基于JavaScript的编译算法演示系统

招蕴豪

February 2, 2012

目录

前言

对编译原理的一点感觉

早在两年前，我听说大三有编译原理的课，于是就毫不犹豫地随着减价流买了《编译原理》这本书。这本书一直安安静静地躺在我的书柜上，我好像从来就没有想过主动去翻阅它，生怕它里面深奥的内容会触动我敏感的神经。到了大三，终于得接受这门课的洗礼，这本尘封两年的龙书终于重见天日。

对于编译原理，我有说不出的喜爱。原因有很多，最主要的还是因为我对语言的喜爱。这里的语言偏向于人类语言，我喜欢倾听不同国家的确所特有的韵律，并喜欢观察其语法特点。纵观多种语言，由于中文基本元素（汉字）很多，所以语法结构是比较混乱的。正是因为我们是中国人，我们没有必要过于严谨地去学习中文语法，所以才没有能体会到对于一个完全不会中文的人接触到中文，会是如何地艰辛。当时我就在想，有没有什么办法能真正系统地将这些语法总结出来呢？对于其它语言，我们都可以在书店看到很多相关的语法书，可以说这些语法书在一定程度上总结了该语言总体的语法特点，但是实际上，那是及其不严谨的语法说明，其中存在着诸多特例，以及二义性的描述。但同时我又想，或许正是人类语言中的那些充满二义性的表达，才真正使得人类语言如此迷人。

语法作为指导语言元素使用者能够尽量准确使用语言元素的规则，在人类学习某一门语言的时候充当着及其重要的作用。使得语言初学者能够将其学到的语言元素组合起来，并且能够指引语言学习者能够以一种宏观的角度来观察语言的结构以及性质。从机器的角度而言，语法充当这类似的作用。所以当我第一次接触跟编译相关概念的时候，我非常惊讶，对机器能完成跟人脑类似的工作感到相当神奇。虽然机器所需要识别的语法结构跟人类语言的结构差异巨大，但能向前迈出这样的一步，实在是不容易啊。当然，机器所做的事情比人脑做的事情还多了一些，那就是在理解输入以后还得将其转换为目标代码。

编译原理的课程带给了我很大的快乐，感谢老师允许我用自己的方式来阐述我对编译原理的喜爱以及理解；再有是感谢为编译理论做出过贡献的为人们，你们所

创造出来的知识使我身心愉悦，心情舒畅。

有关本文档

自从大一接触LaTeX之后，我就深深爱上了这个排版系统。无论是我的数学作业还是博客，都是靠它来帮我排版。但由于LaTeX对中文支持比较差，在Archlinux下面配置比较痛苦，所以本来打算用英语来完成这个文档。但考虑到各个方面的原因，还是勉强使用中文来完成。

Chapter 1

系统概述

1.1 动机

由于编译原理这门课程非常重要，能从里面学到的思想有很多，例如流水线以及局部处理局部优化等思想，所以我想更认真地去学习这一门课程。但由于课程内容本身非常抽象，所以我并不能够单单靠阅读书本去理解。虽然用纸笔演算可以解决很多问题，但是当我在演算过程中感受到神奇并不能真切地表现出来，以至于过一段时间以后，我依然对书中的算法感到非常模糊。当然，对算法进行演算无疑是有助于理解，但是由于算法太多，并且复杂，所以有必要借助另外的途径来寻求对其的更深刻的理解。

于是我有了一个想法，那就是将书中的算法实实在在地用程序语言来实现，并找到一种能比较好展示算法演算过程的表达方式展现出来。通过形象的过程描述，或许能使我对算法的流程更为深刻，同时，对算法的实现又可以使我对算法有更深刻的理解。我向来是一个希望究其源而不满足于现象的人，为了这样的目标，我开始了这趟旅途。

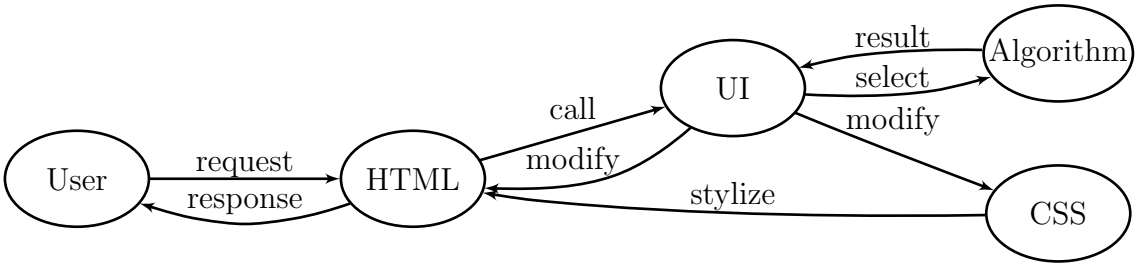
1.2 语言的选择

既然下定了决心，我就开始考虑实现的语言了。由于我的目标并不是去实现一个实实在在的编译器，也不是为了将这些算法的效率提高多少个数量级，我的目标只是为了用一种形象的算法来展示这些算法，所以我没有使用一些比较低级的语言，比如C或者是C++。因为基于这些语言的高效编译器已经有很多了。考虑到我需要的是语言的展示能力，我决定使用JavaScript来完成这个任务。

用JavaScript有一些好处，那就是JavaScript的表达能力比低级语言要强，这样在我实现的过程中就可以忽略掉一些并不需要在算法中关注的细微的现实问题，比如一些低级数据结构的实现；再有，JavaScript可以跟HTML5/CSS3相结合，使得界面的设计更为的便捷与轻松，这样就可以把经历真正放到了对算法的理解上。

1.3 系统结构

这个编译算法演示系统可以分为两个独立的部分，一是核心算法，一是用户界面。而用户界面又可以分为结构，样式以及行为三个部分。鉴于本份文档的任务所在，用户界面并不会花太多的篇幅进行介绍。



从上图可以看出整个演示系统的工作流程。用户向界面发送请求，界面根据用户的输入或者选择来适当调用相应的算法，并将参数传给算法。当相应的算法运算结束后，将结果返回给界面接口，该接口通过修改页面的结构以及样式，来将结果以一种合适的形式来显示给用户，从而完成一次交互。

这种结构实际上是参照网络应用框架的MVC模型，这种模型的好处是容易管理，扩展性强，各部分相互独立，可以单独进行编写以及测试。而整个开发过程则得益于这种模式，可以以增量的方式进行开发，使得基本上没有浪费太多的时间。

Chapter 2

语法分析

语法分析是编译流水线的第二个部分，语法分析器接受词法分析器所提供的词法单元流，根据给定的语法判断词法单元流是否符合语法。语法分析有两种主要的方法，一种是自顶向下语法分析，另一种是自底向上语法分析。在自顶向下语法分析中，语法分析器从语法的开始符号出发，构造一棵词法单元流的语法分析树；而在自底向上的语法分析中，语法分析器根据移入归约原则，将词法单元流转化为语法的初始符号，其过程中的每一步都对应着该词法单元流最右推导的中间过程。

在语法分析部分，对于自顶向下的语法分析，我实现了$LL(1)$预测分析；对于自底向上的语法分析，我实现了SLR以及$LR(1)$。在这些方法的实现过程中，需要到很多的辅助函数，而这些辅助函数都起到了至关重要的作用，下面结合我对语法分析的理解，逐一介绍它们的实现过程。

2.1 语法模型

在语法分析的实现过程中，首当其冲的问题就是为语法选择一个合适的数据结构，一个高效的数据结构非常重要，但高效的数据结构同时又可能难以令人理解。为了均衡高效以及良好阅读性的矛盾，数据结构必须仔细地进行设计。由于JavaScript并没有提供什么数据结构，所以必须自己根据需要来实现。同时，由于数据结构是根据自身来进行实现的，所以比较灵活。下面围绕着书中的表达式文法作为例子，来阐述我的设计过程。一个具有非左递归的文法如下：

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE'|\epsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT'|\epsilon \\
F &\rightarrow (E)|id
\end{aligned}
$$

从上面的语法可以看出，语法最直观的一个模型就是数组，或者是链表，其基本元素是产生式。其实在JavaScript里面，只提供了对象以及数组两个比较高级的数据结构，其实数组也是对象，所以在我的整个系统里面，数据结构基本上都是根据简单的数组组合构建而成的。既然语法是一个产生式的数组，那么产生式又应该如何表示呢？观察下面的产生式：

$$
\underbrace{E'}_{\text{产生式头}} \rightarrow \underbrace{+TE'}_{\text{产生式体1}} | \underbrace{\epsilon}_{\text{产生式体2}}
$$

可以观察到产生式由两个部分组成，一个是产生式头，另一个是产生式体，而由于一个产生式可以有多个产生式体，所以可以用数组来存放一个产生式的所有产生式体。在这里有一个问题是需要注意的，那就是**在这里假定每个非终结符号对应一个产生式的数据结构。**简单地说，就是不会出现下面的结构：

$$
\begin{aligned}
E' &\rightarrow +TE' \\
E' &\rightarrow \epsilon
\end{aligned}
$$

当然，这也是一种合法的语法表示形式，是没有理由禁止的，但为了处理的方便，当用户以这样的方式进行输入的时候，一个称为$reduceRedundancy$的函数会消除这种冗余，即将这样的情况转化为上面的那种用“|”来表示的形式。

当然，上面的描述足以表示产生式以及语法两个抽象概念，但是考虑到运算时有一些额外的量是需要的，将这些量绑定到这两个数据结构有助于提高运算效率。例如，在求$First$和$Follow$集的时候，需要查看语法中的终结符号，所以有必要把终结符集合也绑定到语法的数据结构中；又如，在计算预测分析表的时候，需要

随时用到某一个非终结符的$First$和$Follow$集的结果，所以把这两个集合也绑定到产生式中也是非常重要的。基于这些考虑，可以得到下面的语法数据结构和产生式数据结构的伪代码。

```
Grammar:
  terminals   -> Array(String)     # 终结符集合
  productions -> Array(Production) # 产生式集合

Production:
  head   -> Char             # 产生式头
  bodies -> Array(String)    # 产生式体集合
  first  -> Array(String)    # First集
  follow -> Array(String)    # Follow集
```

2.2 消除冗余以及消除左递归

只要有了上面的模型，即有了解决语法分析问题的工具。我们首要面临的问题就是解决上面所遇到的基本假设，那就是将要进行语法分析的的语法没有冗余，对于自顶向下分析的语法没有左递归。对于消除冗余的问题，解决办法是比较容易的，只要顺序扫描，对于每个产生式头，都检查前面有没有出现过相同的产生式头，如果存在相同的产生式头，将其合并。这样即可消除冗余。最后产生的语法的产生式将是没有重复产生式头的。对于消除左递归，必须先了解下面的立即左递归的消除原理。假设有如下的产生式:

$$A \to A\alpha|\beta$$

对于这样的左递归式，有既定的模式，即递归产生式体以及递归终结符。作为合法的文法，必定要有这两个部分，如果没有了递归部分，那么就不算是递归文法了，如果没有递归终结符，那文法将无法终止，也是不合适的，所以上面的模式描述了递归文法的通用特点。当用这样的文法对句子“$\beta\alpha\cdots\alpha$”进行分析的时候，它会尽量展开，一直展开知道最后遇到“β”，所以说这样的分析是无法预计结束点的，所以得尽早将确定的部分先进行代入解决，这就需要左递归的消除了。**其实在我看来，对左递归的消除就是将左递归转化为右递归。**正是因为语法分析过程是对输入串的从左向右的扫描过程，所以可以正常处理右递归，只要句子是有限的，那么递归就一定可以结束。为了转化为右递归，我们先把终结符号先分析出来，于是得到下面的转换:

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' | \epsilon \end{aligned}$$

将这种情况推广一下，即可得到一种比较通用的情况，即将：

$$A \rightarrow A\alpha_1 | A\alpha_2 | \cdots | A\alpha_m | \beta_1 | \beta_2 | \cdots | \beta_n$$

转化为：

$$\begin{aligned} A &\rightarrow \beta_1 A' | \beta_2 A' | \cdots | \beta_n A' \\ A' &\rightarrow \alpha_1 A' | \alpha_2 A' | \cdots | \alpha_m A' | \epsilon \end{aligned}$$

这样的转化其实在代码上面也不是那么容易实现的，不过主要思想还是一致的。即先对所有产生式进行扫描，对于其中的某一个产生式，再对出现在这一产生式前面的所有产生式进行扫描，用上面给出的通用方法进行左递归的消除，这样等于两次循环递增消除文法中的左递归。当然，在计算的时候有必要将可以用前面产生式替换的后面非终结符号替换掉，这样保证不会产生新的左递归。

2.3 First集与Follow集

毫无疑问，对于文法非终结符号的$First$和$Follow$集的计算属于最基本的计算，所以处理好这两个算法非常重要。这两个函数其实是独立于文法的，即使文法是左递归的，这两个集合也应当可以正常地算出来。

$Fisrt$集被定义为可以从α推导得到的串的首符号的集合，其中α为任意文法串。$First$集的作用可以生动地从其定义看出，即对将要进行分析的动向进行一个判断，这些书中都有描述，这里不再详述。对于$First$集合的计算方式也是递归的，计算的时候有以下三种情况：

1. 如果X是一个终结符号，那么$First(X) = X$。

2. 如果X是一个非终结符号，且$X \to Y_1 Y_2 \cdots Y_k$是一个产生式，其中$k \leq 1$，那么如果对于某个i，a在$First(Y_i)$中且ϵ在所有的$First(Y_1), \cdots, First(Y_{i-1})$中，就把$a$加入到$First(X)$中。

3. 如果$X \to \epsilon$是一个产生式，那么将ϵ也加入到$First(X)$中。

从上面的$First$集计算过程可以看出，这个计算过程是递归的，所以不可能直接就可以按着这个计算步骤进行计算。可以看看下面的特例：

$$\begin{array}{lcll} A & \to & Ba & | \quad \cdots \\ B & \to & Cb & | \quad \cdots \\ C & \to & Ac & | \quad \cdots \end{array}$$

从这个例子结合上面的算法可以看出，如果要计算$First(A)$，那首先得计算$First(B)$，而要计算$Fisrt(B)$，又首先得计算$First(C)$，于是在计算$First(C)$的时候就回归到最开始的问题了，所以不能写一个递归的程序并期待它能自动地给与我们答案。对此必须想一个办法来检测这种“有环”文法。解决办法也很简单，即在每次进行递归的时候，都将当前的状态放到一个寄存状态的地方，而这个地方是可供每一层的递归可见，实现这样的功能可以靠一个简单的状态数组实现。每次将需要求$First$集的非终结符添加进数组，并在每次递归开始之初检测状态数组，看看数组里面有没有这个状态，如果这个状态已经在数组里面了，那说明环已经出现了。要解除这个环的简单方法就是放弃这一次的计算。比如上面的情况，可以直接放弃最后一条产生式的考察，也可以放弃最初要求的计算，因为出现了环，所以只要一端解锁，再算环另一头的时候，就肯定会回到这一头，所以根本不会错过什么计算。

由此可以看一下计算$First$集的伪代码，这个函数所需要的三个参数分别是：

1. $grammar$ 计算这个$First$集所在的语法环境。

2. $nonterminal$ 当前所需要计算$First$集的非终结符。

3. $preHeads$ 这是一个数组，用于记录递归求$First$集过程中所遇到的首个产生式头部。

首先判断传入的非终结符是否为终结符或者空，如果那就是一个终结符或是空，可以直接返回。如果不是的话，就对整个产生式的每个体进行迭代。对于这个产

生式的每个产生式体，又会对其每个符号进行迭代，对每个迭代的符号都递归调用这个函数。然后将返回结果与当前的结果合并。如果返回结果里面有空，那根据$First$集的计算规则，将继续计算下一个符号，否则，就可以直接跳出循环了。

```
function firstSetIter(grammar, nonterminal, preHeads)

  if terminals contains nonterminal:
    return { nonterminal }

  if nonterminal is epsilon:
    return { epsilon }

  result = {}

  for bodies in a production:
    if preHeads contains the head of current body:
      continue

    for symbols in the current body:
      push the current symbol to preHeads
      subFirstSet = firstSetIter(grammar, currentSymbol, prevHeads)
      merge subfirstset with the current result
      if subfirstset does not contains the epsilon:
        break

   return result
```

对于$Follow$集来说，它对预测的作用还是挺大的。$Follow(A)$被定义为可能在某些句型中紧跟在A右边的终结符号集合。这个描述正好是这种情况的一个写照：即当分析进行到某一个时刻，即此时我们的焦点在一个句子的某一个位置，那么我们可以看看紧跟在当前这个位置有可能会出现什么样的终结符号，而这些可能的集合，正是$Follow$集所要描述的东西。书中给出的三个计算规则如下：

1. 把\$放到$Follow(S)$中，其中S是开始符号，而\$是输入右端的结束标记。
2. 如果存在一个产生式$A \rightarrow \alpha B \beta$，那么$First(\beta)$中除了$\epsilon$之外的所有符号都在$Follow(B)$中。

3. 如果存在一个产生式$A \to \alpha B$，或者存在一个产生式$A \to \alpha B\beta$且$First(\beta)$包含ϵ，那么$Follow(A)$中的所有符号都在$Follow(B)$中。

对于程序来说，是可以直接把这三条规则直接翻译为程序语言的，当然为了消去跟在计算$First$集时遇到的循环，我们也同样可以引入一个记录计算$Follow$集的非终结符，具体可以参看下面的伪代码：

```
function followSetIter(grammar, nonterminal, prevHeads)
  result = {}

  if nonterminal is the start symbol:
    merge result and { $ }

  for each production in the grammar:
    for each body in the current production:
      for 0 to pos:
        if nonterminal is not in the production:
          continue
        if nonterminal is at the end of the body and
          prevHeads does not contain the head of the production:
          push current production head to prevHeads
          subfollowset = followsetiter(grammar, production head, prevHeads)
          merge result and subfollowset
        else:
          if the first set of the rest of the body does not
            contains the epsilon:
            merge result with the first set of the rest of the body
          else:
            exclude epsilon out of the first set
            merge result with the first set of the rest of the body
            if prevHeads does not contain the production head:
              subfollowset = followsetiter(grammar, production head, prevHeads)
              merge result and subfollowset

  return result
```

其中的参数含义与$First$集的是一样的。不过在计算$Follow$集的时候，有必要对整个语法都进行遍历，因为仅仅根据一个非终结符的产生式组并不能完全得到

其$Follow$集。函数看起来虽然很长，但是结构性还是挺强的。首先第一个if语句仅仅是简单的异常判断。第二个if是针对规则2的，而$else$则是针对规则3的。其中$else$里面的if是规则3的第一个部分，$else$里面的$else$是规则3的第二部分。

在这里结束$First$与$Follow$集的讨论。由于这两者的计算实在是非常重要，但更为细致的代码讨论会使得整个报告的结构冗长，所以这里也只是简要的提到一下重点，即递归循环的避免，其余的均不难从伪代码翻译为代码。

2.4 LL(1)语法分析

$LL(1)$语法分析是一种自顶向下的语法分析，它的整个过程相当于为一个句子构建一棵语法分析树。从这个角度可以很好的理解二义性的问题，一个具有二义性的文法可以构建出多棵语法分析树，所以具有二义性的文法并不能用$LL(1)$进行语法分析。其实语法分析都是表驱动的，整个分析可以分为两个步骤，一个是建表，而另一个就是根据表来进行分析。

在分析过程中，我们首要面临的问题就是用什么数据结构进行表的表示。在考虑所用的数据结构之前，我们应该思考一下需求。对于查表动作，可以用下面的动作来描述：

$$Action = ActionTable(CurrentState)$$

在这里，我们通过用当前状态来进行查表的输入，然后根据表中查得的结果进行动作的执行。对于没有二义性的语法，每个时刻所要执行的动作都是固定的，所以每个表项里面要么为空，要么就是只有一个动作，所以根据这样的性质来考虑，数据结构可以用$hash$表，或是**字典**（其实字典这样的数据结构实质上也还是一个$hash$表）。在JavaScript里面，数组实际上就是一个对象，而一个对象其实也是一个字典的模拟，所以我直接就使用了JavaScript里面的数组来模拟预测分析表，以致使我可以如下进行表查询：

$$Production = PredictiveTable[Nonterminal][InputSymbol]$$

由此，我们得到预测分析表的数据结构如下：

```
Predictivetable:
  table       -> Array(String) # 预测分析表
```

```
  startSymbol -> String         # 开始符号
  terminals   -> Array(Char)    # 终结符号集
```

对于表的构建，书中列出了两个规则，分别是针对于$First$里面是否含有ϵ的。如果不含，那么直接用$First$集合就可以判断到后面的预测方向，如果有ϵ，那么就需要结合$Follow$进行判断，具体如下：

1. 对于$First(\alpha)$中的每个终结符号α，将$A \to \alpha$加入到$M[A, \alpha]$中。
2. 如果ϵ在$First(\alpha)$中，那么对于$Follow(A)$中的每个终结符号b，将$A \to \alpha$加入到$M[A, b]$中。如果ϵ在$First(\alpha)$中，且\$在$Follow(A)$中，也将$A \to \alpha$加入到$M[A, \$]$中。

这些步骤都非常直观，可以直接翻译为程序语言。看如下伪代码：

```
function generateTable(grammar)
  for productions in grammar:
    for body in current production:
      # 规则2：First集里有epsilon
      if first set of current body contains epsilon:
        for symbol in follow set of current body:
          table[current production head][symbol] = current body
        if epsilon in follow set of current body:
          table[current production head]['$'] = current body
      # 规则1：对于First集里面的其他符号
      for symbol in first set of current body:
        table[current production head][symbol] = body
```

对于伪代码不作过多解释，因为实在是非常直观的翻译。不过这里可以稍微谈一下报错的问题。可以看到上面的伪代码是直接为表项赋值，但可以进一步改进。在为表项赋值之前，可以先看一下表项当前是否为空，如果为空，那么可以直接进行赋值；但如果不为空的话，那么就说明在这个表项所对应的情况下，可以做出不同的动作，这同时说明了语法是具有二义性的。所以在生成表达的时候可以顺便对文法是否具有二义性进行一个判断。并且也能准确地将错误报告出来。而将这个错误显示出来只涉及到UI的问题，所以这里不进行详述。

在建立表之后，就可以进行预测分析。预测分析的伪代码书上有，所以这里就不给出了，只是谈论一下细节的问题。由于在文法或者句子输入的时候，并不会特别地加上一个$终结符，所以这个符号必须要另外处理。可以简单地在分析之前检测一下句子末尾是否有这个符号，如果没有的话就自动地添加一个。而对于分析结果的保存也是需要注意的。由于我的目标是将整个分析过程的结果都输出出来，所以在分析的过程中就需要把结果保存下来。考虑到最后呈现出来的结果是一个表格形式的状态表，所以可以使用HTML的表格来进行记录。每一个状态的各种信息就是表中的一行，将这样的信息转化为格式化字符串，再将其用HTML的表格语句包装起来即可。

这里同样可以考虑报错的问题，由于整个分析过程都是透明的，如果有某个地方出现了错误，都可以清晰地知道。比如在某个查表动作中，表项并没有对应的动作，这样即可判断需要分析的语句并不是该文法的句子。再有，如果给出的终结符或非终结符并不在文法之中，这样在查表的时候根本就不会指到合法的表项中去，这样也可以进行一个报错。这些都是涉及到UI的问题，这里也不再细说。

2.5 项集族

项集族是项集的集合，而项集又是项的集合。项则是文法的一个产生式以及位于产生式体中的一个点组成。个人感觉这个描述是富有想象力的，那个点代表着当前所分析到的位置，随着点的不断往后移动，分析的进程也在向前推进，但是如何去描述一个句子已经分析完成呢？在物体的外部是无法看到物体内部状态的终结，如果将分析器看成是一个黑盒子，这种比喻也是合理的。因为我们无须关注它内部的工作模式及其原理，我们对此仅有的判断力来源于这个黑盒子的入口以及出口，只要一个句子从黑盒子的入口进去，并从其出口成功出来，我们就知道这个句子符合语法了。而在黑盒子的内部，分析自动进行。这种情形实际上是靠增广文法的项来描述的，一个文法描述这个黑盒子的内部工作模式，而增广文法则是相对于我们在盒子外面看到的情况。

有必要找到一种适合描述项的数据结构，由于我们已经有了文法，产生式的结构，所以根本就没必要重新想出一种完全新奇的数据结构来表示。考虑到状态其实就是产生式的某一个位置，所以我们可以直接为**产生式的体**附上一个整数，这个整数所表示的正是当前所分析到的产生式体的位置。所以一个项的数据结构可以如下表示:

```
Item:
  head     -> Char          # 产生式头
  body     -> Array(String) # 产生式体
```

```
position -> Int            # 位置
```

有了项的表示，项集以及项集族就可以用数组来表示了，其元素是低一级的概念。即项集可以用项的数组来表示，而项集族又可以用项集的数组来表示。这里涉及到几个设计上的问题。拿项集来说，项集仅仅是项的一个数组而已，有没有必要将其声明为一个类呢？是需要操作的时候再用一个数组表示项集呢，还是将其声明为一个类，再在必要的时候对这个类里面的数组进行操作？前者代码看起来比较简介，但是比较难懂，而后者则会比较有结构性，但是每次引用项集数组都会比较繁琐。我最后还是选择了后者，即将其声明为一个类。这是因为考虑到两个基本函数$Closure$和$Goto$都是以项集为基本操作单元的，所以可以将这些方法都绑定到项集这个类里面去，这样引用那些方法起来也会比较自然。那项集族呢？按照同样的原因，我也将其单独声明为一个类。

2.6 Closure函数与Goto函数

在这一小节专门讨论$Closure$函数与$Goto$函数。这两个函数都是以项集作为基本单元。其实本质上，这两个函数都是对LR状态机的一个描述，$Closure$函数是状态机某一个状态的扩充，即将一个状态向其所有可能状态进行扩充；而$Goto$函数则是描述状态机状态之间的转移，这两者结合起来，其实就间接地给出了一个文法的LR分析表了，这个后面再说。

$Closure$函数的计算规则也挺简单，书上给出了两点：

1. 一开始，将I中的各个项加入到$Closure(I)$中。
2. 如果$A \to \alpha \cdot B\beta$在$Closure(I)$中，$B \to \gamma$是一个产生式，并且项$B \to \cdot\gamma$不在$Closure(I)$中，就将这个项加入其中。不断应用这个规则，直到没有新项可以加入到$Closure(I)$中为止。

在阐述这两条规则之前，先说一下增广文法，因为这里涉及到增广文法的一个小小的问题。增广文法被定义为：如果G是一个以S为开始符号的文法，那么G的增广文法G'就是在G上加上新开始符号S'和产生式$S' \to S$而得到的文法。这里涉及到增广文法新开始符号的命名问题。如果用户所输入的产生式中，已经有非终结符号S'，那么增广文法再使用这个名字的话会导致非终结符号的重复，而且也出现了不是我们所希望的文法。所以保险的做法是为增广文法的开始符号选择一个与当前文法所有非终结符号都不相同的一个符号，这样的选择有很多，最为简单的一种办法就是将所有的非终结符号连接在一起作为增广文法的一个开始符号，这样做比较稳妥，而且使得增广文法的新开始符号与当前文法的所有非终结符号

都不一样，但是这样挺罗嗦的，所以我直接采用“$S\#$”作为新的开始符号，考虑到用户输入这种非终结符号的机率会比较低。

解决了这个问题以后，计算上面那两个函数以及其他跟增广文法有关的函数的时候就不会有问题了。现在先来看看$Closure$函数。第一条规则相当于一个起始条件，没什么好说的；而第二条规则则负责扩充状态，比如项$A \to \alpha \cdot B\beta$，这说明分析已经过了$\alpha$，现在期望在未来的输入中会看到由$B\beta$推导出来的字串，而由于$B$肯定非空，所以推导出来的字串必定需要以$B$为产生式头的产生式，所以应该将形如$B \to \cdot\gamma$的产生式都加入到项集里面去。那为什么是加入到当前的项集而不是加入到下一个项集中去呢？由于$Goto$函数的计算是跟当前项集以及紧接着的下一个输入符号有关，当前所预测的可能性是通过紧接着的输入符号来确定转移的方向，比如当前发现有这个可能性，那得靠下一个输入来确定这个可能性是否起作用了，所以这些期望用到的项应该放到当前的项集中去。

$Closure$函数的计算没有太多特别的地方，书中给出了伪代码，在实现中也没有太多特别的问题。

$Goto$函数指明状态机的状态转移线路，对于项集中的每个产生式，以状态点所在位置后面的符号作为输入符号，求出转移的项集以后，记录转移项集的编号即可。由于书中没有给出伪代码，所以请参见下面的伪代码：

```
function goto(grammar, symbol, isLR_1):
  itemSet = {}

  if symbol is invalid
    return itemSet

  for item in items:
    if item.position is valid and
       symbol ==  item.nextSymbol:
      itemSet.push(item)

  for item in itemSet:
    item.position++

  if isLR_1: itemSet.lr_1Closure(grammar)
  else:      itemSet.closure(grammar)

  return itemSet
```

这里代码需要说明一下。其中的一个变量$isLR_1$是用来指明文法是否为$LR(1)$文法，由于$LR(1)$文法跟SLR文法的基本区别就在于$LR(1)$文法比SLR文法多看了一个输入符号，而在其他方面都非常的相似，甚至他们的$Goto$函数结构都是基本相同的，所以这个函数没有必要写两次。所以加入了一个标记来指明这种区别。从代码可以看出，除了最后一步以外，SLR文法的$Goto$函数和$LR(1)$文法的$Goto$函数执行过程是一样的。

有了$Closure$函数和$Goto$函数的帮助，就可以求出一个文法的规范$LR(0)$项集族了。在求项集族的过程中，生成了一个$Goto$转移表，这个表描述了项集族中状态之间的转移情况，这一个算法同样使用一个$isLR_1$的标记位指明文法是否为$LR(1)$文法，对于不同的文法使用不同的闭包函数。

2.7 SLR文法与LR(1)文法

这两种文法同属于自底向上的文法，于自顶向下文法不同的是，自底向上文法采用移入归约的策略，在整个分析过程中的每一个步骤所得到的结果，都是该句子最右分析的一个中间结果，即一个最右句型。在面对输入符号的时候，执行移入或是归约的决定是通过查语法分析表知道的，这也说明了这两个文法的分析也是表驱动的分析，所以关键在于构造这两个文法的语法分析表。

由于这两个文法非常相似，所以只需要用一个函数就能实现这两种分析。这里涉及到一个设计模式的问题。方案一可以是为这两个语法分析实现两套函数；方案二是找出其共同点作为一个函数模板，然后在把不同点分路处理。前者能够使得代码更为的明了，以及模块化更为清晰，但是却导致很多代码冗余，所以我最后还是直接在一个函数里面进行分路处理，没有将其分开。以下给出所生成表的算法:

```
function generateTable(grammar, itemSetCollection, isLR_1):
  for itemSet in itemSetCollection:
    for item in itemSet:
      switch(case)
        case: S# -> S
          generateAction('$', 'a')
        case: A -> x.ab
          generateAction(symbol, 's', nextState)
        case: A -> a.
          if isLR_1:
            generateAction(item.next, 'r', item)
```

```
        else:
          generateAction(follow, 'r', item)

    for production in productions:
      generateGoto()
```

这个函数结构性尚算良好，前半部分是生成$Action$表，后半部分是产生$Goto$表，两部分结合起来形成语法分析表。其中表里面的字母标记分别表示：

1. a：接受状态
2. s：移入状态
3. r：归约状态

其中移入后面所跟的数字代表状态的编号，而归约后面所跟的数字代表用该编号的产生式进行归约。补充说一句，那就是这里的语法分析表跟$LL(1)$文法分析里面所用到的预测分析表的数据结构是一样的。都是用了字典的结构，所以在那些没有表项的地方，是空的。而对于有表项的地方，需要查询的时候也可以在$O(1)$的时间内做到。

Chapter 3

词法分析

词法分析跟语法分析有很多相同的地方，工作在不同的抽象层次，所以有很多地方可以进行类比理解。其一：在词法分析中，分析的基本单元是字母表里面的符号；而在语法分析中，分析的基本单元是词法分析所向上提供的词法单元，可以知道，词法单元是由字母表中的字母所组成的，这说明语法分析工作在比词法分析更高的层次上。其二：在词法分析中，对于分析机理的描述采用的是正则表达式，而在语法分析中，对分析机理的描述是采用语法描述，这两者的本质其实都是状态机，词法分析中，状态转移接受字母表的基本符号作为输入，而在语法分析中，状态转移接受词法单元作为输入，它们的本质都是一样的，只不过工作在不同抽象层次上面；其三：在词法分析中，分析的目标是得出词法单元，而在语法分析中，分析的目标是句子，而句子正是由词法单元组成，再一次说明工作所在的抽象层次不一样。由此我们可以对比着对两者进行理解，从而找到其共性。

在词法分析中，输入是字符串，而输出是词法单元流。有两种主要的分析流程可以达到这样的目的，其中第一种为：

1. 为中序正则表达式添加连接符。
2. 将中序正则表达式转化为后序正则表达式。
3. 根据后序正则表达式建立NFA。
4. 将NFA转化为DFA。
5. 将DFA的状态最小化。

第二种方式为：

1. 为中序正则表达式添加连接符。
2. 将中序正则表达式转化为后序正则表达式。
3. 根据后序正则表达式建立抽象语法树。
4. 计算抽象语法树的$nullabe$，$firstpos$，$lastpos$以及$followpos$。
5. 根据得出的数据直接构造DFA。

由上面的步骤可以看出，两种方法都必须要先对正则表达式进行处理，鉴于时间关系，我没能把两种方法都实现，于是只实现了第一种方法。

3.1 正则表达式

正则表达式的实现是一个复杂的问题。在JavaScript里面，有现成的正则表达式工具，但基于我这次项目的目的，我打算自己实现正则表达式。在实现的过程中遇到了不少的问题。

首先，正则表达式分为基本正则表达式以及扩展正则表达式。在不同编程语言里面所提供的正则表达式工具都是强大的扩展正则表达式，其中具有很多的操作符以及诸多强大的功能。而基本的正则表达式只有三种操作:

Operations	**Definitions**
$Union$	$L \cup M = \{s \| s \in L \| s \in M\}$
$concatenation$	$LM = \{st \| s \in L \&\& s \in M\}$
$Kleeneclosure$	$L^* = \cup_{i=0}^{\infty} L^i$

这三种操作已经可以完全表示扩展表达式中的其他操作，所以为了实现上的简洁，我实现的是基本正则表达式。

在实现过程中遇到的另外一个重要的问题是连接符的问题，由于在输入正则表达式的时候是不会显式输入连接符，这给正则表达式的分析带来了很大的困难。比如考虑正则表达式$(a|b)*abb$，如果直接对其进行中序转后序的操作，我们得到的将会是$ab|*abb$，然而实际上这样做是不行的，因为连接操作并没有在这里体现出来，如果就这样直接进行NFA的构建，那么将会得到一个错误的自动机。解决的办法有两个，一个是在中序正则表达式加上连接符以后再转化为后序正则表达式；二是在中序正则表达式转化为后序正则表达式之后再添加连接符；考虑到括号为我们提供的诸多信息，我的实现是先为中序的正则表达式添加上连接符，

如上面的正则表达式，添加上连接符以后变成：$(a|b)* \sim a \sim b \sim b$（这里假设用$\sim$来表示连接符）。这样处理以后，转化出来的后序正则表达式为：$ab|*a \sim b \sim b \sim$，这才是正确的后序转换，其后构建状态机才不会出错。

解决了这个问题以后，下一个问题随即出现：应该在什么地方添加连接符呢？之前正是为了这个原因才选择基本正则表达式的，其可能出现的情况比较少，下面可以来穷举一下几种需要添加连接符的情况：

$$
\begin{array}{rcl}
Char \quad Char & \rightarrow & Char \sim Char \\
Char \quad (& \rightarrow & Char \sim (\\
) \quad Char & \rightarrow &) \sim Char \\
* \quad Char & \rightarrow & * \sim Char \\
* \quad (& \rightarrow & * \sim (\\
) \quad (& \rightarrow &) \sim (\\
any \quad endmarker & \rightarrow & any \sim endmarker
\end{array}
$$

上面的情况应该也很容易理解，就拿第一种情况来说，如果两个字母表中的字符粘连在一起，那么它们之间必定有一个连接符，因为没有0元操作符可以连接两个字符。又比如右括号和字符之间必定有连接符，右括号预示着一个部分的结束，它后面除非是结束，不然一定需要一个左结合的运算符将其与后面的部分连接，所以这种情况也必须要添加连接符。其余的如此类推。

解决了这些问题以后，就可以思考正则表达式的数据结构了。首先正则表达式是一个字符串，故可以用字符串来表示正则表达式，考虑到正则表达式需要时时查询字母表，所以有必要把字母表也记录下来，故正则表达式的数据结构可以简单地表示为：

```
ReExpression:
    reExp    -> Array(Char) # 正则表达式
    alphabet -> Array(Char) # 正则表达式所表示语言的字母表
```

有了正则表达式的基本模型以后，可以根据上面的讨论很轻易地写出添加连接符的$insertConcatenation$函数，其伪代码如下：

```
function insertConcatenation:
  left  = reExp[0]
  right = reExp[1]

  while (right != endmarker):
    if left and right satisfy the conditions above:
      reExp.insert('~')

    left moves forward
    right moves forward
```

这里在实现的时候有一个问题需要注意，那就是插入连接符的动作其实是对原字符串进行了修改，如果整个过程都在原字符串上面操作，那么在每次插入之后都必须显式将下标向前移动，以配合新插入的连接符。如果上面的步骤是在一个全新的字符串上面进行操作，那么就不会出现下标错位的问题。

3.2 中序表达式与后序表达式

在处理了连接符的问题以后，就可以开始考虑怎样把中序表达式转化为后序表达式了。首先必须明白为什么要将中序表达式转化为后序表达式。因为中序表达式具有二义性，而后序表达式没有，再者，在中序表达式中，括号仅仅起着分组的作用，并没有任何的运算功能，所以没有必要去保留。在后序表达式中，运算符均在其相关操作数的后方，所以可以直接根据操作符号的元数来进行操作数个数的选取。对于这一点来说，中序表达式是不能做到的，因为在运算符后方仍有与其相关的操作数。

了解了中序表达式到后序表达式的重要性，我们还得明白整个转换过程的机理。由于中序表达式具有二义性，所以在转换的时候需要作出一些规定。看如下算术表达式：$a + b \times c$，其转化为后序表达式以后可以有两种形式：

1. $ab + c\times$
2. $abc \times +$

可以看到，对于第一种转换方法，是先将a于b相加，然后再与c相乘，而对于第二种转换方法，则是先将b于c相乘，然后再与a相加，这样就说明了转换为后序表达

式的时候会对运算符的优先级有所依赖，由此，在转换的时候需要根据运算符的优先级进行转换，使得转换出来的结果符合我们所需要的语义。那上面的例子来说，即我们需要的结果是第二种转换方式。

根据正则表达式的语义，我们可以人为表示出各个运算符的优先级，具体看下表:

Operator	**Precedency**
(	0
)	0
*	9
~	8
|	7

其实这个优先级的数字并没有太多的含义，只要数字之间能够体现其优先级的先后关系即可，这里对数字的选取其实也为了后面对更多运算符的添加预留了空间，所以才从7开始。之所以将括号的优先级设到最低，是因为它们仅仅是为了分组而已，并没有运算功能。下面给出中序表达式转成后序表达式的伪代码:

```
function toPosfix(precedencyTable, dimensionTable):
  opStack = {}
  postfix = {}

  while True:
    # 正则表达式的末尾
    if symbol is endmarker:
      while opStack is not empty:
        postfix.push(opStack.pop())
      break

    # 遇到左括号
    if symbol is ‘‘(’’:
      opStack.push(symbol)

    # 遇到右括号
    else if symbol is ‘‘)’’:
      while !opStack.empty and opStack.top != ‘‘(’’:
        postfix.push(opStack.pop)
```

```
      opStack.pop

    # 遇到非括号的操作符
    else if symbol is operator:
      while !opStack.empty          and
            opStack.top is operator and
            opStack.top.precedency >= symbol.precedency:
        if symbol is not bracket:
          opStack.push(symbol)

    # 遇到操作数或字符
    else:
      postfix.push(symbol)

  return postfix
```

可以看到，在转换的过程中有多个情况处理。当遇到正则表达式的末尾时，应当把当前所有栈中残留的操作符都顺序抛出，以确保不会有操作符遗留在操作符栈中。当遇到左括号的时候，现将其压进栈中，等待与右括号的匹配。当遇到右括号的时候，应该抛出与它匹配的左括号与它之间的所有操作符，这相当于处理完一个组。当遇到其他操作符的时候，根据优先级进行处理，如果栈中有优先级比当前运算符高的，将其抛出，这避免了二义性，抛出完以后，再将当前的操作符压栈，这样也确保如果将来有优先级比它高的运算符，可以在更位于栈顶的地方。如果遇到的是操作数，那么可以直接将其输出到后续表达式中。

3.3 状态机图结构

理论上在得到后序正则表达式以后即可进行NFA的构建，但在此先用一个小节来谈谈NFA的结构。NFA与DFA实际上都是图，其状态对应图的节点，其状态的转移对应于图的边，所以必须确定一种图结构来描述状态机，才能够使得状态机的建立尽可能的方便。

考虑NFA，每个状态用状态编号唯一标记，并且可以作为多条边的起始节点；对于状态的转移，可以用一个三元组来标记，即起始状态，触发转移的输入符号，以及结束状态。这样已经可以初步确立状态以及边的数据结构，看下面的伪代码：

```
NFAState:
  id    -> Int             # 唯一标识状态的整型
  edges -> Array(NFAEdge) # 以该状态为起始状态的边的集合

NFAEdge:
  prev  -> Int  # 起始状态的id
  next  -> Int  # 结束状态的id
  input -> Char # 触发转移的输入字符
```

这里要辨析一下起始状态以及结束状态的概念。这里的起始状态并不是指整个状态机的起始状态，结束状态亦然。这两个概念仅仅针对于状态机中的一条边所关联的两个状态而已。既然状态节点以及状态转移的边的数据结构都有了，那么很自然的就可以得出图的结构：

```
Graph:
  begin  -> NFAState        # 图的开始状态
  end    -> Array(NFAState) # 图的结束状态集合
  states -> Array(NFAState) # 图的状态集合
```

图的开始状态有且仅有一个，而结束状态可以有多个，从上面可以看出，图的结构确实可以非常简单。如果要将这些子图连接在一起，那么可以为两个图之间的某些状态添加边。但这里涉及到一个状态编号唯一性的问题。如果有两个图，其中一个状态编号为1到9，而另一个图的状态编号为1到7，那么这两个图合并以后，状态的编号就不唯一了，显然这是不行的。

解决的办法也很容易，我另外写了一个ID分配器，这个分配器是将标识号递增分配的，所以每次获得的标识号都不一样。当然，前提是每次分配都使用同一个分配器的实例。这个分配器代码简单，贴上来也不妨：

```
function IDGenerator(begin) {
  this.id = begin || 0;
}

IDGenerator.prototype.nextID = function() {
  return this.id++;
```

```
}

IDGenerator.prototype.currID = function() {
  return this.id;
}

IDGenerator.prototype.resetID = function() {
  this.id = 0;
}
```

至此以万事俱备，可以进入到NFA的构建了。

3.4 NFA的构建

对于NFA的构建，我才用的是效率较低，但非常直观的$McMaughton\text{-}Yamada\text{-}Thompson$算法。这种算法是一个递归的构建NFA方法，从细致处出发，慢慢构建成庞大的NFA。由于书中对算法进行了详细的描述，这里就不细说了。下面谈谈实现的问题。先看看伪代码:

```
function NFAConstruction(beginID, postfix):
  nfa1, nfa2, begin, end, edge
  nfaStack = {}
  id = new IDGenerator(begin || 0)

  for symbol in postfix:
    begin = new NFAState(id.next)
    end   = new NFAState(id.next)

    if symbol is not operator:
      begin.addEdge(begin, end, symbol)

    else:
      switch symbol:
        case ‘‘|’’: generate ‘‘|’’ NFA
        case ‘‘~’’: generate ‘‘~’’ NFA
        case ‘‘*’’: generate ‘‘*’’ NFA
```

```
nfaStack.push(begin, end)
```

首先这一个NFA的构建函数接受两个参数（真正的代码会有所不同），其中一个是$beginID$，另一个是$postfix$。$beginID$在现阶段是没有作用的，等到后面构建词法分析器的时候，需要将各个正则表达式的NFA合并的时候，这个参数能够指明在NFA构建的时候，选取的第一个标识号为多少，通过这样指定，能够使得不同NFA里面的每个状态的标识号都不会有重复。至于第二个参数，即后序的正则表达式。

整个算法对后序正则表达式里面的每个符号进行扫描，在每一轮循环里面，先判断符号是不是操作符，即$\sim$，|或者$*$，如果不是的话，那么直接为开始状态添加一条以该符号作为触发符号的转移边。如果当前符号是操作符的话，那么就意味着此时需要对过去的子NFA进行组合构建了。这样的子NFA合并跟操作符的操作数有关，对于|和$\sim$，会需要到$nfa1$和$nfa2$，而由于$*$是一元操作符，所以只需要到$nfa1$。

上面对子NFA的组合进行了非常简略的描述，仅仅用了伪代码“$generate\quad symbol\quad NFA$”来表示，下面以操作符|为例阐述一下合并的过程。