基于JavaScript的编译算法演示系统

招蕴豪

January 30, 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。

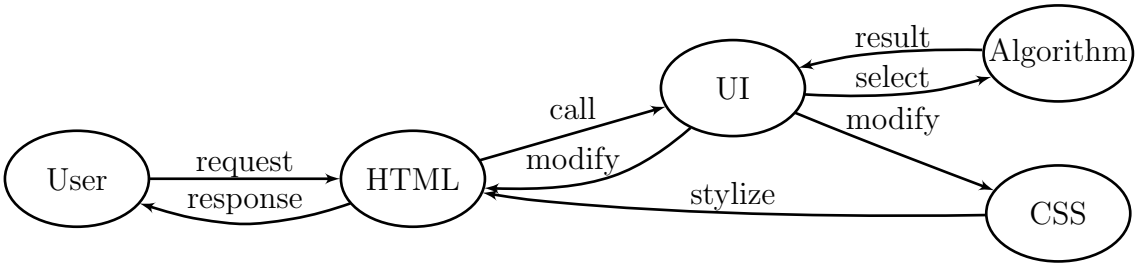
1.2 语言的选择

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。