则，本程序仍然需要手动消除左公因子和左递归，语法树输出布局修改起来不够方便，重复代码多，这也是日后需要改进的地方。

其中最突出的问题是，消除左公因子和左递归不但使语法树的结构变得臃肿，而且在一定程度上破坏了模块性：往往需要多个内部节点来表达消除前的语法结构，在输出布局的实现上带来了极大的不便。以Tiny语法的exp为例，

```
"exp'' -> List("simple-exp", "comp-simp-exp"),
"comp-simp-exp'' -> List("comparision-op", "simple-exp"),
"comp-simp-exp'' -> List(""),
```

消除左递归后引入了comp-simp-exp的中间量。为了输出语法树，当遍历到exp节点时，需要深入到下一层子节点才能决定当前节点的输出布局，这可以说是自动化生成语法树的重大缺陷之一。

2.5 Tiny Parser (解释器)

有了前面提到的这么多工具，就可以组合成一个完整的Tiny解释器了：

```
object Parser {
  def parseIntoTree(code: String): String = {
    try {
     val parsingTable = grammar.Algorithm.buildParsingTable(Productions.tinyProductions)
      val tokens = Scanner.split(code).get
      val root = grammar.Algorithm.buildParseTree(parsingTable, tokens).get

      Productions.buildString(root)
    } catch {
      case e: Exception => "Error: '' + e.getMessage
    }
  }
}
```

建立LL(1)分析表，对源程序分词，组合二者建立语法树，最后用特定的布局转换为字符串。