# 第 5 章　语法分析

语法分析是指对给定的符号串，判定其是否是某文法的句子。通俗讲，就是判定用户源程序中有无语法错误。给定一个用户源程序，通过第四章词法分析方法能够得到源程序对应的单词 Token 串（即符号串，每一个单词 Token 可看作一个符号），语法分析的目标是分析该单词 Token 串是否是某高级程序设计语言文法的句子，即单词 Token 串是否满足某高级程序设计语言文法。同时，在语法分析的过程中，我们还可以得到用户源程序（单词 Token 串）的结构，即语法树。

基于文法的语法分析，主要有两类方法，一种是自顶向下的，即基于文法推导的方法，一种是自底向上的，即基于文法进行规约的方法。这两种方法中包含多种算法，在本书中，将介绍自顶向下算法包括递归下降子程序法以及 LL(1) 方法，自底向上算法包括 LR() 分析法以及简单优先分析法。

这一章的内容包括语法分析的基本概念，以及常用语法分析方法的设计与实现。

## 5.1　语法分析的基本概念

给定一个符号串 α 和一个文法 G(Z)，语法分析用于判定 α 是否是文法 G(Z) 的句子。基于文法的语法分析，主要有两种方法，一种是自顶向下的，即基于文法推导的方法，一种是自底向上的，即基于文法进行规约的方法。语法分析定义如下：

**定义 5.1** 语法分析是指对给定的符号串 α，判定其是否是某文法 G(Z) 的句子。即

1. 自顶向下法（推导法）

自顶向下法是从文法的开始符号出发，自顶向下构造语法树，不断地使用文法规则进行推导，即用文法规则的右部替换左部的非终结符，最终试图使树叶全体恰好是给定的符号串 α，即对给定的符号串 α，判定其是否存在 $Z \xrightarrow{+} \alpha$；其中：Z 是开始符号。我们约定，当一个句型中有多个非终结符可以进行推导替换时，优先替换最左非终结符，即习惯上采用“最左推导法”。

**例 5.1** 给定文法 G(E)，给定一个符号串：α=a*(b+c)，判断 α 是否是文法的合法句子？

文法定义

$$E \to T \mid E+T$$
$$T \to F \mid T*F$$
$$F \to i \mid (E)$$

采用自顶向下的语法分析思路如下：

1. 分析过程：$E \Rightarrow T \Rightarrow T*F \Rightarrow F*F \Rightarrow a*F \Rightarrow a*(E) \Rightarrow a*(E+T) \Rightarrow a*(T+T) \Rightarrow a*(F+T) \Rightarrow$

图 5.1: 基于推导的语法树

图 5.2: 基于规约的语法树

$a*(b+T) \Rightarrow a*(b+F) \Rightarrow a*(b+c)$

2. 构造语法树

从上述分析过程不难看出，若分析的第一步不是选择 E→T，而是选择 E→E+T，则将导致分析失败。可见，对于自顶向下的算法，主要解决的是当同一个非终结符有多个产生式时，如何选择正确的产生式进行推导，即自顶向下分析的关键技术是如何确定具有相同左部的产生式之侯选者。

2. 自底向上法（归约法）

自底向上法是从给定的符号串 α 出发，自底向上构造语法树，不断地使用文法规则进行规约，即用文法规则的左部非终结符替换右部的符号串，最终试图使树根是且仅是文法的开始符号，即对给定的符号串 α，判定其是否存在 $\alpha \xrightarrow{+} Z$；其中：Z 是开始符号。我们约定，当一个句型中有多个简单短语可以进行规约时，优先规约最左简单短语，即习惯上采用“最左规约法”。

对于例 5.1，我们采用自底向上的语法分析思路如下：

1. 分析过程：$a*(b+c) \Rightarrow F*(b+c) \Rightarrow T*(b+c) \Rightarrow T*(F+c) \Rightarrow T*(T+c) \Rightarrow T*(E+c) \Rightarrow T*(E+F) \Rightarrow T*(E+T) \Rightarrow T*(E) \Rightarrow T*F \Rightarrow T \Rightarrow E$
2. 构造语法树：从树叶向树根方向构造语法树。

从上述分析过程不难看出，若分析的第三步不是选择 $b \rightarrowtail F$，而是选择 $T \rightarrowtail E$，则将导致分析失败。可见，规约也是相同的道理，无论是哪一种自底向上的算法，需要关注它是如何通过当前的句型，来判断句柄，即自底向上分析的关键技术是如何确定当前句型的句柄。

在语法分析过程中，如果没有任何外部支持，很难进行规则的选择判断，而一旦判断错误，就会连锁导致后面的推导失败。最简单的方法，就是利用回溯方法，所谓回溯方法，就是把所有产生式规则都平等对待，随机选一个，走不通再换其他规则，理论上来说一定能找到正确的推导过程，但是效率较低。所以，我们后续讲到的语法分析方法，不论是基于推导的还是基于规约的方法，本质上都是为了提高效率。如果文法不存在二义性，正确的分析过程是唯一的，我们希望找到一种策略，或者一些知识来指导系统，不进行回溯，以提高语法分析效率。

## 5.2 递归子程序法

递归子程序法，又名递归下降子程序法，是一种自顶向下分析方法，也就是一种基于推导的方法。从开始符号出发，对每一个非终结符，构造一个子程序，用以识别该非终结符所定义的符号串。递归子程序法的重点在于“递归”以及“子程序”。所谓“递归”，就是程序不断地直接调用或间接调用自己。文法的递归性，使得通过有限的产生式，能够表达无限的语句集合。若文法是递归的，则子程序也递归，故称递归子程序。

递归下降子程序法比起回溯方法，能够准确判断每一步推导选用的产生式，没有试错后的回溯，效率有所提升，但由于递归有大量进栈弹栈操作，效率还有进一步提升的空间。该方法的优点在于简单且易于实现。

递归子程序法的关键在于如何选择正确的候选式来完成推导的过程。递归子程序法的基本原理是为文法中每一个非终结符构造一个子程序，来识别产生式右部；子程序以产生式左部非终结符命名，以产生式右部构造子程序内容。

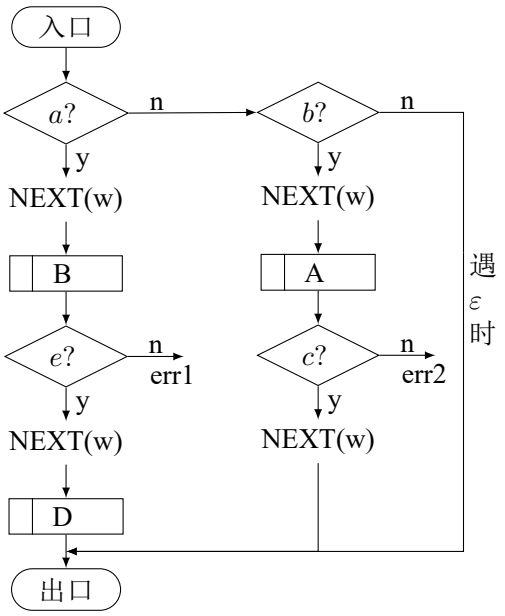



图 5.3: 例 5.3 对应的递归子程序

### 5.2.1 递归子程序法示例

递归子程序法的核心思想是为文法中每个非终结符构造一个子程序，子程序以产生式左部非终结符命名，以产生式右部构造子程序内容。文法中会有很多相同左部的产生式，通常我们将同一个非终结符的产生式，都放到同一个子程序中。

下面通过具体例子来分析递归子程序法的原理。

**例 5.2** 给定文法：A→aBeD（1）| bAc（2）| $\epsilon$（3），非终结符 A 对应的递归子程序如图5.4所示。

其中，NEXT(w) 读的是下一个待处理的单词，即 token 串（token 序列）中下一个单词。

在语法分析过程中，我们用当前符，表示正在处理的字符。如图 5.5 所示，产生式（1）中第一个字母是 a，判断当前符是否为 a，如果不是，则出错，如果是，则读入下一个字符，再进入子程序 B 中，对于 A 这个子程序来说，它并不会去判断 B 里面的内容，而是直接调用 B 的子程序，B 的子程序负责匹配 B 生成的串。从 B 子程序返回后，判断当前符是否为 e，与前面类似，如果不是，则出错，如果是，再读入下一个字符，并进入子程序 D，从 D 返回后，即可到达出口.

图 5.5 所示子程序入口处读入的当前符不是 a，则判断是不是 b。如果是 b，就再取一个字符，然后调用 A 的子程序。A 处理完之后，判断是不是 c。如果不是 c，则报错；如果是 c，那第二个分支执行完毕。

图 5.5 所示子程序入口处读入的当前符既不是 a，又不是 b，是不是应该报错呢？假设是 f，在这里 f 不一定是 A 推出来的，那 f 为什么能进入到这个程序里呢？可能是这样的情况，例如文法中还有一条产生式 C→Af，可以看出 f 是由调用 A 的子程序 C 生成的；A 推出来的是空，因为是 C 调用 A，所以 C 要处理 A 后面的 f。因此当前符既不是 a，又不是 b 时，可执行子程序中第三条分支，返回到调用子程序 A 的程序中，由其进行后续处理。

通过上述分析不难看出，同一个非终结符对应的多个产生式放到同一个子程序中，每一个产生式对应子程序中一个分支。

读者可能会对何时读入下一字符存在疑问，这里给出对于接口问题的约定：

1. 进入子程序前，当前符已准备好，图中进入子程序 B 前，准备好了当前符。

2. 子程序出口前，当前符已读进来，图中判断当前符是否为 e，这里的当前符在子程序 B 中已读入。

接口的约定保证了调用任何一个子程序前，当前符是有效的，即已经读入，但没有被其他程序使用，而当匹配成功后，一定要读入下一字符，更新当前符。如果缺少对于接口的约定，整个程序都会混乱。

接下来，给出使用图 5.5 所示子程序进行语法分析的示例。

**例 5.3** 给定用户输入符号串 α=abed，采用递归子程序法判断符号串 α 是否能由例 5.3 中文法推出。

分析：用 ω 表示当前符。递归子程序入口处首先判断当前符是否为 a，而此时当前符 ω=a，发现匹配成功，如果当前符不是 a，则会进行后续判断。匹配成功后，读入下一个字符，更新当前符 ω=b，因为产生式中接下去的字符 B 是非终结符，非终结符 B 有自身对应的子程序，因此直接进行调用该子程序，B 子程序中具体执行步骤，这里不用考虑。B 处理完后，会自动更新当前符 ω=e，与产生式中对应进行匹配，如果匹配失败，就出错，发现匹配成功，读入下一个字符，并更新当前符 ω=d。产生式中接下去的字符 D 是非终结符，因此直接进行调用 D 对应的子程序，如果能从 D 中正确地返回，就到达出口。

### 5.2.2 递归子程序构造方法

递归子程序法为每一个非终结符构建一个子程序，为构建一个完整程序，我们还需要一个主程序。给定文法 G(E)，其中 E 为文法初始符。上一节中关于子程序入口，约定进入每一个子程序前，都需要准备好当前符，而利用 E 产生式构建的子程序作为主程序，会出现缺少当前符问题，因此我们无法将文法初始符的子程序作为主程序。为解决这一问题，我们会增加一个新的产生式 Z→E，这个产生式很简单，左边是一个新的符号，右边是原本的文法初始符。把这个新产生式构造的子程序作为主程序，帮助第一个子程序，读入当前符。给定一个文法 G(E)，其中 E 为文法初始符。递归子程序构造包括以下几个步骤：

1. 扩展文法：定义一个新的产生式 Z→E。把这个新产生式构造的子程序作为主程序，用于将待分析符号串的第一个字符读进来。

2. 子程序内容设计：

   (a) 遇到终结符：判断当前符与该终结符是否一致，如果一致，则读下一个字符；

   (b) 遇到非终结符：调用非终结符的子程序，在非终结符返回之后不需要再读一个字符，因为返回的时候，已经把下一个待处理的字符读进来了；

   (c) 遇到空串：直接连接出口。

下面举例说明递归子程序构造方法。

**例 5.4** 给定文法 G(S): S→aAb|bS, A→cd|ε. 试构造该文法的递归子程序。

首先，构造主程序。增加一个新的产生式 Z→S，把这个新产生式构造的子程序作为主程序，帮助第一个子程序 S，读入当前符。然后调用原来文法初始符 S 的子程序，返回后，判断当前符是否为结束符 #，如果不是则出错，如果是则分析成功，程序结束。主程序如下图所示。

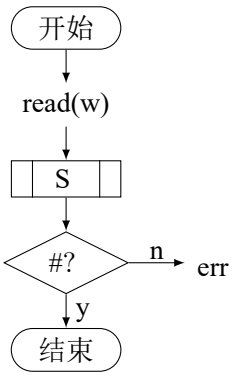



图 5.4: 例 5.4 的主程序

非终结符 S 的子程序，如图 5.7 所示。首先要判断当前符是不是 a。如果是 a，就再读一个字符（单词），这里调用 A 的子程序，A 执行完之后，无需 next(w) 函数读下一个字符，因为 A 的出口会把下一个单词读进来。接着判断 b，如果是 b，则要把下一个字符读进来。S 子程序入口处判断当前符不是 a，则判断是不是 b，如果是则读下一个字符，然后调用 S。调用完 S 后，不需要再读字符，直接出来就可以。如果既不是 a，也不是 b，就会报错。

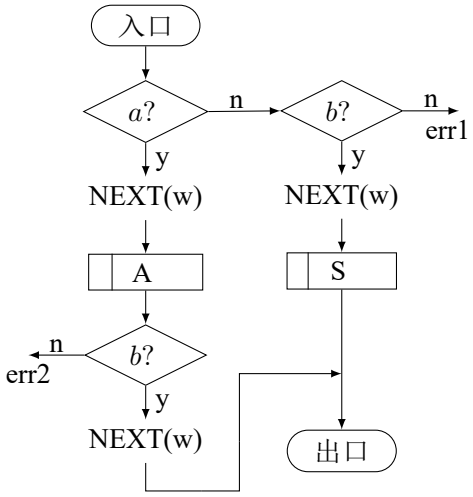



图 5.5: 例 5.4 的子程序 S

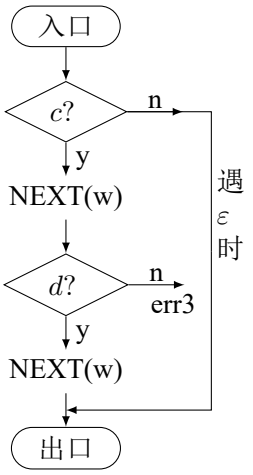



图 5.6: 例 5.4 的子程序 A

非终结符 A 的子程序，如图5.7所示。先判断当前符是否为 c，如果是就读下一个字符，然后判断 d。如果不是 d 就报错，如果是 d 就接着读，接着就是出口。子程序 A 入口处当前符如果不是 c，就会直接从 A 的子程序出去，然后在 S 的子程序里接着判断是否是 b。

### 5.2.3 递归子程序法适用范围

递归子程序是不是适用于所有的语法分析问题呢？我们分析两种情况。

1. 如果两个具体有相同左部产生式的第一个字符相同，例如 S→bA|bB，两个产生式首符号均是 b，如果当前符是 b，此种情况递归子程序无法判断应该选择哪一个产生式。可能的解决方案有：看下一个字符，来判断选择合适的产生式。但是在递归子程序法中，不能往后看一个字符，它只有通过当前符来进行判断。

   因此，同一个非终结符的不同产生式，如果它们的首符号相同，则无法设计相应的递归子程序。

2. 如果文法中有这样一条产生式规则 A→Ab，进了 A 子程序之后就调用 A，无限调用永远不停止，陷入

死循环状态。从这里我们可以看出，如果产生式存在直接左递归，就会进入死循环，因此运用递归子程序法的文法中，不允许直接左递归产生式存在。

综上，给定一个文法，如果不存在直接左递归，并且具有相同左部的各产生式，首符号不同，则该文法就可以采用递归子程序的方法进行语法分析。

**例 5.5** 给定文法 G(E)，试构造该文法的递归子程序。

G(E)

$$E \to T \mid E\omega_0 T$$
$$T \to F \mid TE\omega_1 F$$
$$F \to i \mid (E)$$

其中产生式 E→E+T 和 T→T*F 中存在直接左递归，不能直接运用递归子程序法，需要先消除直接左递归。进行消除时，必须保证变换前后是等价的。我们通过文法变换消除左递归，结果如文法 G′(E) 所示。

G′(E)

$$E \to T\ \omega_0 T$$
$$T \to F\ omega_1 F$$
$$F \to i(E)$$

其中 “ 和 “ 是一种特殊的符号，并不是终结符，它表示括号里的内容是可选的，可以是 0 个，可以是 1 个或多个。

转换成上述形式后，不存在直接左递归，下面利用递归子程序法进行语法分析。首先构造主程序。增加一个新的产生式 Z→S，把这个新产生式构造的子程序作为主程序。主程序同图 5.6。

非终结符 E 的子程序如图 5.8 所示。从入口进入后，先调用 T 的子程序，因为后边里的内容是可选的，所以从 T 的子程序返回后，先判断当前符是否为 $\omega_0$ ，如果匹配成功，就读入下一字符，调用 T 的子程序，因为里的内容可能有多个，所以从 T 的子程序返回后，回到判断当前符是否为 $\omega_0$ ，当前符不是 $\omega_0$ 时，就认为后面不属于该产生式，到达出口。我们发现当前符不是 $\omega_0$ 时，可能不是正常结束，而是出现错误，但在递归子程序法中，在出口前的位置，无法判断是否出错，而是让后面的程序来判断是否出错，这是一种滞后报错。

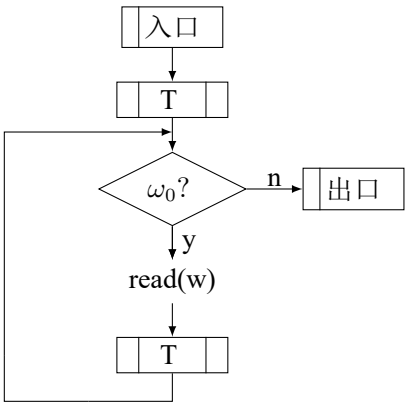



图 5.7: 例 5.5 的子程序 E

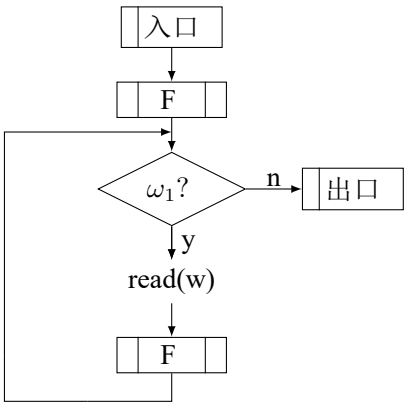



图 5.8: 例 5.5 的子程序 T

非终结符 T 和 F 的子程序，分别如图 5.9 和 5.10 所示，具体构造过程，此处不再赘述。

递归子程序法在推导过程中，面对同一个非终结符多个产生式，无需回溯，根据当前状态便可自动准确判断应用哪条产生式，其关键在于对文法进行了约束，即同一个非终结符的多个产生式的首符号不相同，且

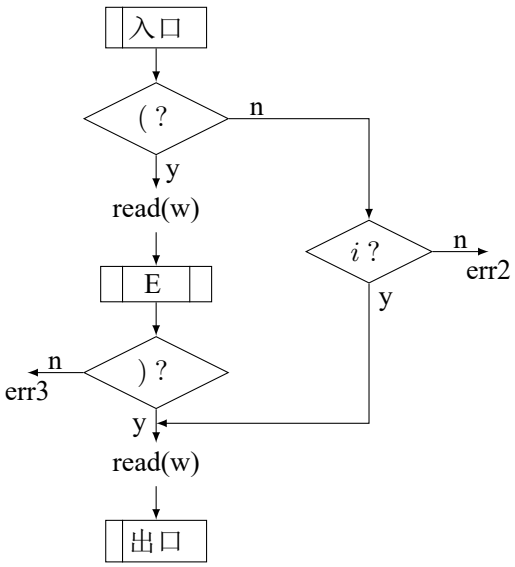



图 5.9: 例 5.5 的子程序 F

文法不存在直接左递归。递归子程序法简单有效，但它对文法的约束较强，对大量文法的语法分析难以有效。另外，由于递归子程序法其递归性的存在，不断的递归调用，导致整个方法的效率下降。因此，虽然递归子程序法的思路清晰，实现简单，在实际应用中，也往往转化为非递归的方法，从而提高整体效率。

## 5.3 LL(1) 文法定义

上一节中我们提到过，语法分析方法通常分成两大类，分别是自顶向下法（推导法）和自底向上法（归约法）。递归子程序是其中的一种自顶向下的方法。

下面要介绍的是第二种自上而下的方法，也是一种基于推导的方法，叫 LL(1) 分析法。为什么叫 LL(1) 分析法？这两个‘L’都是有意义的：第一个‘L’指的是自左向右扫描，即算法（程序）对字符串进行分析的时候是自左向右扫描的；第二个‘L’指的是最左推导，即算法要求推导最左边可推导的非终结符。括号里的“1”指的是在推导过程中，只查看当前的一个符号。为了简单的理解上述内容，我们给出一个例子。

**例 5.6** 如有以下文法

文法定义

S → 主谓宾
主 → 我 | 你 | 它 | 小牛
谓 → 研究 | 学习
宾 → 主 | 自然语言技术

现在需要检查**目标语句**“小牛研究自然语言技术”是否归属于这个文法，我们从开始符号 S 出发得到**分析短语**“主谓宾”。对目标短语从左往右逐个扫描的分析策略是第一个‘L’，对分析短语自左向右推导就是第二个‘L’(具体的说，就是当分析短语存在多个非终结符的时候，我们总是从最左侧的非终结符开始推导)。

1. S → 主谓宾
2. 从目标语句中读入“小”字，确定主 → 小牛
3. 从目标语句中读入“牛”字，匹配

4. 从目标语句中读入“研”字，确定谓 → 研究
5. 从目标语句中读入“究”字，匹配
6. 从目标语句中读入“自”字，确定宾 → 自然语言技术
   .... 接下来就全是匹配过程

在上面的分析流程中，我们确定推导式子，只需要读入一个字符即可，这就是 LL(1) 的 1 的含义。既然有 LL(1)，那么也可以存在让算法观察更多字符来决定选用什么式子进行推导的方法，也就是 LL(2)、LL(3) 等等。观察更多的字符，对相同左部的产生式右部有相同的前缀会有更多的容忍（允许产生式右部有相同的前缀），比如例5.5的‘主’如果还能推导出“小刚”，那我们需要再多看一个字符才能决定主语要推出什么右部。

LL(1) 分析法还有一种别的称呼，叫预测分析法。“预测”指的是根据当前的状态，能够知道接下来使用的是哪条产生式。刚才已经提到，LL(1) 分析法与递归子程序一样，都是自上而下的，而且是确定性文法的分析方法。对于 LL(1) 分析法，有三个基本的要点：

※ 三个基本要点

1. 利用一个分析表，登记如何选择产生式的知识；
2. 利用一个分析栈，记录分析过程；
3. 此分析法要求文法必须是 LL(1) 文法。

更详细的补充说明上述三点：

1. LL(1) 分析法必须要有一个分析表，这和递归子程序不一样。分析表的用途是什么？分析表登记了一些如何选择产生式的知识，即在某一个时刻状态下，应该用哪个产生式。
2. LL(1) 分析法里面有一个分析栈，是在执行分析方法的时候用的，目的是记录算法分析的过程，就是要把分析的状态，通过一个栈的形式记录下来。
3. LL(1) 分析法的前置条件是，其所分析的文法必须是 LL(1) 文法。回顾一下 LL(1) 文法的两个条件：第一，左部相同的产生式，其右部产生式的首字母不能相同；第二，不能有左递归。.... 接下来就全是匹配过程

## 5.4 LL(1) 分析法的完整流程

**例 5.7** 一个具体例子的演示

先对 LL(1) 文法进行感性的认识。首先，把产生式进行编号。

G(Z):

Z → dAZ ① | bAc ②
A→aA ③ | $\varepsilon$ ④

四个产生式都进行了编号，其中 ①、②两个产生式的左部都是 Z，③、④两个产生式的左部都是 A。对于这些产生式，可以得到一个分析表。这个分析表，每一行都对应了一个非终结符，每一列都对应了一个终结符，以及结束的标志‘#’。

| | a | b | c | d | # |
|---|---|---|---|---|---|
| Z | | ② | | ① | |
| A | ③ | ④ | ④ | ④ | |

这样的一个表称为分析表，通过一个终结符和一个非终结符，我们就能查到一个表项。表项表示什么？比如，②表示的意思是，当非终结符 Z，看到了终结符‘b’，就要使用产生式②进行推导。其它项以此类推。

有了分析表之后，就可以用 LL(1) 分析法进行分析。例如对符号串 α=“abc#”进行分析。分析时需要有一个栈结构，记录当前的符号是什么，剩余的符号是什么，操作是什么。

对符号串：α=bac # 的分析过程：

| 栈 | 当前符号 | 剩余序列 | 栈操作 |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

图 5.10: 对符号串 α=“abc#”的分析过程表

首先，要把‘#’压栈，这是规定。之后还要把起始符号 Z 压到栈里，因为推导得从起始符号推导。此时，待处理的第一个符号是‘b’，后面的‘a’、‘c’、‘#’是还没处理的剩余序列。这个时候该如何执行分析方法？栈顶符号是一个非终结符，就用非终结符 Z 和当前符号‘b’到分析表里进行查找，对应的是产生式②这一项，因此就要用产生式②进行推导。

✻对符号串：α=bac # 的分析过程：

| 栈 | 当前符号 | 剩余序列 | 栈操作 |
|---|---|---|---|
| # Z | b | a c # | 选择 Z→bAc② |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |


图 5.11: 根据栈顶元素和当前符号选择操作 2

下一个操作是让 Z 出栈，然后把 Z 的右部“bAc”逆序压栈，即‘c’、‘A’、‘b’依次进栈。因为栈是先进后出的结构，这样‘b’就能在栈顶，可以先匹配，‘A’次之，‘c’最后。现在栈顶符号和当前符号都是‘b’，也是终结符，因此操作就是匹配‘b’。如图 5.13所示。


图 5.12: 根据当前符号和栈顶元素进行匹配

匹配‘b’之后，就将‘b’出栈，栈顶符号变成‘A’，当前符号是‘a’，剩余序列是‘c#’。‘A’是一个非终结符，当前符号是‘a’，到分析表里查，得到的对应项是③，所以应该用产生式③进行推导。如图 5.14所示。


图 5.13: 根据栈顶元素和当前符号选择产生式

‘A’推导完后，就把‘A’弹出栈，再把产生式③的右部逆序压栈。这时候栈顶是‘a’，当前符号也是‘a’，于是匹配‘a’。如图 5.15所示。


图 5.14: 根据当前符号和栈顶元素进行匹配

匹配完‘a’后，弹出栈顶符号的同时，当前符号读取下一个，即‘c’，剩余序列只剩一个‘#’。对于栈顶符号‘A’和当前符号‘c’，查表得④。如图 5.16所示。


图 5.15: 根据当前符号和栈顶元素选择产生式

这时候选择是空，操作是一样的，把栈顶的非终结符弹出去，右部逆序压栈，空的逆序压栈即没有元素压栈。当前符号是‘c’，所以匹配‘c’。

匹配了‘c’之后把栈顶符号弹出去，这时候就剩下‘#’，当前符号也处理完了，只剩下‘#’，说明匹配结果是正确的。


图 5.16: 最终的匹配流程图

总结一下以上 LL(1) 分析过程:

1. 首先，给文法规则的产生式进行编号，得到一个分析表，分析表的每一行代表一个非终结符，每一列代表一个终结符或表示结束的‘#’。
2. 接下来是对符号串 α=“bac#”的分析，先把‘#’压进栈，然后压入一个起始非终结符，之后就根据栈顶符号和当前符号去查分析表找到要做的操作，弹出栈顶符号并将右部逆序压栈。
3. 如果栈顶符号和当前符号是相同的终结符，就进行匹配，然后把匹配的元素弹出栈，更新要识别的当前符号。以此类推，直到栈顶符号和当前符号都是‘#’，即分析正确。

这里有两个问题，分别是:

1. 选择推导产生式后，为什麽要逆序压栈?
2. 当栈顶为 A, 当前单词为 c 时，为什么选择 A $\rightarrow \varepsilon$?

这个地方我不太确定，试着给出这两个问题的答案:

1. 我们选择一个产生式是通过分析表，而分析表的形成主要是通过相同左部，分析它右部各子产生式最左端不相同的前缀字符形成的。LL1 文法从左侧开始扫描目标语句，每次读入一个字符，正好可以和选中的右部子产生式的第一个左侧字符匹配，所以可以发现每次选取非 $\varepsilon$ 的产生式时下一步一定是一个匹配流程。又因为栈是 FILO 结构，所以逆序压栈才能保证产生式的左端在栈顶。
2. 栈顶是非终结符号且当前元素不是其产生式的最左字符，说明这个非终结符处理不了这个情况，只能交给栈中在其下的终结符进行匹配或者非终结符进行生成再匹配。这个问题就好像你想要一杯牛奶结果店里只有咖啡，店员不可能收了你的钱给你一杯咖啡，只会让你去找另一个有牛奶的店买牛奶一个道理。

### 5.4.1 抽象的流程表示

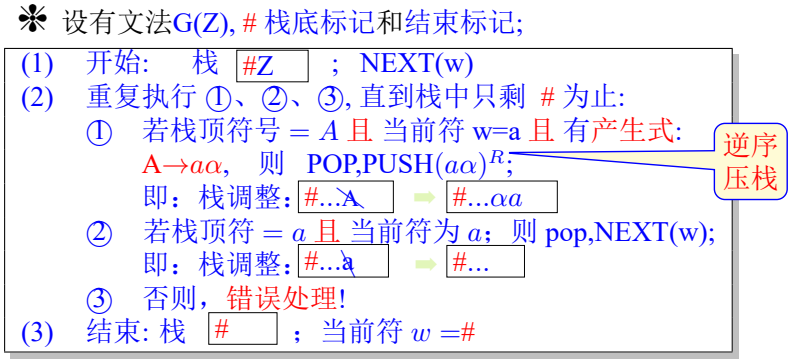



图 5.17: LL1 流程

**Algorithm:** LL(1) 分析流程 $(G, W)$

**Input:** 文法集合 $G$ 待匹配目标字符串 $W$

**Output:** 一个状态，接受字符串或者错误

先把'#' 和开始符号压入栈中，从目标字符串左侧中读入一个字符。初始化栈结构 **Stack**，初始化分析表 **T**。

**for** $w \in W \quad and \quad w! =' \#'$ **do**
  **if** $Stack.top()$ is $Non - Terminator$ **then**
    generate=lookup(T)
    Stack.push(generate.reverse())
    Stack.pop()
  **else if** $Stack.top() == w$ **then**
    Stack.pop()
  **else**
    **return** "$Error$"
  **end**
**end**
**if** $Stack.pop()! =' \#'$ **then**
  **return** "$Error$"
**return** "$Accept$"

开始如果栈顶是非终结符，根据当前元素查找分析表选择产生式，弹出栈顶非终结符，逆序将产生式右部压栈。如果栈顶是终结符，且与当前元素相同则弹出栈顶元素即可。否则报错。循环这个过程直到栈顶和当前元素都是'#' 时，认为可以接受，否则报错。

## 5.5 LL(1) 文法及其判定

前面我们说过 LL(1) 分析依赖于特定的文法特征，大家可以回忆一下前面所提到的三个基本要点的第三点。同时我们还没有给出一个很规范的流程，怎么得到分析表，这依赖于我们即将提出的三个概念。为了更清晰的表达这个概念，我们借助一个具体的文法展开

**文法定义**

$G(Z) = (V_N, V_T, Z, P)$

$A \to \alpha \in$ P, 这里的 $\alpha$ 是一个串，可以是终结符也可以是非终结符组成。

### 5.5.1 首符号集合、后继符集合与选择符集合

**定义 5.2** 首符号集合

$$first(\alpha) = \{t | \alpha \stackrel{*}{\Rightarrow} t..., t \in V_t\}$$

定义在 α 上，这里 α 是一个串，是一个产生式的右部。first 集的定义就是，所有 α 能推导出来的串，这

些串的串首的终结符所构成的集合称之为 α 的 first 集，也叫首符号集合。通俗一点讲就是，把 α 能推导出来的第一个终结符的集合，称之为 α 的首符号集合。

**定义 5.3** 后继符集合

$$follow(A) = \{t | Z \overset{*}{\Rightarrow} ...At..., t \in V_t\}$$

注意，follow 集和 first 集定义的范围不一样，first 集是在一个串，也就是产生式右部上定义的，follow 集是在左部，也就是非终结符上定义的。A 的 follow 集表示的是 A 这个非终结符，在推导的过程中，后面能跟哪些终结符。

**定义 5.4** 选择符集合

$$select(A \to \alpha) = \begin{cases} first(\alpha), \alpha \overset{*}{\nRightarrow} \varepsilon \\ first(\alpha) \cup follow(A), \alpha \overset{*}{\Rightarrow} \varepsilon \end{cases}$$

要形成分析表，只需要观察选择集。选择集定义的范围是在产生式上，比前面 first 集合在产生式右部、follow 集在产生式左部的范围更大一些。当整个文法面临一个非终结符和待生成的终结符，选择集告诉我们我们应该选择哪一条产生式进行推导。

一个右侧不能推出空的表达式，他的选择集就是他的 first 集合，这个理由非常明显，既然右侧不能推出空，最后一定会形成一个只有非终结符的串，那这个非空串的串首字符显然就是这个非终结符在碰到它时应选择什么产生式的凭证。

一个右侧能推出空的产生式，他的选择集多了一个产生式左部的 follow 集合，这也很好理解，因为当非终结符为空的时候，他不会影响那一个紧挨着他的终结符的推导过程，这也是我们前面所说的去牛奶店买牛奶而不是去咖啡店买牛奶（当然也许咖啡店也卖牛奶）。

注:

1. $\alpha \overset{*}{\Rightarrow} \varepsilon$($\alpha$ 可空), $\alpha \overset{*}{\nRightarrow} \varepsilon$($\alpha$ 不可空)；
2. 若 $\alpha$=$\varepsilon$ 则 first($\alpha$)={ } 是空集；
3. 设 # 为输入串的结束符，则 #∈ follow(Z);

这三条注解其实是偏定义的东西，很难说明为什么要这么做。就第一条来说，能推出空他自己就可以是空，不能推出空他自己不能是空，说这么多就是 **禁止无中生有**，科学也得遵循哲学指导不是。

第二条定义也非常自然，空了啥玩意都推不出来了，那还有啥终结符呢。这个式子就是告诉我们**一分耕耘，一分收获**，你今天不播种，那肯定没有收成。理解成物质守恒也行，咋好记你就咋记。

唯独需要小心的是第三条，他很容易被忘记，你可以认为每个文法都隐含着这样一条产生式 Z⇒Z#。Z 是开始符号，# 是结束符号，用咱中国人的话说那就是**有始有终**，建议同学们在科学学习中贯穿哲学文化思想。

**例 5.8** 一个分析表的构建实例

经过这么多的定义，得拿起武器开始干活了。大家也学过编程，这个像分段函数一样的选择集定义，在编程上就是个 if-else 的分支结构。所以我们拿到一个产生式的时候，得先去做个判断，判断产生式的右侧能不能为空，我们不需要很严谨的把整个产生式都推导到全是终结符，一旦出现一个终结符，就说明右侧不为空了，因为终结符不能继续推导，这个就是**赚来的钱，交的朋友都可能会离开你，他们是非终结符; 学到的知**

**识是终结符，它永远空不了。**

我们从这个文法中说明怎么形成分析表

G(Z):

Z → dAZ ① | bA ②
A→aA ③ | ε ④

①显然不为空，因为出现了非终结符 d；②显然不为空，因为出现了非终结符 b；③同理不为空；④为空。

不为空的①②③，select 集合就是 first 集，分别是 d、b、a。

能为空的④，select 集合是 first 集与 follow 集的并集，但是因为他不是能为空，而是本身是空，根据上面的“物质守恒”原则，他的 first 集也是空，他的 follow 集合就是 d 和 b 以及 #，这个地方可能有点难，我给你们推荐个经验，先从别的产生式里面找到这个非终结符，如果非终结符后头跟的是终结符，直接塞进 follow 集，如果跟的是非终结符，把这个后头的非终结符的 first 集塞进 follow 集合即可。所以 A 这个非终结符，出现在了①②中。在①中，其后跟的是非终结符，把 Z 的 first 集合也就是 {d,b} 加入 A 的 follow 集合。其次是’#’，这个根据我们前面提到的“有始有终”原则，所有的开始符号后面都隐含跟随着一个’#’ 号，所以②中可以认为是 Z → Z# → bA#，于是 A 的 follow 集合就还有 # 号。

到这个地方，select 集其实就是分析表的等价表示，如果你还没想明白怎么从 select 集构建分析表，你需要下去花点功夫，当然也许睡一觉起来就想明白了。这个分析表的结果在上面 LL1 的分析例子中给出来了。

### 5.5.2 LL(1) 文法及其判定

**定义 5.5** LL(1) 文法

LL(1) 文法是指文法中，具有相同左部的产生式，其 select 集合没有交集

这个文法是递归子程序法和 LL(1) 分析法的使用前提。因为一旦相同左部的产生式的 select 集有交集，我们就不知道在面对一个元素的时候需要选择哪一条产生式，也就是出现了歧义。

我们举一个新的例子，

G(Z):

Z → Zb ① | a ②

$$\because \left.\begin{array}{l} select(①) = a \\ select(②) = a \end{array}\right\} \quad Selection\ set\ intersection$$

这两个产生式的选择集合相交了，就不属于 LL(1) 文法了。这里也能看到，在递归子程序里，不能有左递归，一旦有左递归，选择集合一定会相交。具有左递归的文法，一定不是 LL(1) 文法！那有左递归，该怎么办呢？我们可以尝试把左递归改写成右递归形式。

再扩展一下，左递归有两种，一种是直接左递归，一种是间接左递归，上面的例子是直接左递归，这两种左递归都需要消除。一个通用的算法，如下

**Algorithm 2:** 消除左递归算法

```
以某种顺序排列非终结符 A1，A2，……，An;
for int i = 1; i<=n; i++ do
    for int j = 1; j<=i-1; j++ do
        将每个形如 Ai → Ajγ 的产生式替换为产生式组 A i → ξ1γ | ξ2γ | …… | ξkγ，其中，Aj→a1
          | a2 | …… | ak 是所有的当前 Aj 产生式
    end
end
消除关于 Ai 产生式中的直接左递归性 }
```

转换为右递归文法后，select 集不相交了

G'(Z):

Z→aA ①

A→bA ②| $\varepsilon$ ③

$$\left.\begin{array}{l} select(②) = b \\ select(③) = \# \end{array}\right\}$$

∴ G'(Z) 是 LL(1) 文法，可以使用 LL(1) 分析。

## 5.6 LL(1) 分析器设计 (实现)

分析器由两部分构成，第一部分是分析资源，在 LL(1) 分析里面就是分析表，第二部分是程序，程序通过读分析表来完成分析过程。简单的说，就是构建分析表，和利用分析表解析待识别串。

LL(1) 分析表 + LL(1) 控制程序

### 5.6.1 LL(1) 分析表的构造

LL(1) 分析表是存储文法选择集合的知识表。表列是终结符，行是非终结符，每个元素是产生式序号。展示一下 C++ 版本的一些粗糙思路供同学们参考。

```
struct Element{
    int number; //产生式序号
}
//这两个Map需要一段初始化程序，根据你选择的输入方式决定，本质上就是一段从字符串里分离 $\rightarrow$ 左右部分的处理。选择使用map是因为需要一个从char到int的映射，才方便从表里存取表项，很自然的，我们还需要维护两个int类型的index变量，来为每一个独立的char进行一个index++的初始化。这段地方留给同学们自己想想。
Map<char,int> terminator;
Map<char,int> Nterminator;
Element L[terminator.size()][Nterminator.size()];
```

```
8    //数据结构定义到这里了仍然存在一些小问题，那就是我们该怎么记录产生↩
         式，不然我们拿到了序号也没有意义，当然我们可以不存序号，将element↩
         类里改成string类型，直接存下产生式的右部，还有很多选择，我就不赘↩
         述了，这个地方同学们后期做编译原理课设会有很多机会去实现的。
```

这个地方的代码是我根据 PPT 的意思写的，不太符合工程实现的感觉。你也可以按你喜欢，觉得更工程的方法来完成这个步骤。如果你已经把数据结构的定义和初始化想好了，那我们可以开始下一个问题，也就是分析表的初始化。这个需要我们去求 first 和 follow 集合，还是挺有挑战性的。我是这么想的，如果对编程不熟悉的同学，可以考虑手动输入每个产生式的 first 和 follow 集合；如果编程能力比较强的同学，可以尝试自动从产生式里解析出来 first 和 follow 集合，因为后期课设的产生式非常多非常丰富，如果纯靠脑袋去想会很累，当然最重要的是很不酷。

我给出一点我的想法，希望是启发，不是约束。我尽可能避开具体的代码，只给出注释。

```
1    map<char,set<char> firstSet;
2    void genFirstSet(){
3        //先初始化终结符的firstset，当然是自身。
4        for(){
5            //遍历终结符集合，逐个插入对应的firstSet中
6        }
7
8        //这里是难度重头戏，需要遍历产生式，为非终结符生成firstSet，我建↩
             议流程是这样的。
9        // 这里应该有一个循环
10       while(){
11           // 取出产生式的左部，去firstSet里面拿到其对应的firstSet，可能↩
                 是空，也可能是不完整的firstSet，也可能是完整的。
12           //判断这个产生式的右部是不是终结符或者是空，且是否存在这个左↩
                 部的firstSet中，不在就加入
13           //否则如果是非终结符，判断右部第一个字符能不能是空，如果能，↩
                 就把他的firstSet复制过来，继续往后看，这里应有一个循环。
14           //如果右侧第一个非终结符不空，那直接复制firstSet就好
15           //这里还有个小困难，就是如何结束这个循环，我建议是当没有改变↩
                 时逃出，通过维护一个bool变量，在有改变的分支里不断变换他↩
                 的值，在循环一开始改动他的值。
16       }
17   }
```

first 集合的自动生成大概如上所示，一个定义的不错的结构，会让你的思路事半功倍，如果做完 first 集合生成，相信 follow 集合也不会难倒你. 加油.

接着是生成 select 集合，这个地方根据上面的定义做就好。在做完这一切，得到分析表之后，你可以看一下

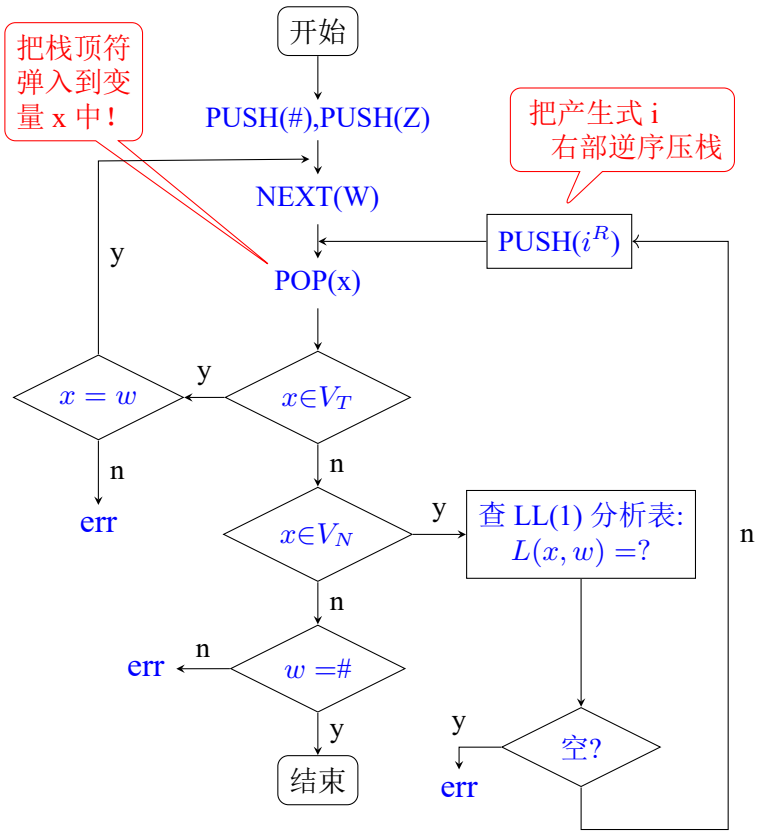



图 5.18: 流程图

我们回头再梳理一遍要做的事情

1. 消除左递归
2. 生成 firstSet
3. 生成 followSet
4. 生成 selectSet
5. 生成分析表
6. 借助控制程序进行分析

## 5.7 LR() 分析法的介绍

### 5.7.1 LR() 分析法的“统治地位”

LR 分析是当前最一般的分析方法。因为它对文法的**限制最少**，现今能用上下文无关文法描述的程序设计语言一般均可用 LR 方法进行有效的分析。而且在分析的效率上也不比诸如不带回溯的自顶向下分析、一般的“移进归约”以及算符优先等分析方法逊色。

**上下文无关文法的简要回顾**

上下文无关文法就是说这个文法中所有的产生式左边只有一个非终结符，很多人可能对上下文这个概念一知半解，我举一个反例。aZc→abZbb，当我们需要匹配'Z' 的时候，需要确保这个'Z' 的两侧有正确的上文'a' 和下文'c'，这个就是上下文有关文法。

LR(1) 分析是我们重点关注的内容，与以前的分析方法依赖的文法包含关系如图 5.20 所示。需要说明，LR(0) 和 LL(1) 没有包含关系，但是有交集。

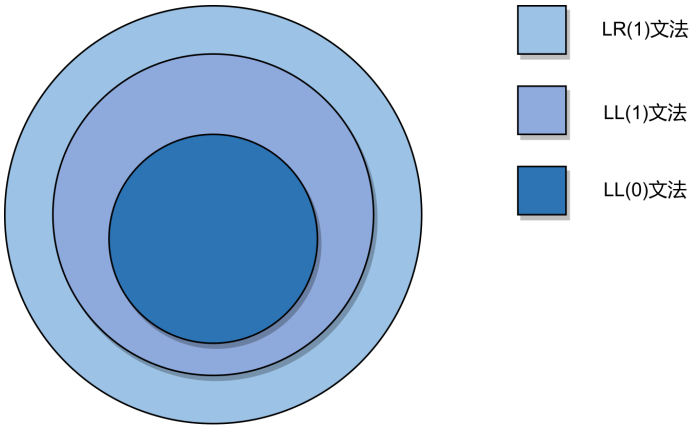



图 5.19: 常见的三种文法的包含关系

一些课外的扩展知识研究中提到一个由 LR(K) 文法产生的语言均可以等价的由某一 LR(1) 文法产生，这也是我们为什么在课程只研究 k ≤ 1 的情况，因为对于程序编译基本够用了，感兴趣的同学可以自行查找证明资料。我在编纂这个的过程也找到了一个对 LL 分析和 LR 分析进行对比的 blog[1], 也可供同学们做课外读物之用。

### 5.7.2 LR() 分析法的定义

LR( ) 分析法是指从左到右扫描、最左归约 (LR) 之意；它属于自底向上分析方法。推导和归约是一对互逆的过程，推导是从非终结符产生零到多个终结符的过程，归约则是从多个终结符逆推回开始符号的过程。相信在上一节 LL(1) 分析法刚刚接触过扫描和推导的读者，应该不会对这个表述很陌生。

在本节学习中，我们需要认识 LR(0)，掌握 LR(1) 分析，括号中的数字是需要看几个当前元素才能决定归约产生式的意思，和 LL(1) 中的'1' 没有什么不同。

为了让同学们更深入的理解最左归约和最左推导这个 LR 和 LL 分析法唯一的不同, 我们会给出一个例子。但是在此之前, 我需要先介绍两个概念.

**略显晦涩的定义**

短语: 如果有 $S \overset{*}{\Rightarrow} \alpha A\delta$ 且 $A \overset{+}{\Rightarrow} \beta$, 则称 $\beta$ 是句型 $\alpha A\delta$ 关于非终结符 A 的短语。特别的，如果 $A \Rightarrow \beta$，则称 $\beta$ 是句型 $\alpha A\delta$ 关于非终结符 A 的**直接短语**。
句柄：一个句型的最左直接短语称为该句型的句柄。

最左归约的任务就是要在扫描特定数量的元素之后，确定当前句型的**句柄**。

[1] https://blog.reverberate.org/2013/07/ll-and-lr-parsing-demystified.html

接着到具体的例子上，展示一个 LR() 分析过程，注意不是 LR(1)

**文法定义**

$G(Z) = (V_N, V_T, Z, P)$

$Z \to aBAd, A \to bc|c, B \to bB|c$

符号串 $abccd$ 最左归约过程:

1. 读入 a，没法归约，此时存着 a
2. 读入 b，没法归约，此时存着 ab
3. 读入 c，此时存着 abc，其中的 c 可以被归约成 B，也就是 $B \leftarrow c$，归约完后存着 abB。这里是一个难点，会有同学思考为什么不是 $A \leftarrow bc$，从而只存着 aA。
4. 读入 c，此时存着 abBc，$B \leftarrow bB$，归约完后存着 aBc
5. 读入 d，此时存着 aBcd，$A \leftarrow c$，归约完后存着
6. 最后 $Z \leftarrow aBAd$，归约结束

整个流程是

1. $ab\underline{c}cd \to a\underline{bB}cd \to aB\underline{c}d \to \underline{aBAd} \to Z$
2. $ab\underline{c}cd \to aA\underline{c}d \to aABd$ 无法进行下去了

我们可以解释一下第二条路径为什么行不通，上面的短语定义有一个前提，那就是开始符号能推出这个串，但是 aAcd 并不能被开始符号 Z 推出。从语法树角度来说，一个短语应该是从语法树的叶节点开始，囊括一个完整的子树，如果从这个角度思考，b 和 c 同为叶节点，但是他们之间并没有通路，所以不能算是一个完整的子树，既然不是短语，那就不会是句柄了。图 5.21中展示了按序号顺序构建语法树的过程。

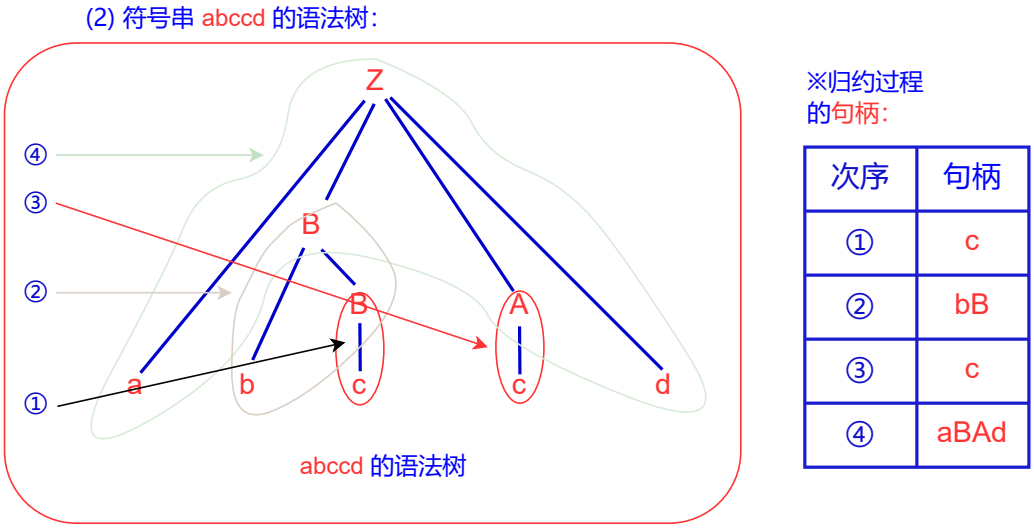



图 5.20: 最左归约形成的语法树

我们用规整的数据结构来表示上面的最左归约过程

(3) 利用分析栈记录分析过程:
设 待分析的符号串：abccd#

| 分析栈 | w | 剩余串 | 句柄产生式 | 操作 |
|---|---|---|---|---|
| # | a | b c c d # | | 移进，NEXT |
| # a | b | c c d # | | 移进，NEXT |
| # a b | c | c d # | | 移进，NEXT |
| # a b c | c | d # | B→c | 归约, |
| # a b B | c | d # | B→bB | 归约, |
| # a B | c | d # | | 移进，NEXT |
| # a B c | d | # | A→c | 归约, |
| # a B A | d | # | | 移进，NEXT |
| # a B A d | # | | Z→aBAd | 归约 |
| # Z | # | | | OK |

句柄

图 5.21: 分析栈记录的分析过程

**句柄识别器的构造**

在具体程序中，我们需要找到更简单的方法来辨别目前保存的串中是否有句柄。我们通过标注产生式右部的文法符号位置的扩展文法实现 (同时还用一个新的开始符号替换了旧的开始符号)。

**文法定义**

扩展前 $Z \to aBAd, A \to bc|c, B \to bB|c$

扩展后

$Z' \to Z_1$ ⓪ , $Z \to a_2B_3A_4d_5$ ① , $A \to b_6c_7$ ②$|c_8$ ③, $B \to b_9B_{10}$④ $|c_{11}$ ⑤

这么做的好处在于，同一个 c，$c_8$ 和 $c_{11}$ 就不会在归约时产生歧义，从不同位置归约就不一样了。我们把他们的位置当做状态，构建自动机来识别句柄。

具体的做法如下，我们假设开始状态都是 0，有了状态，自动机还需要状态转移方程 $\delta(state.token) = next\_state$ 才完整。构建完成的自动机如图 5.23所示。其中移进状态是 0、2、3、4、6、9, 在这个状态下需要继续读入字符，才能判断下一步行为。归约状态是 5、7、8、10、11，在这个状态下可以进行归约操作。接受状态是 1，OK 表示归约结束。我们用刚刚识别 abccd# 的过程举个例子，以 $b_9B_{10}$ 来说，当到达归约状态时弹出 $b_9B_{10}$，回到 2 状态，2 状态见到 B 就到了 3 状态，别的就没什么不好理解的地方了。

(#0,a) ⇒ (#0$a_2$,b) ⇒ (#0$a_2b_9$,c) ⇒ (#0$a_2b_9c_{11}$, c)
⇒ (#0$a_2b_9B_{10}$, c) ⇒ (#0$a_2B_3$,c) ⇒ (#0$a_2B_3c_8$, d)
⇒ (#0$a_2B_3A_4$, d) ⇒ (#$a_2B_3A_4d_5$, #) ⇒ (#0$Z_1$, #) ⇒ OK

句柄

图 5.23: 句柄自动机运用后的识别图

句柄识别器又称“活前缀图”：意思是在最左归约过程中，识别了句柄，实际上也就识别了以句柄为后缀的该句型（规范句型）的前部符号串。

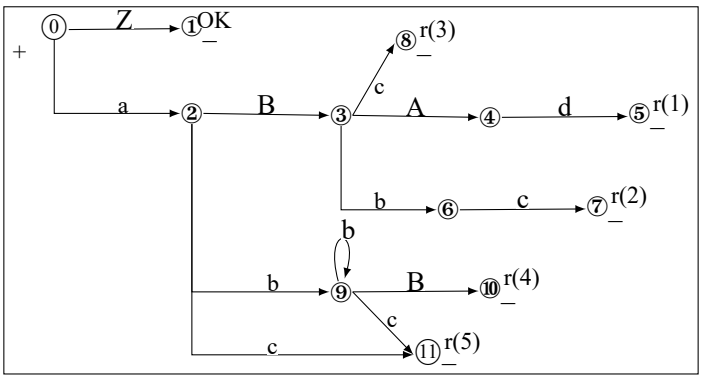



图 5.22: 自动机

## 5.8 LR(0) 分析器设计

有了自动机（句柄识别器）后，就能做自底向上的分析，就可以构建一个 LR(0) 分析器。LR(0) 分析器跟 LL(1) 分析器非常像，由两部分构成：第一部分，分析表，即自动机，涵盖了识别句柄也就是活前缀图的信息；第二部分是控制程序，程序要读取分析表，同时根据用户输入的词串进行分析。再强调一下，LR(0) 中的 0，是指不必查看当前符号就可确认句柄的意思。LR(0) 分析法要求文法应是 LR(0) 文法。

LR(0) 分析表 ⊕ LR(0) 控制器

图 5.24: LR(0) 分析器的基本组成

### 5.8.1 LR(0) 文法及其判定

满足下述特点的文法称为 LR(0) 文法：

1. 句柄识别器中，移进和归约不冲突；即移进和归约不同时发生。
2. 归约时不查看当前符号。

### 5.8.2 LR(0) 分析表构造

LR() 分析表是 LR() 分析法的知识表，是句柄识别器的一种机内表示形式：纵轴表示既有终结符和‘#’，又有非终结符，因为 LR() 分析里状态的转移之间读入的符号，既可能是终结符，也可能是非终结符。横轴表示的是状态编码，代表对状态的编号，0 表示起始。

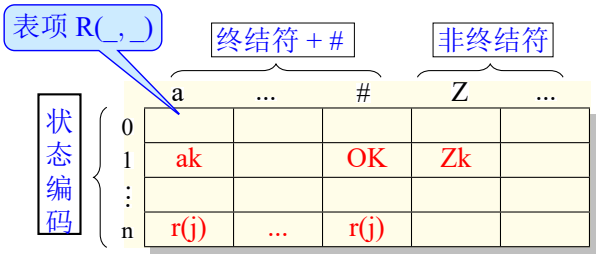



图 5.25: LR(0) 分析表

接下来要根据句柄识别器，填写表格里的表项。如果当前状态是 i，读入 x，到达 k，即 $\delta(i,x)=k$ 就把 xk 填到相应的表项里，x 既可以是终结符也可以是非终结符，k 表示的是到达的目标状态。如果是归约态，则

把这一行全写成归约，不管后面出现什么符号，都要归约。如果在 1 状态看到“#”，就是 ok。用形式化的语言描述如下：

【算法】
(1) 扩展文法，构造句柄识别器；
(2) 根据句柄识别器，填写LR(0) 分析表:
① 若 $\delta$(i,x)=k, x∈($V_N + V_T$), 则 R(i,x) := xk ;
② 若状态 i 标记有（-,r(j)），
则对任何 a ∈($V_T$ + #), R(i,a):=r(j) ;
③ R(1, #) := OK 。

图 5.26: LR(0) 分析表构建算法描述

在下面这个文法上实操一下

**文法定义**

$Z' \to Z_1$ ⓪ , $Z \to a_2B_3A_4d_5$ ① , $A \to b_6c_7$ ②$|c_8$ ③, $B \to b_9B_{10}$④ $|c_{11}$ ⑤

他的自动机图5.23我们已经在上面给出过，其分析表按算法填完如下图：

| ＼ | a | b | c | d | # | Z | A | B |
|---|---|---|---|---|---|---|---|---|
| 0 | a2 | | | | | Z1 | | |
| 1 | | | | | OK | | | |
| 2 | | b9 | c11 | | | | | B3 |
| 3 | | b6 | c8 | | | | A4 | |
| 4 | | | | d5 | | | | |
| 5 | r(1) | r(1) | r(1) | r(1) | r(1) | | | |
| 6 | | | | c7 | | | | |
| 7 | r(2) | r(2) | r(2) | r(2) | r(2) | | | |
| 8 | r(3) | r(3) | r(3) | r(3) | r(3) | | | |
| 9 | | b9 | c11 | | | | | B10 |
| 10 | r(4) | r(4) | r(4) | r(4) | r(4) | | | |
| 11 | r(5) | r(5) | r(5) | r(5) | r(5) | | | |

图 5.27: 分析表实例

### 5.8.3 LR(0) 控制程序设计

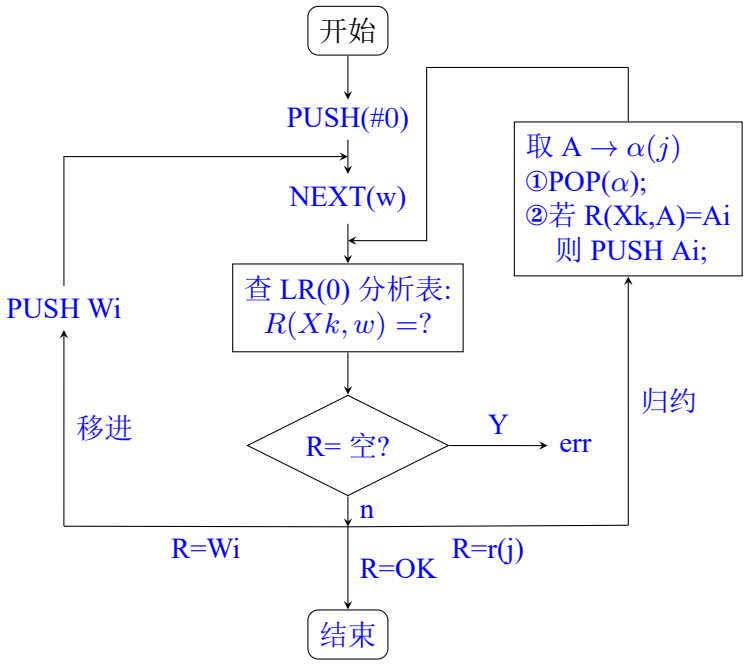



图 5.28: 控制程序流程图

## 5.9 项目集和可归约前缀图

下面讲另一种句柄识别器的构造方法，即用项目集的方式构造。

### 5.9.1 扩展文法

假设有一个文法 G(Z)，要构造它对应的句柄识别器，有以下的步骤：先将原始文法加入位置信息和替换开始符号，形成扩展文法。加入 Z'->Z，引入 Z'，目的是让程序能够启动。这个技巧在递归子程序也讲过，起始的主程序要读入一个字符，保证进入子程序之前已经读了一个字符。接着对所有产生式右部的所有符号进行编号，包括终结符和非终结符，编号没有特别的要求，可以任意编号，编号只代表状态。

G(Z): Z → aBAd
A → bc | c
B → bB | c

图 5.29: 原始文法

$Z^{'} \rightarrow Z_1(0)$
$Z \rightarrow a_2B_3A_4d_5(1)$
$A \rightarrow b_6c_7(2)|c_8(3)$
$B \rightarrow b_9B_10(4)|c_11(5)$

图 5.30: 扩展文法

### 5.9.2 由扩展文法构造可归约前缀图（句柄识别器）

首先把 Z'→ • Z 放到项目集里, • 后面是非终结符 Z，把 Z 的产生式 $Z \rightarrow \cdot aBAd$ 写到下面，形成 $I_0$ 状态。

$I_0$ 状态把 · 后移一位，可以读入 Z 或者 a，读入 Z 形成 Z'→ $Z\cdot$，也就是 $I_1$ 状态，我们把这个状态称作 OK 状态, 代表解析正常结束，是一个结束态。

对于 $I_0$ 状态还可以读入 a，形成 $I_2$ 状态，接下来希望看到 B。因此，我们把 B 的产生式附着在 $I_2$ 后面，形成了 $I_2$ 状态。

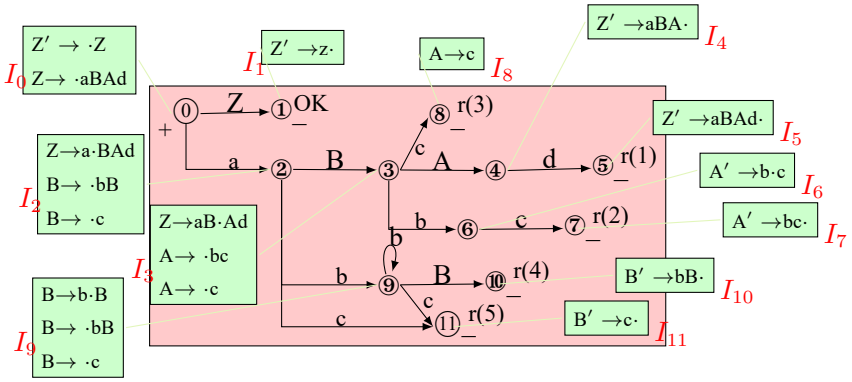



图 5.31: 项目集图

对于 $I_2$ 状态，现在可以读入 B，形成 $I_3$ 状态，遇见 B 之后希望看到 A，所以把 A 的表达式也接在 $I_3$ 后面，形成了 $I_3$ 状态。

我们通过四个状态的形成的例子，表达了前缀图的形成原则，· 后紧跟着的非终结符或终结符是我们希望预见的符号，所以如果是非终结符，我们需要把他的产生式加入新的状态中，因为是归约，是从句柄慢慢回到开始符号，句柄在句法树的底端都是终结符。如果是终结符，直接后移预见符号即可。最后形成

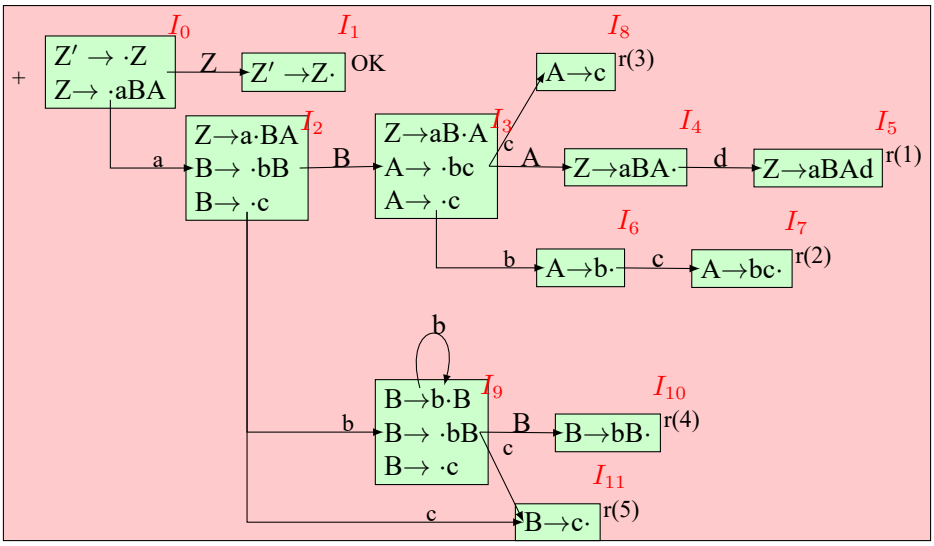



图 5.32: 由前缀图形成的自动机

### 5.9.3 由可归约前缀图构造 LR(0) 分析表

得到这样一个自动机后，就可以构造 LR() 分析表。0 状态看到 Z 到 1 状态，就把“Z1”写到 0 行 Z 列所在的表项。1 状态看到 #，代表结束，即 OK。5 状态稍微麻烦一点，I5 表示对产生式（1）进行归约，就把这一行全写成“r(1)”，表示不看当前字符，全部进行归约。其它同理。

| | a | b | c | d | # | Z | A | B |
|---|---|---|---|---|---|---|---|---|
| 0 | a2 | | | | | Z1 | | |
| 1 | | | | | OK | | | |
| 2 | | b9 | c11 | | | | | B3 |
| 3 | | b6 | c8 | | | | A4 | |
| 4 | | | | d5 | | | | |
| 5 | r(1) | r(1) | r(1) | r(1) | r(1) | | | |
| 6 | | | | c7 | | | | |
| 7 | r(2) | r(2) | r(2) | r(2) | r(2) | | | |
| 8 | r(3) | r(3) | r(3) | r(3) | r(3) | | | |
| 9 | | b9 | c11 | | | | | B10 |
| 10 | r(4) | r(4) | r(4) | r(4) | r(4) | | | |
| 11 | r(5) | r(5) | r(5) | r(5) | r(5) | | | |

图 5.33: LR(0) 分析表

### 5.9.4 LR(0) 分析法过程示例

LR() 分析是一个移进归约算法，有一个栈，只有两个操作，一是往栈里压入一个符号，二是对栈顶符号进行归约。通过自动机，就能知道用哪个操作。首先，先压入“#0”，表示起始状态，然后读入一个符号到 w，注意 w 是待处理的下一个符号，还未压入栈中。剩余串为“bccd#”。0 状态看到 a，通过查分析表，到 2 状态，做移进操作 PUSH(a2)，a 是待处理的符号，2 是当前状态编号。a 被压入栈了，当前 w 为空，所以读入下一个符号 b，剩余串为“ccd#”。2 状态看到 b，查表，到 9 状态，则 PUSH(b9)，并读入下一个字符 c，剩余串为“cd#”。当前状态是 9 状态，看到 c，到 11 状态，所以 PUSH(c11)。11 状态是一个归约态，要用产生式（5）B->c 进行归约，即把栈顶的 c11 归约成 B。c 被归约后，弹出栈，当前栈顶是 9 号状态，看到被归约后的 B，查表得 B10，写到栈顶。现在到了 10 状态，仍然是归约态，用产生式（4）B→bB 归约，即把“b9B10”弹出，在 2 状态看到被归约出的 B，查表得 B3，到 3 状态。3 状态看到 c，查表得 c8，做移进操作，PUSH(c8)。接下来的分析同理，这里不一一赘述。

| 分析栈 | w | 剩余串 | 操作 |
|---|---|---|---|
| #0 | a | bccd# | PUSH(a2),NEXT(w) |
| #0 a2 | b | ccd# | PUSH(b9),NEXT(w) |
| #0 a2 b9 | c | cd# | PUSH(c11),NEXT(w) |
| #0 a2 b9 c11 | c | d# | REDUCE(5) |
| #0 a2 b9 B10 | c | d# | REDUCE(4) |
| #0 a2 B3 | c | d# | PUSH(c8),NEXT(w) |
| #0 a2 B3 c8 | d | # | REDUCE(3) |
| #0 a2 B3 A4 | d | # | PUSH(d5),NEXT(w) |
| #0 a2 B3 A4 d5 | # | | REDUCE(1) |
| #0 Z1 | # | | OK |

图 5.34: LR(0) 分析过程

### 5.9.5 LR(0) 分析法实例

做一个小练习，对于黄色框的文法，构造可归约前缀图。

✽构造下述文法的 LR(0) 分析表:

G(Z): $Z \to aAb, A \to cA|d$

(1) 扩展文法: G′(Z′) $Z' \to Z_1(0)$
(2) 构造句柄识别器: $Z \to a_2A_3b_4(1)$
(3) 构造可归约前缀图: $A \to c_5A_6(2)|d_7(3)$

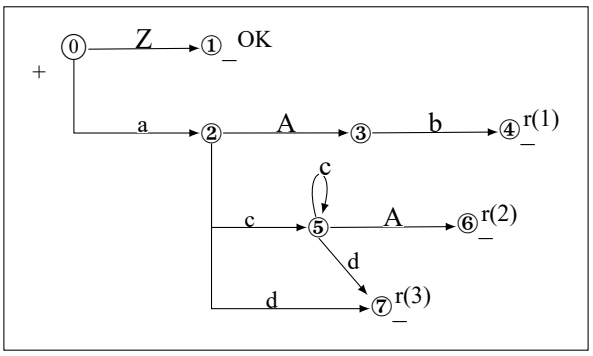



图 5.35: 句柄识别器

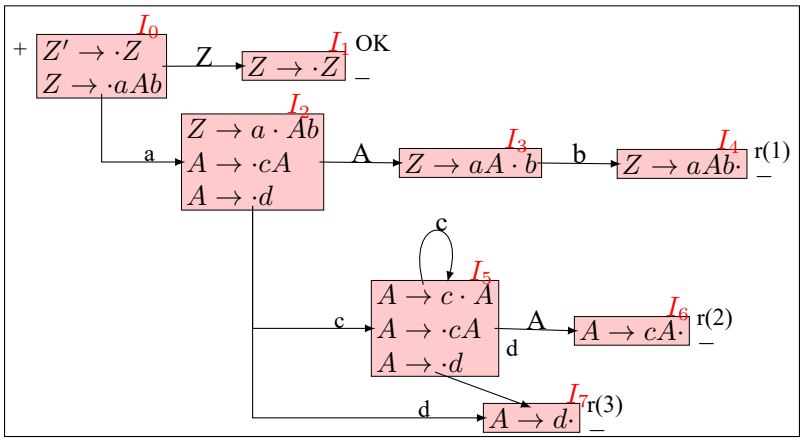



图 5.36: 可归约前缀图

| | a | b | c | d | # | Z | A |
|---|---|---|---|---|---|---|---|
| 0 | a2 | | | | | Z1 | |
| 1 | | | | | OK | | |
| 2 | | | c5 | d7 | | | A3 |
| 3 | | b4 | | | | | |
| 4 | r(1) | r(1) | r(1) | r(1) | r(1) | | |
| 5 | | | c5 | d7 | | | A6 |
| 6 | r(2) | r(2) | r(2) | r(2) | r(2) | | |
| 7 | r(3) | r(3) | r(3) | r(3) | r(3) | | |

图 5.37: LR(0) 分析表

## 5.10 LR(0) 分析法的扩展

LR() 分析要求文法是 LR() 文法，LR() 文法有两个最基本的要求。第一，对于任何一个状态，不能产生移进/归约冲突，即不能既有移进操作又有归约操作。也不能有归约/归约冲突，即一个终止状态有两个产生

式可以归约，这样在实现系统的时候是无法执行的，因为不知道是归约还是移进。

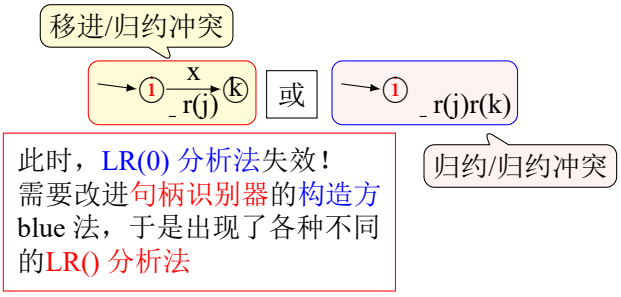



图 5.38: 句柄识别器的冲突

此时，LR(0) 分析法失效，因为产生了歧义，需要改造句柄识别器的构造方法。第一种改造方法，把 LR(0) 分析表进行简单的扩展，称为 SLR(1) 分析法。如何把 LR(0) 简单地扩展一下，变成 SLR(1) 方法呢？从字面意思理解，S 表示简单，没有太复杂的操作，括号里的 1 表示要看待处理字符 w 才知道要做什么操作。

看一个两个非终结符 Z、A 的文法例子

**文法定义**

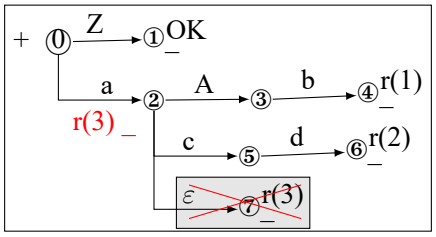
$Z \to aAb, A \to cd|\varepsilon,$

1. 扩展文法，构造句柄识别器

写成如黄色框所示的扩展形式，画成自动机。

$G'(Z')$ :

$Z' \to Z_1(0)$ ;
$Z \to a_2 A_3 b_4$
$A \to c_5 d_6(2)|\varepsilon_7(3)$




这个自动机跟以前的自动机相比，多了一个空（$\varepsilon$），2 状态看到空串到 7 状态，是一个归约产生式（3）的状态。从 2 状态到 7 状态不需要读符号，进行确定化之后，可以把 7 状态删除，将 7 状态进行归约的操作放到 2 状态执行，即 2 状态是一个终止态，要归约产生式（3）。这时候就产生了移进/归约冲突，2 状态既可以移进 c，也可以归约产生式（3）。这样当到 2 状态的时候，就不知道该到 3 状态或 5 状态，还是直接归约产生式（3）。

∵ ②：{c5,r(3),A3}
∴ 称为 移进/归约冲突！

图 5.39: 移进/归约冲突

这里的解决方法有两种思路，一是变换文法，但是在 LR() 分析里变换文法并不方便，因为变换文法后的自动机并不能直接看出来。二是改变方法，在 2 状态的时候，一个关键的决策是如何知道要归约产生式（3）?

在 LR(0) 的时候，不需要看待处理的符号 w 再进行归约。反过来说，如果能够看 w，就能知道是不是要进行归约。这里思考一下，w 是什么？是归约完的符号后面跟的符号，比如，w 是 a，栈顶如果能归约成 A 的话，A 后面马上就能跟 a，所以 w 是归约的左部（非终结符）的 follow 集的元素。在 2 状态的时候，如果想用规则（3）进行归约，则要求下一个符号是规则（3）左部对应的 follow 集，即 b。如果在 2 状态要归约产生式（3）到 A，下一个符号只能是 b。2 状态能移进的符号只有 c，到 5 状态。反过来说，b 代表要进行归约操作，c 代表要移进下一个符号。Follow 集跟移进操作的集合不一样。

∗ 通过查看“当前单词”，是否可以解决？为此：
求：$follow(A)$ =b　看到：$\{b\} \cap \{c\} = \Phi$;
∴ 若 w=c 则 c5; 若 w=b 则 r(3)。

图 5.40: 解决方法

2. 扩展句柄识别器，构造 SLR(1) 分析表

把这个过程用项目集的方式写出来。图中红色的部分 $A \to \cdot\varepsilon$，可直接写出归约。对于 $I_2$ 状态，有两种操作，看到 b 则归约，看到 c 则移进。这种方式称为 SLR(1) 文法，在 2 状态需要看 w 才能决定要不要归约，其它状态不需要看。

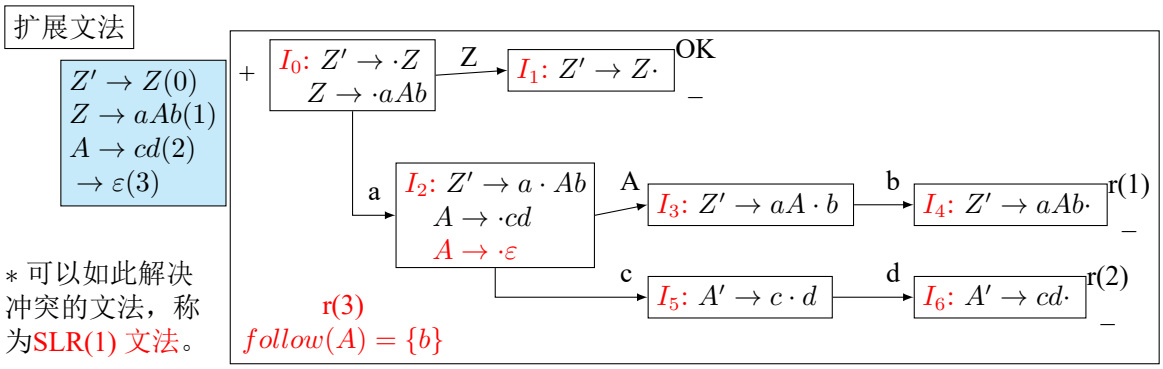



图 5.41: 可归约前缀图

它的分析表比较好写，唯一要注意的是 2 状态，之前的分析表是一整行都要写归约，因为不需要看 w。在 SLR(1) 里，需要看 w，当 w 是 b 的时候，归约产生式（3），是 c 的时候移进。注意这里与 LR(0) 分析表的区别，当是归约态是，LR(0) 里一整行都要归约，而 SLR(1) 不是。其它部分与 LR(0) 一样。

$$G(S): S \to aAd|aec|bAc|bed;\ A \to e$$

图 5.42: 分析表

## 5.11 SLR(1) 分析法的扩展

尽管查看当前单词，SLR(1) 方法仍然对许多文法的 LR(0) 项目集合中存在的状态冲突不能解决。比如，产生移进/归约冲突，follow 集里的元素包括移进的元素，这时候就不能说要归约该元素，因为有可能要移进。如果集合带有交集，就要对 SLR(1) 进一步进行扩展。看一个例子。

SLR(1) 分析表

| | a | b | c | d | # | Z | A |
|---|---|---|---|---|---|---|---|
| 0 | a2 | | | | | Z1 | |
| 1 | | | | | OK | | |
| 2 | | r(3) | c(5) | | | | A3 |
| 3 | | b4 | | | | | |
| 4 | r(1) | r(1) | r(1) | r(1) | r(1) | | |
| 5 | | | | d6 | | | |
| 6 | r(2) | r(2) | r(2) | r(2) | r(2) | | |

注意与 LR(0) 分析表的区别！

### 5.11.1 扩展文法

$G'(S')$ :

$S' \to S(0)$ ;
$S \to aAd\ (1) \mid aec\ (2)$
$S \to bAc\ (3) \mid bed\ (4)$
$A \to e\ (5)$

### 5.11.2 构造可归约前缀图

首先是 S‘到 S，能归约出 S 的产生式有四个：（1）、（2）、（3）、（4），都写下来。对于 $I_0$ 状态，可以跳转到 3 个状态。第一种情况，可以看到 S，到 OK 状态，即结束态。第二种情况，看到 a，希望看到 a 的是 $S \to \cdot aAd, S \to \cdot acc$，把这两条产生式的 • 往后移一位，分别表示看到 a 后希望看到 Ad，以及看到 a 后希望看到 ec。A 是非终结符，所以把 A 的产生式写下来。对于 $I_2$ 状态，可能看到两种情况，A 或 e。如果看到 A，就到 I3 状态，再看到 d，则到结束态 $I_4$。如果看到 e，把 $S \to a \cdot ec, A \to \cdot e$ 的 • 往后移一位，得到状态 $I_5$。$I_5$ 这两个产生式含义不一样，$S \to ae \cdot c$ 表示希望看到 c，$A \to e \cdot$ 表示要进行归约，归约的是产生式（5）。再看到 c 就到 $I_6$ 状态。同理，在 $I_0$ 状态看到 b，也能得到类似的结果。

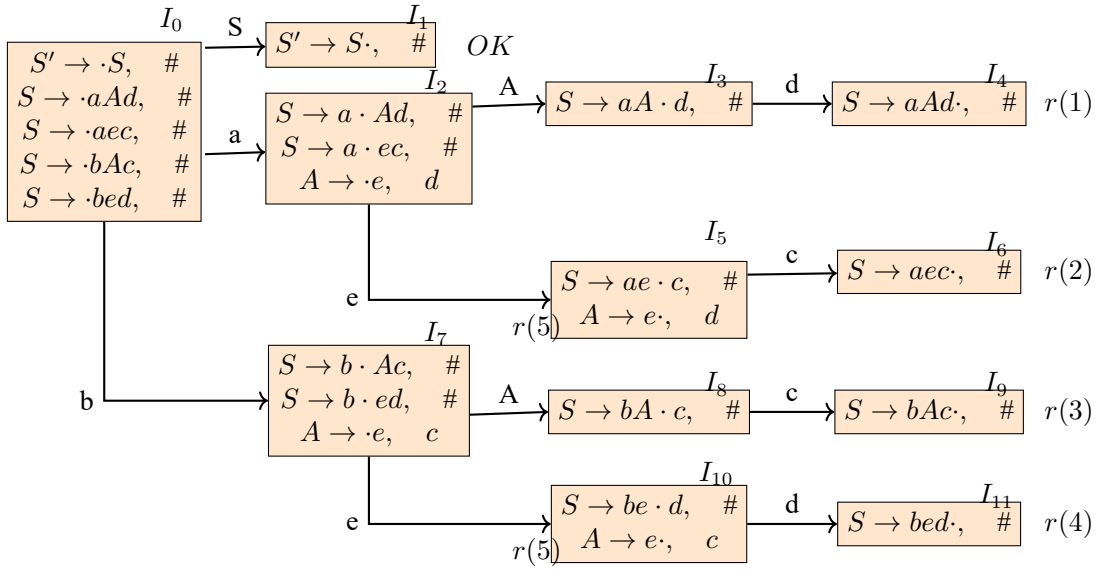



图 5.43: 可归约前缀图

这里出现的问题是，在 I5 和 I10 状态分别出现了冲突，既有归约又有移进。看归约产生式的左部的 follow 集是否跟移进操作相交，比如这里 follow(A)=c,d，即 $\{c\} \cap \{c,d\} \neq \emptyset, \{d\} \cap \{c,d\} \neq \emptyset$，不能用 SLR(1)，查看“当前单词”不能解决冲突。

解决思路：对有冲突的状态，仍然是查看‘当前单词’，但要看的更精确一点，以确定动作。（*）构造 LR(1) 可归约前缀图还是用项目集的方式，但是在产生式后面加一个符号，这个符号表示如果这个产生式被归约了，它后面应该出现什么符号，比如说 I0 状态第一条产生式 S'→·S, 如果 S 被归约成了 S‘，那么 S‘后面应该跟着 #。注意这里跟 follow 集不一样。如果 S 被推导完了，即 S'→S·，那么后面也跟着 #，如 I1 状态。在 I0 状态时，读入 a，产生式 S→·aAd 的 • 往后移一位，得 S→a·Ad，后面的符号也是 #。产生式 S→·aec 同理。对于 I2 项目集，S→a· Ad 中的 A 怎么得到？只有产生式（5）能归约出 A。如果 e 出现了，归约出 A，A 后面应该跟什么符号？A 后面只有跟 d，才能做推导，即 A→·e,d。在 $I_2$ 看到 e 之后，到 I5，并没有产生移进/归约冲突，因为 w 如果是 d，就归约 A→e·，如果不是，就移进 c。虽然 c 在 A 的 follow 集里，但不代表在 I5 状态里看到 c 就要进行归约，因为进行归约的前置条件是，在 I2 状态时 A 后面跟一个 d，因此，只有在 I5 状态里 A→e· 这个推导完后，必须跟 d，才需要进行归约。

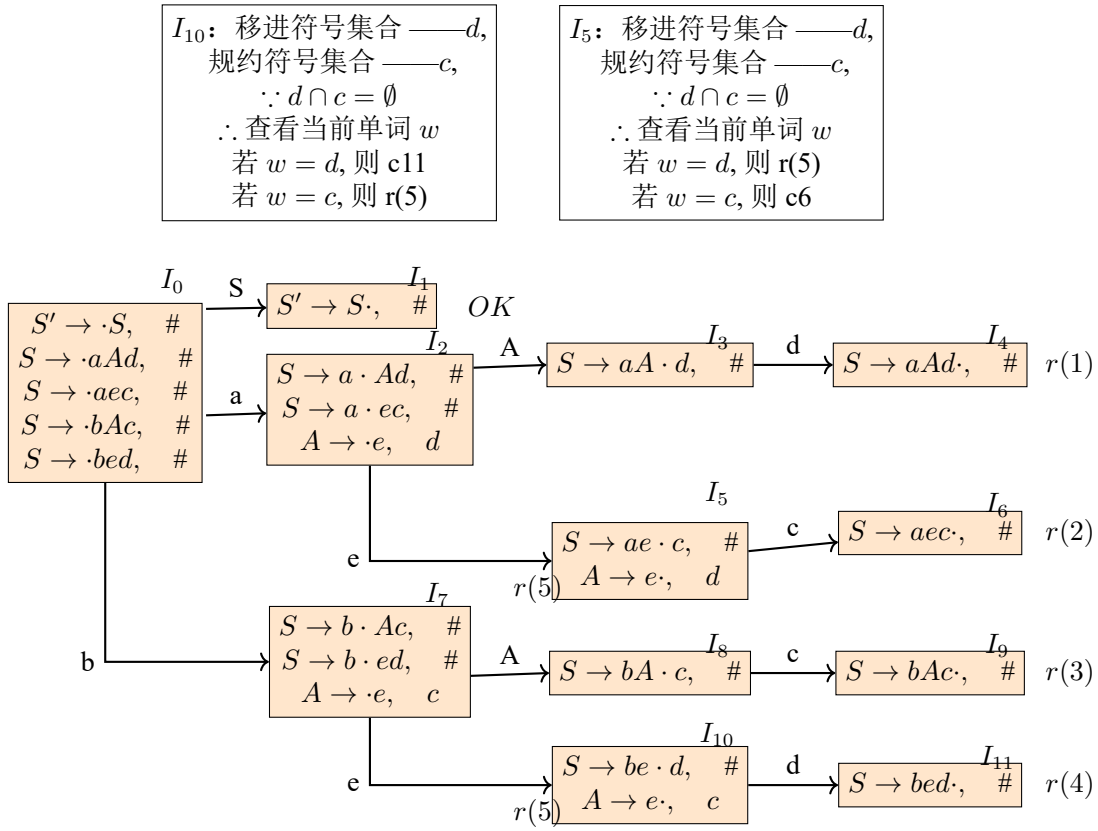



图 5.44: 完整的可归约前缀图

解决办法：

当项目 $I_j$(例如 $I_2$) 中同时含有 $A \to \alpha \cdot B\beta$ 和 $B \to \cdot Y$ 时，则向前看一个符号 $a$, 只有 $a \in First(\beta)$ 时，才能进行 $B \to Y$ 归约。

3. 构造 LR(1) 分析表

在每个有可能产生冲突的地方进行标注，在 LR(1) 里所有归约的地方，都要用一个符号来记录，看到 w 的哪个元素进行归约。红色里表达的意思是，我出现这个符号的时候需要进行归约。比较特殊的是 I5 和 I10 状态，分别看到 d 和 c 需要进行归约，否则就是 $I_5$ 读入 c 到 $I_6$，或 $I_{10}$ 读入 d 到 $I_{11}$。# 表示归约的条件，看到 # 就归约。LR(1) 并没有像 LR(0) 那样，如果归约一行全是 r，要多看一个 w。基于这样一个想法，就很容易把 LR(1) 分析表写出来，在 4 状态的时候，标准的 LR(0) 在一行全都要归约，但 LR(1) 要看 w，只有 w 是 # 的时候才需要归约。5 状态比较特殊，看到 c 的时候到 6 状态，看到 d 的时候归约产生式（5）。这里 LR(1) 一行不只有归约，还有移进，和 SLR(1) 一样。判断是移进还是归约，看后面可能跟的元素，在这个集合里就

归约，不在就移进。

## 5.12 简单优先分析法基本概念

### 5.12.1 什么是简单优先分析法

简单优先分析法是一种从左到右扫描、最左归约分析法。简单优先和 LR() 分析的策略是一样的，都是从左到右扫描，最左归约。它属于自底向上的分析方法。到现在已经了解了两种自下而上的分析方法，LR() 分析和简单优先分析。分析的目的是进行归约，要找句柄。简单优先分析法找句柄的方法与 LR() 分析不一样，LR() 是用句柄识别器，做一个自动机，进行归约。简单优先利用文法符号之间的优先关系来确定待归约的句柄，来确定当前句型的句柄。简单优先分析法的基本要点有三：

（1）利用一个分析表，登记选择句柄产生式的知识；

（2）利用一个分析栈，记录分析过程；

（3）“分析算法”依次读取单词，并进行如下操作：当栈顶出现句柄是，归约之，否则移进。所以简单优先分析也是一个移进归约的方法。

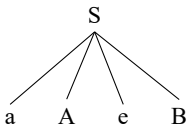
跟 LR() 分析不同的是判断句柄出现的方法，LR() 分析时根据句柄识别器的状态，简单优先分析要根据分析表里的一些规则或关系。

### 5.12.2 简单优先分析过程示例

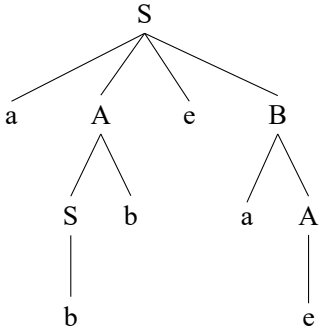
设有文法：

| G(S): | S→aAeB\|b |
|---|---|
| | A→Sb\|e |
| | B→aA |

现分析符号串 $\alpha = abbeae\#$，求分析树。首先，从 S 开始，由产生式 S->aAeB，可推导出




这里有两个非终结符，A 和 B，分别进行进一步的推导。A 推导出 S 和 b，S 进一步推导出 b；B 推导推导出 a 和 A，A 再推导出 e。



这棵树对应了文法推导的过程，如果是自下而上就是归约的过程。

利用分析栈记录行分析过程

首先，往栈里放一个 #，当前待处理的符号 w 是 a，栈顶不能归约，所以移进 a，w 读取下一个符号 b。还是没有句柄，继续移进 b，w 读 b，注意此时栈里是第一个 b，w 是第二个 b。此时出现句柄，b 要归约成 S。w 里还是 b，栈顶没有句柄了，移进，读取下一个符号 e。这时又出现句柄 Sb，归约。依次类推，不断进行归约操作，得到最终结果。

设 待分析的符号串:abbeae#

| 分析栈 | w | 剩余串 | 句柄产生式 | 操作 |
|---|---|---|---|---|
| # | a | b b e a e # | | 移进，NEXT |
| # a | b | b e a e # | | 移进，NEXT |
| # a b | b | e a e # | S→b | 归约 |
| # a S | b | e a e # | | 移进，NEXT |
| # a S b | e | a e # | A→Sb | 归约 |
| # a A | e | a e # | | 移进，NEXT |
| # a A e | a | e # | | 移进，NEXT |
| # a A e a | e | # | | 移进，NEXT |
| # a A e a e | # | | A→e | 归约 |
| # a A e a A | # | | B→aA | 归约 |
| # a A e B | # | | S→aAeB | 归约 |
| # S | # | | | OK |

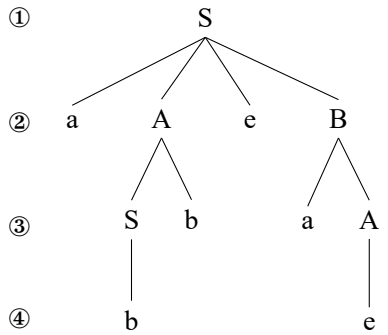
句柄

这个过程跟 LR() 分析的过程很相似，唯一区别在求句柄的地方。何时栈顶出现句柄？怎样求当前句柄产生式？

### 5.12.3 文法符号之间的优先关系

**归约过程中如何确认句柄?**

是否是句柄，还要看其所在符号串中的位置。句柄要在产生式的右部，产生式右部与树有什么样的关系呢？首先，树是有层次的，底下的符号要比上面的符号先归约。如下图，第④层要比第 ③层先归约，第 ③层比第②层先归约，第②层比第①层先归约。




从语法树上，找出优先关系（指相邻符号之间）如下：

①同时归约者为相等关系，记作 ≡

②左后归约者为小于关系，记作 $<\cdot$。优先关系的小于号，左侧晚于右侧归约。

③左先归约者为大于关系，记作 · >

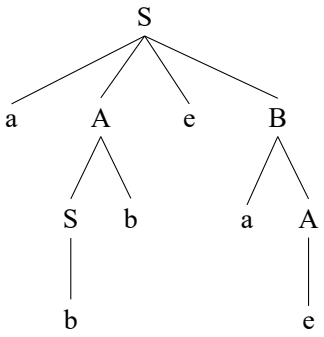
这些符号的左和右的含义并不是代数上的等号、小于号和大于号的含义，左和右分别表示连续出现的两个符号左边那个符号和右边那个符号。对于例子 b· >b，前一个 b 表示的是连续出现两个符号左边那个符号是 b 的情况，后一个 b 表示连续出现两个符号右边那个是 b 的情况。表示如果连续出现两个 b，左边的 b 要早于右边的 b 先归约。

假如知道了任意两个符号的大于、等于、小于关系，就能把关系符号填到一个串任意两个符号中间，特殊的一点是这里面最左边和最右边的 # 是约定，表示字符串的起始和结束。最左边和最右边的 # 比所有符号都小。

$$\# \lessdot a \lessdot b \gtrdot b \gtrdot e \lessdot a \lessdot e \gtrdot \#$$

图 5.45: 待分析符号串的优先关系

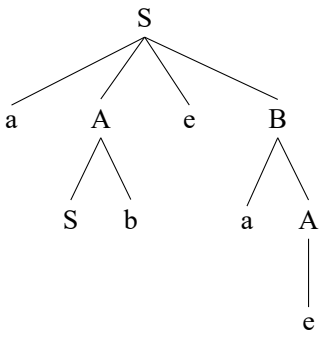
中间的画横线的部分，b 被夹在中间，这里的 b 对应语法树左下角的 b。b 与左侧的 a 是 < 的关系，与右侧的 b 是 > 的关系，意思是比左边（右边）先规约，即它应该先规约。




归约完后，b 变成了 S，即

$$\# \lessdot a \lessdot S \doteq b \gtrdot e \lessdot a \lessdot e \gtrdot \#$$

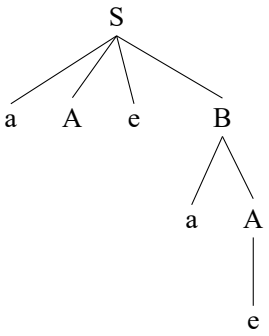
语法树变成：




左下角的 a 与 S 的关系是，S 比 a 要先归约，S 与 b 的关系是 ≡，同时归约，b 要先于上面的 e 归约，所以 S 和 b 要被归约，归约成 A，即

$$\# \lessdot a \doteq A \doteq e \lessdot a \lessdot e \gtrdot \#$$

得到的树如图：




此时被归约的是划横线的 e，归约成 A，接着再把 a、A、e、B 归约。

$$\# \lessdot a \doteq A \doteq e \lessdot a \doteq A \gtrdot \#$$
$$\# \lessdot a \doteq A \doteq e \doteq B \gtrdot \#$$

总结：一个句型的句柄，位于第一次（自左至右）出现在 < 和 > 之间的符号串，且要求该符号串之间的关系是 ≡。

### 优先关系的定义

文法

G(S): S→aAeB|b
A→Sb|e
B→aA

从文法中获取优先关系。设 $s_i$，$s_j$ 是两个文法符号。

①如果产生式右部连续出现的符号是 si 和 sj，那 si 和 sj 相等，即 $s_i \equiv s_j$。

②如果产生式右部推导出符号 si，si 后面跟着非终结符 W，W 经过若干次推导，推导出的符号是 sj，即 sj 要被归约成 W，就说 sj 是早于 si 归约的，即 $s_i < \cdots_j$。比如，S→aAeB|d 中，A 推导出的第一个符号要早于 a 归约。

③如果一个非终结符 V 右边跟着符号 sj，V 推导出的最后一个符号 si，si 要早于 sj 归约，即 $s_i > s_j$

① $S_i \doteq S_j$, 当且仅当有 U→...$s_i, s_j$...;
如: a≐A, A≐e, e≐B, S≐b;
② $S_i \lessdot S_j$, 当且仅当有 U→...$s_i$W..., 且 W$\xrightarrow{+}$ $s_j$...;
如:a⋖e, a⋖S, a⋖a, a⋖b, e⋖a.....
③ $S_i \gtrdot S_j$, 当且仅当有 U→...V$s_j$..., 且 V$\xrightarrow{+}$...$s_i$;
如：b⋗b, B⋗b, A⋗b, e⋗b。

图 5.46

### 头符号集合和尾符号集合

头符号集合指一个非终结符通过推导出现的第一个符号是什么。头符号集合不要求集合里的元素都是终结符，所有的符号都可以推导出。尾符号集合指非终结符推导出的串的最后一个符号是什么。这个定义和 select 集、first 集、follow 集是不一样的。在 LL(1) 分析的时候，select 集要求的是推导出来的终结符，头符号集合和尾符号集合没有这个要求，所有的符号都行。

设 A∈ $V_N$,$s_i, s_j$ 是两个文法符号；则：

FIRSTVT(A)= $\{s_i | A \overset{+}{\rightarrow} s_i.....\}$

LASTVT(A)= $\{s_j | A \overset{+}{\rightarrow} .....s_j\}$

图 5.47: 头符号集合和尾符号集合的形式化表述

**例 5.9**

文法 G(S): S→aAeB| b, A→ Sb|e, B→ aA

图 5.48: 文法定义

### 求头符号和尾符号集合

根据 $S \rightarrow aAeB|b$，S 的头符号集合是 a，b。B 和 b 属于 S 的尾符号集合，B 是一个非终结符，B 的尾符号集合也属于 S 的尾符号集合。根据 $B \rightarrow aA$，B 的尾符号集合是 A。A 也是一个非终结符，A 的尾符号集合也属于 S 的尾符号集合，A 的尾符号集合是 b，e。综上，S 的尾符号集合是 B，b，A，e。

根据 $A \rightarrow Sb|e$，S 和 e 属于 A 的头符号集合。S 是非终结符，S 的头符号集合 a，b 属于 A 的头符号集合。A 的尾符号集合是 b，e。

同理，B 的头符号集合是 a，尾符号集合是 A，b，e。

| | S | A | B |
|---|---|---|---|
| FIRSTVT | a,b | S,e,a,b | a |
| LASTVT | B,b,A,e | b,e | A,b,e |

### 优先矩阵

得到了头符号集合与尾符号集合之后，就可以求优先关系。优先关系体现在优先矩阵里，优先矩阵准确描述了任意两个符号之间的优先关系。纵轴表示关系符号左边的符号，横轴表示右边的符号。比如图中 S 所在行，A 所在列的表项，填的是左边出现 S ，紧接着右边出现 A，它们两个之间的优先关系。优先关系矩阵的横轴和纵轴是一样的，而且所有终结符和非终结符都要写上。

| | S | A | B | a | b | e |
|---|---|---|---|---|---|---|
| S | | | | | | |
| A | | | | | | |
| B | | | | | | |
| a | | | | | | |
| b | | | | | | |
| e | | | | | | |

图 5.49: 优先矩阵

如何填该表？根据优先关系的定义，产生式连续出现两个符号，就是 ≡；如果一个终结符，一个非终结符，非终结符的头符号集合和左边的终结符构成 < 关系，非终结符的尾符号集合和右边的终结符构成 > 关系。

具体看一下。首先看产生式 $S \rightarrow aAeB|b$，推出一个符号就不用考虑了，因为至少要两个符号。先看 aA，是 ≡ 关系，注意这里填的是 a 所在行，A 所在列，表示 a 在左边，A 在右边，而不是 A 所在行，a 所在列。A 是一个非终结符，a 跟在非终结符 A 后面，那 a 要晚于 A 的头符号集合归约，填上 <。

接着往后看 Ae，A 和 e 是 ≡ 关系，往 A 所在行，e 所在列的表项填上 ≡。左边是非终结符 A，右边是终结符 e，A 的尾符号集合要早于右边出现的 e 归约。A 的尾符号集合是 B，e，所以 B 和 e 要早于 e 归约。千万不要把左右两个符号的顺序弄反了，Ae 的 e 出现在右侧，所以这个 e 要按照纵列去查，而 A 的尾符号集合要按表中左侧的行去查。

这里有个问题，为什么 e>e？如果左边出现 e，右边同样出现 e，表示左边的 e 先归约，这里不是指两个相同符号的代数运算，表达的是优先关系。

同理，其它符号的关系也可以填上。

(2) 优先矩阵

表项 = 空表示两个符号不可能相邻。

| | S | A | B | a | b | e |
|---|---|---|---|---|---|---|
| S | | | | | = | |
| A | | | | | ⋗ | = |
| B | | | | | ⋗ | ⋗ |
| a | ⋖ | = | | ⋖ | ⋗ | ⋗ |
| b | | ⋗ | | | ⋗ | ⋗ |
| e | | | = | ⋖ | ⋗ | ⋗ |

图 5.50: 填好的优先矩阵

表中空的地方表示确定不了两个符号之间的优先关系，这个方法用不了。

## 5.13 简单优先分析器设计

有了优先矩阵表，就可以实现简单优先分析器。

简单优先分析器的基本组成:
优先矩阵分析表 ⊕ 优先分析控制器
✻分析中只查看当前符号就可确认句柄;
✻优先分析法要求文法应是简单优先文法。

简单优先虽然简单，但有适用的范围。

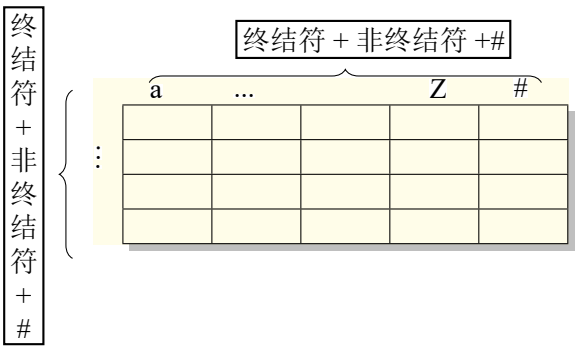
### 5.13.1 简单优先文法及其判定

满足下述特点的文法称为简单优先文法：①文法产生式没有相同的右部。不能同时有 A 推出 α 以及 B 推出 α。在归约的时候，看到的是一个赋值串，有一个句柄要归约，但在简单优先分析里不知道要归约成什么。要想知道用什么归约，只能到归约机里查，查到哪个产生式就用哪个产生式，如果查到两个产生式右部是相同的，就可以用两个产生式进行归约，就会不知道用哪一条来归约。

②文法符号之间至多有一种优先关系。

### 5.13.2 简单优先分析矩阵分析表构造

首先，制作表格如下




算法：①如果产生式右部连续出现两个符号，填 ≡

②如果产生式右部出现了一个非终结符 W，它和它左边的符号 si 就构成了 < 关系，所有 si 小于 W 的头符号集合的元素。

③如果产生式右部出现的非终结符 V，V 后面跟着一个符号 sj，V 的尾符号集合的元素和 sj 构成了 > 关系。

设 W,V∈ $V_N$; $s_i, s_j$ 是两个文法符号；

① $S_i = S_j$，
当且仅当有 U→...$s_i, s_j$...;

② $S_i \lessdot S_j$，
当且仅当有 U→...$s_i$W..., 且 $s_j \in FIRSTVT(W)$;

③ $S_i \gtrdot S_j$，
当且仅当有 U→...V$s_j$..., 且 $s_i \in LASTVT(V)$□

图 5.51: 简单优先分析矩阵分析表构造的形式化描述

### 5.13.3 简单优先控制程序设计

先把 # 压栈，再读一个符号 w，然后查表，看当前栈顶符号和 w 的优先关系。当前栈顶符号 Xk 是左边的符号，w 是右边的符号，查表里是否为空，如果为空表示出错，如果不为空就看是否是，表示的意思是 Xk 早于 w 归约，有可能出现句柄。如果不是，就判断是否要结束，即只剩“#S#”，如果还没结束，就接着移进。如果是，就要归约，找离最近的的中间部分，用 sisi+1...sj 表示，把 sisi+1...sj 弹出，用相应左部的符号替换。这里注意，归约完后，不能移进，w 只是帮助判断栈顶是否出现句柄，还未处理（压栈）。

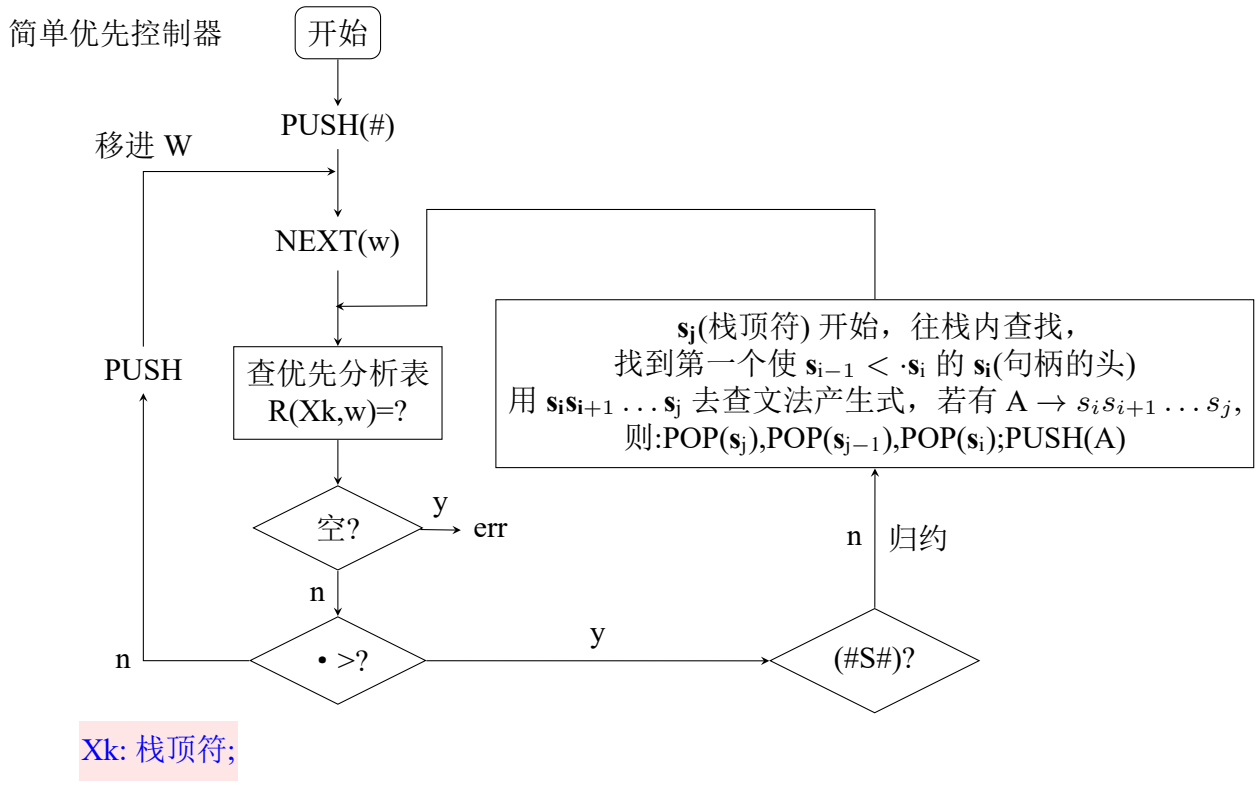



图 5.52: 简单优先控制程序

## 5.14 算符优先分析

### 5.14.1 算符文法

算符优先要求文法的右部不能连续出现两个非终结符，比如右部不能出现 A 后面跟着 B，这样就不能叫算符优先。如果文法里产生式右部没有连续出现的非终结符，就有可能是算符优先文法。

设有一文法 G，如果 G 中没有形如 A→...QR... 的产生式，其中 Q 和 R 为非终结符，则称该文法为算符违法（OG 文法）。

### 5.14.2 头符号集合和尾符号集合

如何定义算符优先？我们重新定义一下头符号集合和尾符号集合。头符号集合不只包含 P 推导出来的串首终结符，而且包含推导出来的串首是非终结符的终结符，例如，A->aB，A=>abB，A=>Bac，这三个产生式的 a 都属于头符号集合，换句话说，不需要考虑非终结符，来确定头符号集合。尾符号集合同理，不考虑非终结符。

设 a∈ $V_T$，P,R∈ $V_N$，则:

FIRSTVT(P)= $\{a|P \to a.....\square P \to Ra.....\}$,
LASTVT(P)= $\{a|P \to .....a\square P \to .....aR\}$。

图 5.53: 头符号集合和尾符号集合

### 5.14.3 算符优先关系定义

跟上面一样，两个符号连续出现就是 ≡。如果出现了像 aQb 这种情况，a 和 b 的关系也是 ≡。注意，在算符优先文法里，不考虑非终结符，可以看成 a 和 b 是连续的。

< 除了要考虑 p→...aR...，还要考虑头符号集合的定义和尾符号集合的定义。

设 a∈ $V_T$, P,Q,R∈ $V_N$,

①a=b,
　当且仅当有 P→...ab... 或 P→...aQb...;
②a<b,
　当且仅当有 P→...aR..., 且 $R \xrightarrow{+} b...$ 或 $R \xrightarrow{+} Qb...$;
③a>b,
　当且仅当有 P→...Rb..., 且 $R \xrightarrow{+} ...a$ 或 $R \xrightarrow{+} ...aQ$;

图 5.54: 算符优先关系定义

### 5.14.4 算符优先文法

如果算符文法 G 中的任何一对终结符 a 和 b 之间，仅满足上述一种关系，则 G 就是一个算符优先文法（OPG）。

**例 5.10** 求文法 G(E) 是简单优先文法吗？先写出头符号集合和尾符号集合，然后把优先矩阵写出来。

文法 G（E）：E → T|E+T, T → F|T*F, F→ i|(E)

| | E | T | F |
|---|---|---|---|
| FIRST | E,T,F,i,( | T,F,i,( | i,( |
| LASTVT | T,F,i,) | F,i,) | i,) |

图 5.55: 头符号集合和尾符号集合

| | E | T | F | + | * | i | ( | ) |
|---|---|---|---|---|---|---|---|---|
| E | | | | = | | | | = |
| T | | | | > | = | | | > |
| F | | | | > | > | | | > |
| + | | =< | < | | | < | < | |
| * | | | = | | | < | < | |
| i | | | | > | > | | | > |
| ( | < | < | < | | | < | < | |
| ) | = | | | > | > | | | > |

文法符号之间优先关系不唯一！

图 5.56: 优先矩阵

简单优先文法有两个条件，不能有相同右部的产生式，任意两个符号之间的关系只能有一种。矩阵有不唯一的关系，所以文法 G(E) 不是简单优先文法。