第4章 语法分析

本章介绍的语法分析方法通常用于编译器中。我们首先介绍基本概念，然后介绍适合手工实现的技术，最后介绍用于自动化工具的算法。因为源程序可能包含语法错误，所以我们还将讨论如何扩展语法分析方法，以便从常见错误中恢复。

在设计语言时，每种程序设计语言都有一组精确的规则来描述良构(well-formed)程序的语法结构。比如，在C语言中，一个程序由多个函数组成，一个函数由声明和语句组成，一个语句由表达式组成，等等。程序设计语言构造的语法可以使用2.2节中介绍的上下文无关文法或者BNF(巴库斯－瑙尔范式)表示法来描述。文法为语言设计者和编译器编写者都提供了很大的便利。

- 文法给出了一个程序设计语言的精确易懂的语法规约。
- 对于某些类型的文法，我们可以自动地构造出高效的语法分析器，它能够确定一个源程序的语法结构。同时，语法分析器的构造过程可以揭示出语法的二义性，同时还可能发现一些容易在语言的初始设计阶段被忽略的问题。
- 一个正确设计的文法给出了一个语言的结构。该结构有助于把源程序翻译为正确的目标代码，也有助于检测错误。
- 一个文法支持逐步加入可以完成新任务的新语言构造从而迭代地演化和开发语言。如果对语言的实现遵循语言的文法结构，那么在实现中加入这些新构造的工作就变得更加容易。

4.1 引论

在本节中，我们将探讨语法分析器是如何集成到一个典型的编译器中的。然后我们将研究算术表达式的典型文法。通过表达式文法已经足以说明语法分析的本质，因为处理表达式的语法分析技术可以用于处理程序设计语言的大部分构造。这一节的最后将讨论错误处理的问题，因为当语法分析器发现它的输入不能由它的文法生成时，它必须作出适当的反应。

4.1.1 语法分析器的作用

在我们的编译器模型中，语法分析器从词法分析器获得一个由词法单元组成的串，并验证这个串可以由源语言的文法生成，如图4-1所示。我们期望语法分析器能够以易于理解的方式报告语法错误，并且能够从常见的错误中恢复并继续处理程序的其余部分。从概念上讲，对于良构的程序，语法分析器构造出一棵语法分析树，并把它传递给编译器的其他部分进一步处理。实际上，并不需要显式地构造出这棵语法分析树，因为正如我们将看到的，对源程序的检查和翻译动作可以和语法分析过程交错完成。因此，语法分析器和前端的其他部分可以用一个模块来实现。

处理文法的语法分析器大体上可以分为三种类型：通用的、自顶向下的和自底向上的。像Cocke-Younger-Kasami算法和Earley算法这样的通用语法分析方法可以对任意文法进行语法分析(见参考文献)。然而，这些通用方法效率很低，不能用于编译器产品。

编译器中常用的方法可以分为自顶向下的和自底向上的。顾名思义，自顶向下的方法从语法分析树的顶部(根结点)开始向底部(叶子结点)构造语法分析树，而自底向上的方法则从叶子结点开始，逐渐向根结点方向构造。这两种分析方法中，语法分析器的输入总是按照从左向右的方式被扫描，每次扫描一个符号。

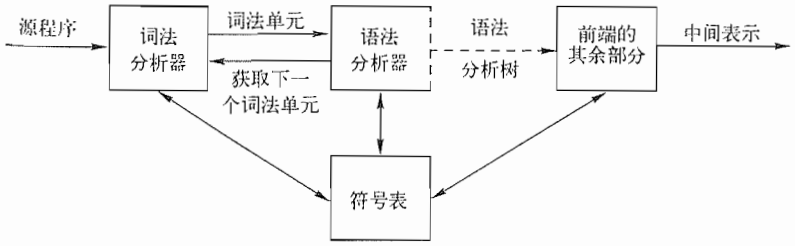

图 4-1 编译器模型中语法分析器的位置

最高效的自顶向下方法和自底向上方法只能处理某些文法子类，但其中的某些子类，特别是 LL 和 LR 文法，其表达能力已经足以描述现代程序设计语言的大部分语法构造了。手工实现的语法分析器通常使用 LL 文法。比如，2.4.2 节中的预测语法分析方法能够处理 LL 文法。处理较大的 LR 文法类的语法分析器通常是使用自动化工具构造得到的。

在本章中，我们假设语法分析器的输出是语法分析树的某种表示形式。该语法分析树对应于来自词法分析器的词法单元流。在实践中，语法分析过程中可能包括多个任务，比如将不同词法单元的信息收集到符号表中，进行类型检查和其他类型的语义分析，以及生成中间代码。我们把所有这些活动都归纳到图 4-1 中的“前端的其余部分”里面。在后续几章中将详细讨论这些活动。

4.1.2 代表性的文法

为了便于参考，我们先给出一些即将在本章中加以研究的文法。对那些以 **while** 或 **int** 这样的关键字开头的构造进行语法分析相对容易，因为关键字可以引导我们选择适当的文法产生式来匹配输入。因此我们主要关注表达式。因为运算符的结合性和优先级，表达式的处理更具挑战性。

下面的文法指明了运算符的结合性和优先级。这个文法和我们在第 2 章中使用的描述表达式、项和因子的文法类似。E 表示一组以 + 号分隔的项所组成的表达式；T 表示由一组以 * 号分隔的因子所组成的项；而 F 表示因子，它可能是括号括起的表达式，也可能是标识符：

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned} \tag{4.1}$$

表达式文法(4.1)属于 LR 文法类，适用于自底向上的语法分析技术。这个文法经过修改可以处理更多的运算符和更多的优先级层次。然而，它不能用于自顶向下的语法分析，因为它是左递归的。

下面给出表达式文法(4.1)的无左递归版本，该版本将被用于自顶向下的语法分析：

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow + \ TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow * \ FT' \mid \epsilon \\ F &\rightarrow (\ E \ ) \mid \mathbf{id} \end{aligned} \tag{4.2}$$

下面的文法以相同的方式处理 + 和 *，因此它可以用来说明那些在语法分析过程中处理二义性的技术：

$$E \rightarrow E \ + \ E \mid E \ * \ E \mid (\ E \ ) \mid \mathbf{id} \tag{4.3}$$

这里的 E 表示各种类型的表达式。文法(4.3)允许一个表达式，比如 $a+b*c$，具有多棵语法分析树。

4.1.3 语法错误的处理

本节的其余部分将考虑语法错误的本质以及错误恢复的一般策略。其中的两种策略分别称为恐慌模式和短语层次恢复。它们将和特定的语法分析方法一起详细讨论。

如果编译器只处理正确的程序，那么它的设计和实现将会大大简化。但是，人们还期望编译器能够帮助程序员定位和跟踪错误。因为不管程序员如何努力，程序中难免会有错误。令人惊奇的是，虽然错误如此常见，但很少有语言在设计的时候就考虑到错误处理问题。如果我们的口语也像计算机语言那样对语法精确性有要求，那么我们的文明就会大不相同。大部分程序设计语言的规范没有规定编译器应该如果处理错误；错误处理方法由编译器的设计者决定。从一开始就计划好如何进行错误处理不仅可以简化编译器的结构，还可以改进错误处理方法。

程序可能有不同层次的错误。

- *词法错误*，包括标识符、关键字或运算符拼写错误(比如把标识符 `ellipseSize` 写成 elipseSize) 和没有在字符串文本上正确地加上引号。
- *语法错误*，包括分号放错地方、花括号，即“{”或“}”，多余或缺失 。另一个 C 语言或 Java语言中的语法错误的例子是一个 `case` 语句的外围没有相应的 `switch` 语句(然而，语法分析器通常允许这种情况出现，当编译器在之后要生成代码时才会发现这个错误)。
- *语义错误*，包括运算符和运算分量之间的类型不匹配。例如，返回类型为 `void` 的某个 Java方法中出现了一个返回某个值的 `return` 语句。
- *逻辑错误*，可以是因程序员的错误推理而引起的任何错误。比如在一个 C 程序中应该使用比较运算符 == 的地方使用了赋值运算符 = 。这样的程序可能是良构的，但是却没有正确反映出程序员的意图。

语法分析方法的精确性使得我们可以非常高效地检测出语法错误。有些语法分析方法，比如 LL 和 LR 方法，能够在第一时间发现错误。也就是说，当来自词法分析器的词法单元流不能根据该语言的文法进一步分析时就会发现错误。更精确地讲，它们具有*可行前缀特性*(viable-prefix property)，也就是说，一旦它们发现输入的某个前缀不能够通过添加一些符号而形成这个语言的串，就可以立刻检测到语法错误。

要重视错误恢复的另一个原因是，不管产生错误的原因是什么，很多错误都以语法错误的方式出现，并且在不能继续进行语法分析时暴露出来。有些语义错误（比如类型不匹配）也可以被高效地检测到。然而，总的来说，在编译时精确地检测出语义错误和逻辑错误是很困难的。

语法分析器中的错误处理程序的目标说起来很简单，但实现起来却很有挑战性：

- 清晰精确地报告出现的错误。
- 能很快地从各个错误中恢复，以继续检测后面的错误。
- 尽可能少地增加处理正确程序时的开销。

幸运的是，常见的错误都很简单，使用相对直接的错误处理机制就足以达到目标。

一个错误处理程序应该如何报告出现的错误？至少，它必须报告在源程序的什么位置检测到错误，因为实际的错误很可能就出现在这个位置之前的几个词法单元处。一个常用的策略是打印出有问题的那一行，然后用一个指针指向检测到错误的地方。

4.1.4 错误恢复策略

当检测到一个错误时，语法分析器应该如何恢复？虽然还没有哪个策略能够证明自己是被

普遍接受的，但有一些方法的适用范围很广。最简单的方法是让语法分析器在检测到第一个错误时给出错误提示信息，然后退出。如果语法分析器能够把自己恢复到某个状态，且有理由预期从那里开始继续处理输入将提供有意义的诊断信息，那么它通常会发现更多的错误。如果错误太多，那么最好让编译器在超过某个错误数量上界之后停止分析。这样做要比让编译器产生大量恼人的“可疑”错误信息更好。

恐慌模式的恢复

使用这个方法时，语法分析器一旦发现错误就不断丢弃输入中的符号，一次丢弃一个符号，直到找到同步词法单元(synchronizing token)集合中的某个元素为止。同步词法单元通常是界限符，比如分号或者}。它们在源程序中的作用是清晰、无二义的。编译器的设计者必须为源语言选择适当的同步词法单元。恐慌模式的错误纠正方法常常会跳过大量输入，不检查被跳过部分的其他错误。但是它很简单，并且能够保证不会进入无限循环。我们稍后考虑的某些方法则不一定能保证不进入无限循环。

短语层次的恢复

当发现一个错误时，语法分析器可以在余下的输入上进行局部性纠正。也就是说，它可能将余下输入的某个前缀替换为另一个串，使语法分析器可以继续分析。常用的局部纠正方法包括将一个逗号替换为分号、删除一个多余的分号或者插入一个遗漏的分号。如何选择局部纠正方法是由编译器设计者决定的。当然，我们必须小心选择替换方法，以避免进入无限循环。比如，如果我们总是在当前输入符号之前插入符号，就会出现无限循环。

短语层次替换方法已经在多个错误修复型编译器中使用，它可以纠正任何输入串。它主要的不足在于它难以处理实际错误发生在被检测位置之前的情况。

错误产生式

通过预测可能遇到的常见错误，我们可以在当前语言的文法中加入特殊的产生式。这些产生式能够产生含有错误的构造，从而基于增加了错误产生式的文法构造得到一个语法分析器。如果语法分析过程中使用了某个错误产生式，语法分析器就检测到了一个预期的错误。语法分析器能够据此生成适当的错误诊断信息，指出在输入中识别出的错误构造。

全局纠正

在理想情况下，我们希望编译器在处理一个错误输入串时通过最少的改动将其转化为语法正确的串。有些算法可以选择一个最小的改动序列，得到开销最低的全局性纠正方法。给定一个不正确的输入 x 和文法 G，这些算法将找出一个相关串 y 的语法分析树，使得将 x 转换为 y 所需要的插入、删除和改变的词法单元的数量最少。遗憾的是，从时间和空间的角度看，实现这些方法一般来说开销太大，因此这些技术当前仅具有理论价值。

请注意，一个最接近正确的程序可能并不是程序员想要的程序。不管怎样，最低开销纠正的概念仍然提供了一个可用于评价错误恢复技术的指标，并已经用于为短语层次的恢复寻找最佳替换串。

4.2 上下文无关文法

2.2节中已经介绍了文法的概念。在那里，它用于系统地描述程序设计语言的构造（比如表达式和语句）的语法。下面的产生式使用语法变量 *stmt* 来表示语句，使用变量 *expr* 表示表达式。

$$stmt \rightarrow \textbf{if}\ (expr)\ stmt\ \textbf{else}\ stmt \tag{4.4}$$

上述产生式描述了这种形式的条件语句的结构。其他产生式则精确地定义了 *expr* 是什么，以及 *stmt* 可以是什么。

这一节将回顾上下文无关文法的定义，并介绍了在讨论语法分析技术时要用到的一些术语。特别地，推导的概念在讨论产生式在分析过程中的应用顺序时非常有用。

4.2.1 上下文无关文法的正式定义

根据2.2节的介绍可知，一个上下文无关文法(简称文法)由终结符号、非终结符号、一个开始符号和一组产生式组成。

1）*终结符号*是组成串的基本符号。术语“词法单元名字”是“终结符号”的同义词。当我们讨论的显然是词法单元的名字时，我们经常使用“词法单元”这个词来指称终结符号。我们假设终结符号是词法分析器输出的词法单元的第一个分量。在(4.4)中，终结符号是关键字 **if** 和 **else** 以及符号“(”和“)”。

2）*非终结符号*是表示串的集合的语法变量。在(4.4)中，*stmt* 和 *expr* 是非终结符号。非终结符号表示的串集合用于定义由文法生成的语言。非终结符号给出了语言的层次结构，而这种层次结构是语法分析和翻译的关键。

3）在一个文法中，某个非终结符号被指定为*开始符号*。这个符号表示的串集合就是这个文法生成的语言。按照惯例，首先列出开始符号的产生式。

4）一个文法的*产生式*描述了将终结符号和非终结符号组合成串的方法。每个产生式由下列元素组成：

① 一个被称为产生式*头*或*左部*的非终结符号。这个产生式定义了这个头所代表的串集合的一部分。

② 符号→。有时也使用::=来替代箭头。

③ 一个由零个或多个终结符号与非终结符号组成的产生式*体*或*右部*。产生式体中的成分描述了产生式头上的非终结符号所对应的串的某种构造方法。

例4.5 图4-2中的文法定义了简单的算术表达式。在这个文法中，终结符号是

id + - * / ( )

非终结符号是 *expression*、*term* 和 *factor*，而 *expression* 是开始符号。□

expression	→	*expression* + *term*
expression	→	*expression* - *term*
expression	→	*term*
term	→	*term* * *factor*
term	→	*term* / *factor*
term	→	*factor*
factor	→	(*expression*)
factor	→	**id**

图4-2 简单算术表达式的文法

4.2.2 符号表示的约定

为了避免总是声明“这些是终结符号”，“这些是非终结符号”，等等，在本书的其余部分将对文法符号的表示使用以下约定。

1）下述符号是终结符号：

① 在字母表里排在前面的小写字母，比如 *a*、*b*、*c*。

② 运算符号，比如 + 、* 等。

③ 标点符号，比如括号、逗号等。

④ 数字0、1、…、9。

⑤ 黑体字符串，比如 **id** 或 **if**。每个这样的字符串表示一个终结符号。

2）下述符号是非终结符号：

① 在字母表中排在前面的大写字母，比如 *A*、*B*、*C*。

② 字母 *S*。它出现时通常表示开始符号。

③ 小写、斜体的名字，比如 *expr* 或 *stmt*。

④ 当讨论程序设计语言的构造时，大写字母可以用于表示代表程序构造的非终结符号。比

如，表达式、项和因子的非终结符号通常分别用 E、T 和 F 表示。

3）在字母表中排在后面的大写字母（比如 X、Y、Z）表示文法符号。也就是说，表示非终结符号或终结符号。

4）在字母表中排在后面的小写字母（主要是 u、v、…、z）表示（可能为空的）终结符号串。

5）小写的希腊字母，比如 α、β、γ，表示（可能为空的）文法符号串。因此，一个普通的产生式可以写作 $A\rightarrow\alpha$，其中 A 是产生式的头，α 是产生式的体。

6）具有相同的头的一组产生式 $A\rightarrow\alpha_1$，$A\rightarrow\alpha_2$，…，$A\rightarrow\alpha_k$（A 产生式）可以写作 $A\rightarrow\alpha_1\mid\alpha_2\mid\cdots\mid\alpha_k$。我们把 α_1，α_2，…，α_k 称作 A 的不同可选体。

7）除非特别说明，第一个产生式的头就是开始符号。

例 4.6 按照这些约定，例子 4.5 的文法可以改为如下更加简单的形式：

$$E\rightarrow E + T \mid E - T \mid T$$
$$T\rightarrow T * F \mid T / F \mid F$$
$$F\rightarrow (E) \mid \mathbf{id}$$

上面的符号表示约定告诉我们 E、T 和 F 是非终结符号，其中 E 是开始符号。其余的符号是终结符号。 □

4.2.3 推导

将产生式看作重写规则，就可以从推导的角度精确地描述构造语法分析树的方法。从开始符号出发，每个重写步骤把一个非终结符号替换为它的某个产生式的体。这个推导思想对应于自顶向下构造语法分析树的过程，但是推导概念所给出的精确性在讨论自底向上的语法分析过程时尤其有用。正如我们将看到的，自底向上语法分析和一种被称为“最右”推导的推导类型相关。在这种推导过程中，每一步重写的都是最右边的非终结符号。

比如，考虑下列只有一个非终结符号 E 的文法。它在文法（4.3）中增加了一个产生式 $E\rightarrow -E$：

$$E\rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id} \tag{4.7}$$

产生式 $E\rightarrow -E$ 表明，如果 E 表示一个表达式，那么 $-E$ 必然也表示一个表达式。将一个 E 替换为 $-E$ 的过程写作

$$E\Rightarrow -E$$

上式读作“E 推导出 $-E$”。产生式 $E\rightarrow(E)$ 可以将任何文法符号串中出现的 E 的任何实例替换为 (E)。比如，$E * E\Rightarrow (E) * E$ 或 $E * E\Rightarrow E * (E)$。我们可以按照任意顺序对单个 E 不断地应用各个产生式，得到一个替换的序列。比如：

$$E\Rightarrow -E\Rightarrow -(E)\Rightarrow -(\mathbf{id})$$

我们将这个替换序列称为从 E 到 $-(\mathbf{id})$ 的推导。这个推导证明了串 $-(\mathbf{id})$ 是表达式的一个实例。

要给出推导的一般性定义，考虑一个文法符号序列中间的非终结符号 A，比如 $\alpha A\beta$，其中 α 和 β 是任意的文法符号串。假设 $A\rightarrow\gamma$ 是一个产生式。那么我们写作 $\alpha A\beta\Rightarrow\alpha\gamma\beta$。符号 $\Rightarrow$ 表示“通过一步推导出”。当一个推导序列 $\alpha_1\Rightarrow\alpha_2\Rightarrow\cdots\Rightarrow\alpha_n$ 将 α_1 替换为 α_n，我们说 α_1 推导出 α_n。我们经常说“经过零步或多步推导出”，我们可以使用符号 $\overset{*}{\Rightarrow}$ 来表示这种关系。因此，

1）对于任何串 α，$\alpha\overset{*}{\Rightarrow}\alpha$，并且

2）如果 $\alpha\overset{*}{\Rightarrow}\beta$ 且 $\beta\Rightarrow\gamma$，那么 $\alpha\overset{*}{\Rightarrow}\gamma$。

类似地，$\overset{*}{\Rightarrow}$ 表示“经过一步或多步推导出”。

如果 $S \dot{\Rightarrow} \alpha$，其中 S 是文法 G 的开始符号，我们说 α 是 G 的一个句型(sentential form)。请注意，一个句型可能既包含终结符号又包含非终结符号，也可能是空串。文法 G 的一个句子(sentence)是不包含非终结符号的句型。一个文法生成的语言是它的所有句子的集合。因此，一个终结符号串 w 在 G 生成的语言 $L(G)$ 中，当且仅当 w 是 G 的一个句子(或者说 $S \dot{\Rightarrow} w$)。可以由文法生成的语言被称为上下文无关语言(context-free language)。如果两个文法生成相同语言，这两个文法就被称为是等价的。

串 $-(\mathbf{id}+\mathbf{id})$ 是文法(4.7)的一个句子，因为存在一个推导过程

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id}) \tag{4.8}$$

串 E、$-E$、$-(E)$、…、$-(\mathbf{id}+\mathbf{id})$ 都是这个文法的句型。我们用 $E \dot{\Rightarrow} -(\mathbf{id}+\mathbf{id})$ 来指明 $-(\mathbf{id}+\mathbf{id})$ 可以从 E 推导得到。

在每一个推导步骤上都需要做两个选择。我们要选择替换哪个非终结符号，并且在做出这个决定之后，还必须选择一个以此非终结符号作为头的产生式。比如，下面给出的 $-(\mathbf{id}+\mathbf{id})$ 的另一种推导和推导(4.8)在最后两步有所不同：

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id}) \tag{4.9}$$

在这两个推导中，每个非终结符号都被替换为同一个产生式体，但替换的顺序有所不同。

为了理解语法分析器是如何工作的，我们将考虑在每个推导步骤中按照如下方式选择被替换的非终结符号的两种推导过程：

1) 在最左推导(leftmost derivation)中，总是选择每个句型的最左非终结符号。如果 $\alpha \Rightarrow \beta$ 是一个推导步骤，且被替换的是 α 中的最左非终结符号，我们写作 $\alpha \underset{lm}{\Rightarrow} \beta$。

2) 在最右推导(rightmost derivation)中，总是选择最右边的非终结符号，此时我们写作 $\alpha \underset{rm}{\Rightarrow} \beta$。

推导(4.8)是最左推导，因此它可以写成

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$$

请注意，推导(4.9)是一个最右推导。

根据我们的符号表示惯例，每个最左推导步骤都可以写成 $wA\gamma \underset{lm}{\Rightarrow} w\delta\gamma$，其中 w 只包含终结符号，$A \to \delta$ 是被应用的产生式，而 γ 是一个文法符号串。为了强调 α 经过一个最左推导过程得到 β，我们写作 $\alpha \underset{lm}{\dot{\Rightarrow}} \beta$。如果 $S \underset{lm}{\dot{\Rightarrow}} \alpha$，那么我们说 α 是当前文法的最左句型(left-sentential form)。

对于最右推导也有类似的定义。最右推导有时也称为规范推导(canonical derivation)。

4.2.4 语法分析树和推导

语法分析树是推导的图形表示形式，它过滤掉了推导过程中对非终结符号应用产生式的顺序。语法分析树的每个内部结点表示一个产生式的应用。该内部结点的标号是此产生式头中的非终结符号 A；这个结点的子结点的标号从左到右组成了在推导过程中替换这个 A 的产生式体。

比如，图 4-3 中，$-(\mathbf{id}+\mathbf{id})$ 的语法分析树是根据推导(4.8)得到的，它也可以根据推导(4.9)得到。

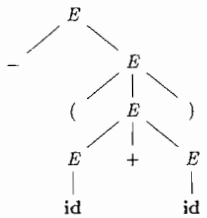

图 4-3 $-(\mathbf{id}+\mathbf{id})$ 的语法分析树

一棵语法分析树的叶子结点的标号既可以是非终结符号，也可以是终结符号。从左到右排列这些符号就可以得到一个句型，它称为这棵树的结果(yield)或边缘(frontier)。

为了了解推导和语法分析树之间的关系，考虑任意的推导 $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$，其中 α_1 是单个非终结符号 A。对于推导中的每个句型 α_i，我们可以构造出一个结果为 α_i 的语法分析树。这个构造过程是对 i 的一次

归纳过程。

基础：$\alpha_1 = A$ 的语法分析树就是标号为 A 的单个结点。

归纳步骤：假设我们已经构造出了一棵结果为 $\alpha_{i-1} = X_1 X_2 \cdots X_k$ 的语法分析树（请注意，按照我们的符号表示约定，每个文法符号 X_i 可以是非终结符号，也可以是终结符号）。假设 α_i 是将 α_{i-1} 中的某个非终结符号 X_j 替换为 $\beta = Y_1 Y_2 \cdots Y_m$ 而得到的句型。也就是说，在这个推导的第 i 步中，对 α_{i-1} 应用规则 $X_j \rightarrow \beta$，推导出 $\alpha_i = X_1 X_2 \cdots X_{j-1} \beta X_{j+1} \cdots X_k$。

为了模拟这一推导步骤，我们在当前的语法分析树中找出左起第 j 个非 ϵ 叶子结点。这个结点的标号为 X_j。向这个叶子结点添加 m 个子结点，从左边开始分别将这些子结点标号为 Y_1、Y_2、…、Y_m。作为一种特殊情况，如果 $m = 0$，那么 $\beta = \epsilon$，我们给第 j 个叶子结点加上一个标号为 ϵ 的子结点。

例 4.10 根据推导(4.8)构造得到的语法分析树的序列显示在图 4-4 中。推导的第一步是 $E \Rightarrow -E$。为了模拟这一步，我们将标号分别为 - 和 E 的两个子结点加到第一棵树的根结点 E 上，得到第二棵语法分析树。

这个推导的第二步是 $-E \Rightarrow -(E)$。相应地，将标号分别为(、E、)的三个子结点加到第二棵树中标号为 E 的叶子结点上，得到结果为 $-(E)$ 的第三棵树。按照这个方法继续下去，我们就得到了完整的语法分析树，即第六棵树。 □

因为语法分析树忽略了替换句型中符号的不同顺序，所以在推导和语法分析树之间具有多对一的关系。比如，推导(4.8)和(4.9)都和图 4-4 中的最后一棵语法分析树关联。

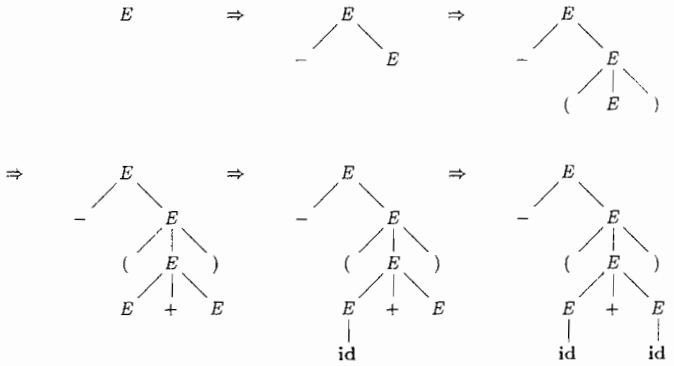

图 4-4 推导(4.8)的语法分析树序列

因为在语法分析树和最左推导/最右推导之间存在一对一的关系，所以在接下来的内容中，我们将频繁地通过构造最左推导或最右推导来进行语法分析。最左或最右推导都以一种特定的顺序来替换句型中的符号，因此它们也过滤掉了顺序上的不同。不难说明，每一棵语法分析树都和唯一的最左推导及唯一的最右推导相关联。

4.2.5 二义性

根据 2.2.4 节的介绍可知，如果一个文法可以为某个句子生成多棵语法分析树，那么它就是二义性的(ambiguous)。换句话说，二义性文法就是对同一个句子有多个最左推导或多个最右推导的文法。

例 4.11 算术表达式文法(4.3)允许句子 **id + id * id** 具有两个最左推导：

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow \mathbf{id} + E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow E + E * E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$

相应的语法分析树如图 4-5 所示。

请注意，图 4-5a 中的语法分析树反映了通常的 + 和 * 之间的优先级关系，而图 4-5b 中的语法分析树则没有反映出这一点。也就是说，按照惯例，应该将运算符 * 当作优先级高于 + 的运算符来处理，相应地，我们通常将 $a + b * c$ 这样的表达式按照 $a + (b * c)$，而不是 $(a + b) * c$ 的方式进行求值。 □

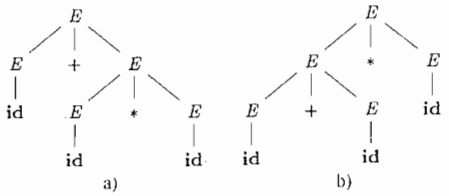

图 4-5 **id + id * id** 的两棵语法树

大部分语法分析器都期望文法是无二义性的，否则，我们就不能为一个句子唯一地选定语法分析树。在某些情况下，使用经过精心选择的二义性文法也可以带来方便。但同时需要使用消二义性规则(disambiguating rule)来“抛弃”不想要的语法分析树，只为每个句子留下一棵语法分析树。

4.2.6 验证文法生成的语言

推断出一个给定的产生式集合生成了某种特定的语言是很有用的，尽管编译器的设计者很少会对整个程序设计语言文法做这样的事情。当研究一个棘手的构造时，我们可以写出该构造的一个简洁、抽象的文法，并研究该文法生成的语言。我们将为下面的条件语句构造出这样的文法。

证明文法 G 生成语言 L 的过程可以分成两个部分：证明 G 生成的每个串都在 L 中，并且反向证明 L 中的每个串都确实能由 G 生成。

例 4.12 考虑下面的文法：

$$S \rightarrow (S) S \mid \epsilon \tag{4.13}$$

初看可能不是很明显，但这个简单的文法确实生成了所有具有对称括号对的串，并且只生成这样的串。为了说明原因，我们将首先说明从 S 推导得到的每个句子都是括号对称的，然后说明每个括号对称的串都可以从 S 推导得到。为了证明从 S 推导出的每个句子都是括号对称的，我们对推导步数 n 进行归纳。

基础： 基础是 $n = 1$。唯一可以从 S 经过一步推导得到的终结符号串是空串，它当然是括号对称的。

归纳步骤： 现在假设所有步数少于 n 的推导都得到括号对称的句子，并考虑一个恰巧有 n 步的最左推导。这样的推导必然具有如下形式：

$$S \underset{lm}{\Rightarrow} (S)S \underset{lm}{\overset{*}{\Rightarrow}} (x)S \underset{lm}{\overset{*}{\Rightarrow}} (x)y$$

从 S 到 x 和 y 的推导过程都少于 n 步，因此根据归纳假设，x 和 y 都是括号对称的。因此，串 $(x) y$ 必然是括号对称的。也就是说，它具有相同数量的左括号和右括号，并且它的每个前缀中的左括号不少于右括号。

现在已经证明了可以从 S 推导出的任何串都是括号对称的，接下来我们必须证明每个括号对称的串都可以从 S 推导得到。为了证明这一点，我们对串的长度进行归纳。

基础： 如果串的长度是 0，它必然是 ϵ。这个串是括号对称的，且可以从 S 推导得到。

归纳步骤： 首先请注意，每个括号对称的串的长度是偶数。假设每个长度小于 $2n$ 的括号对称的串都能够从 S 推导得到，并考虑一个长度为 $2n(n \geq 1)$ 的括号对称的串 w。w 一定以左括号开

头。令(x)是w的最短的、左括号个数和右括号个数相同的非空前缀，那么w可以写成$w=(x)y$的形式，其中x和y都是括号对称的。因为x和y的长度都小于$2n$，根据归纳假设，它们可以从S推导得到。因此，我们可以找到一个如下形式的推导：

$$S \Rightarrow (S)S \stackrel{*}{\Rightarrow} (x)S \stackrel{*}{\Rightarrow} (x)y$$

它证明$w=(x)y$也可以从S推导得到。 □

4.2.7 上下文无关文法和正则表达式

在结束关于文法及其性质的讨论之前，我们要说明文法是比正则表达式表达能力更强的表示方法。每个可以使用正则表达式描述的构造都可以使用文法来描述，但是反之不成立。换句话说，每个正则语言都是一个上下文无关语言，但是反之不成立。

比如，正则表达式$(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$和文法

$$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$$
$$A_1 \rightarrow bA_2$$
$$A_2 \rightarrow bA_3$$
$$A_3 \rightarrow \epsilon$$

描述了同一个语言，即以abb结尾的由a和b组成的串的集合。

我们可以机械地构造出和一个不确定有穷自动机(NFA)识别同样语言的文法。上面的文法是使用下面的构造方法，根据图 3-24 中的 NFA 构造得到的。

1) 对于 NFA 的每个状态i，创建一个非终结符号A_i。

2) 如果状态i有一个在输入a上到达状态j的转换，则加入产生式$A_i \rightarrow aA_j$。如果状态i在输入ϵ上到达状态j，则加入产生式$A_i \rightarrow A_j$。

3) 如果i是一个接受状态，则加入产生式$A_i \rightarrow \epsilon$。

4) 如果i是自动机的开始状态，令A_i为所得文法的开始符号。

另一方面，语言$L=\{a^nb^n \mid n \geqslant 1\}$(即由同样数量的$a$和$b$组成的串的集合)是一个可以用文法描述但不能用正则表达式描述的语言的原型例子。下面用反证法来说明这一点。假设L是用某个正则表达式定义的语言。我们可以构造一个具有有穷多个状态(比如说k个状态)的 DFA D来接受L。因为D只有k个状态，对于一个以多于k个a开头的输入，D一定会进入某个状态两次，假设这个状态是s_i，如图 4-6 所示。假设从s_i返回到其自身的路径的标号序列是a^{j-i}。因为a^ib^i在这个语言中，因此必然存在一条标号为b^i从s_i到某个接受状态f的路径。但是，一定还存在一条从开始状态s_0出发，经过s_i最后到达f的路径，它的标号序列为a^jb^i，如图 4-6 所示。因此，D也接受a^jb^i，但a^jb^i这个串不在语言L中，这和L是D所接受的语言这个假设矛盾。

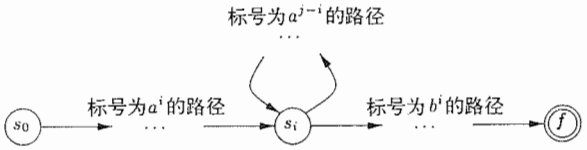

图 4-6 接受a^ib^i和a^jb^i的 DFA D

我们通俗地说“有穷自动机不能计数”，这意味着有穷自动机不能接受像$\{a^nb^n \mid n \geqslant 1\}$这样的语言，因为它不能记录下在它看到第一个$b$之前读入的$a$的个数。类似地，“一个文法可以对两个个体进行计数，但是无法对三个个体计数”，我们在 4.3.5 节中考虑非上下文无关的语言构造时将介绍这一点。

4.2.8 4.2 节的练习

练习 4.2.1：考虑上下文无关文法：

$$S \rightarrow S\ S\ +\ |\ S\ S\ *\ |\ a$$

以及串 $aa+a*$。

1）给出这个串的一个最左推导。

2）给出这个串的一个最右推导。

3）给出这个串的一棵语法分析树。

! 4）这个文法是否为二义性的？证明你的回答。

! 5）描述这个文法生成的语言。

练习 4.2.2：对下列的每一对文法和串重复练习 4.2.1。

1）$S \rightarrow 0\ S\ 1\ |\ 0\ 1$ 和串 000111。

2）$S \rightarrow +\ S\ S\ |\ *\ S\ S\ |\ a$ 和串 $+*aaa$。

! 3）$S \rightarrow S\ (\ S\ )\ S\ |\ \epsilon$ 和串(()())。

! 4）$S \rightarrow S\ +\ S\ |\ S\ S\ |\ (\ S\ )\ |\ S*\ |\ a$ 和串 $(a+a)*a$。

! 5）$S \rightarrow (\ L\ )\ |\ a$ 以及 $L \rightarrow L,\ S\ |\ S$ 和串 $((a,\ a),\ a,\ (a))$。

!! 6）$S \rightarrow a\ S\ b\ S\ |\ b\ S\ a\ S\ |\ \epsilon$ 和串 $aabbab$。

! 7）下面的布尔表达式对应的文法：

$$bexpr \rightarrow bexpr\ \textbf{or}\ bterm\ |\ bterm$$
$$bterm \rightarrow bterm\ \textbf{and}\ bfactor\ |\ bfactor$$
$$bfactor \rightarrow \textbf{not}\ bfactor\ |\ (\ bexpr\ )\ |\ \textbf{true}\ |\ \textbf{false}$$

练习 4.2.3：为下面的语言设计文法：

1）所有由 0 和 1 组成的并且每个 0 之后都至少跟着一个 1 的串的集合。

! 2）所有由 0 和 1 组成的回文(palindrome)的集合，也就是从前面和从后面读结果都相同的串的集合。

! 3）所有由 0 和 1 组成的具有相同多个 0 和 1 的串的集合。

!! 4）所有由 0 和 1 组成的并且 0 的个数和 1 的个数不同的串的集合。

! 5）所有由 0 和 1 组成的且其中不包含子串 011 的串的集合。

!! 6）所有由 0 和 1 组成的形如 xy 的串的集合，其中 $x \neq y$ 且 x 和 y 等长。

! **练习 4.2.4**：有一个常用的扩展的文法表示方法。在这个表示方法中，产生式体中的方括号和花括号是元符号(如→或 |)，且具有如下含义：

1）一个或多个文法符号两边的方括号表示这些构造是可选的。因此，产生式 $A \rightarrow X[\,Y\,]Z$ 和两个产生式 $A \rightarrow XYZ$ 及 $A \rightarrow XZ$ 具有相同的效果。

2）一个或多个文法符号两边的花括号表示这些符号可以重复任意多次(包括零次)。因此，$A \rightarrow X\{YZ\}$ 和如下的无穷产生式序列具有相同的效果：$A \rightarrow X$，$A \rightarrow XYZ$，$A \rightarrow XYZYZ$，…等等。证明这两个扩展并没有增加文法的功能。也就是说，由带有这些扩展表示的文法生成的任何语言都可以由一个不带扩展表示的文法生成。

练习 4.2.5：使用练习 4.2.4 中描述的括号表示法来简化如下的关于语句块和条件语句的文法。

$$\begin{aligned} stmt \rightarrow\ & \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\ |\ & \textbf{if}\ stmt\ \textbf{then}\ stmt \\ |\ & \textbf{begin}\ stmtList\ \textbf{end} \\ stmtList \rightarrow\ & stmt;\ stmtList\ |\ stmt \end{aligned}$$

! **练习 4.2.6**：扩展练习 4.2.4 的思想，使得产生式体中可以出现文法符号的任意正则表达式。证明这个扩展并没有使得文法可以定义任何新的语言。

! 练习 4.2.7：如果不存在形如 $S \Rightarrow wXy \Rightarrow wxy$ 的推导，那么文法符号 X（终结符号或非终结符号）就被称为无用的（useless）。也就是说，X 不可能出现在任何句子的推导过程中。

1）给出一个算法，从一个文法中消除所有包含无用符号的产生式。

2）将你的算法应用于以下文法：

$S \rightarrow 0 \mid A$

$A \rightarrow AB$

$B \rightarrow 1$

练习 4.2.8：图 4-7 中的文法可生成单个数值标识符的声明，这些声明包含四种不同的、相互独立的数字性质。

stmt	→	**declare id** *optionList*
optionList	→	*optionList option* | ϵ
option	→	*mode* | *scale* | *precision* | *base*
mode	→	**real** | **complex**
scale	→	**fixed** | **floating**
precision	→	**single** | **double**
base	→	**binary** | **decimal**

图 4-7 多属性声明的文法

1）扩展图 4-7 中的文法，使得它可以允许 n 种选项 A_i，其中 n 是一个固定的数，$i=1, 2, \cdots, n$。选项 A_i 的取值可以是 a_i 或 b_i。你的文法只能使用 $O(n)$ 个文法符号，并且产生式的总长度也必须是 $O(n)$ 的。

! 2）图 4-7 中的文法和它在 1 中的扩展支持互相矛盾或冗余的声明，比如：

```
declare foo real fixed real floating
```

我们可以要求这个语言的语法禁止这种声明。也就是说，由这个文法生成的每个声明中，n 种选项中的每一项都有且只有一个取值。如果我们这样做，那么对于任意给定的 n 值，合法声明的个数是有穷的。因此和任何有穷语言一样，合法声明组成的语言有一个文法（同时也有一个正则表达式）。最显而易见的文法是这样的：文法的开始符号对每个合法声明都有一个产生式，这样共有 $n!$ 个产生式。该文法的产生式的总长度是 $O(n \times n!)$。你必须做得更好：给出一个产生式总长度为 $O(n2^n)$ 的文法。

!! 3）说明对于任何满足 2 中的要求的文法，其产生式的总长度至少是 2^n。

4）我们可以通过程序设计语言的语法来保证声明中的选项无冗余性、无矛盾。对于这个方法的可行性，本题 3 的结论说明了什么问题？

4.3 设计文法

文法能够描述程序设计语言的大部分（但不是全部）语法。比如，在程序中标识符必须先声明后使用，但是这个要求不能通过一个上下文无关文法来描述。因此，一个语法分析器接受的词法单元序列构成了程序设计语言的超集；编译器的后续步骤必须对语法分析器的输出进行分析，以保证源程序遵守那些没有被语法分析器检查的规则。

本节将先讨论如何在词法分析器和语法分析器之间分配工作。然后考虑几个用来使文法更适于语法分析的转换方法。其中的一个技术可以消除文法中的二义性，而其他的技术——消除左递归和提取左公因子——可用于改写文法，使得这些文法适用于自顶向下的语法分析。我们在本节的最后将考虑一些不能使用任何文法描述的程序设计语言构造。

4.3.1 词法分析和语法分析

如我们在 4.2.7 节看到的，任何能够使用正则表达式描述的东西都可以使用文法描述。因此我们自然会问："为什么使用正则表达式来定义一个语言的词法语法？"，理由有多个。

1）将一个语言的语法结构分为词法和非词法两部分可以很方便地将编译器前端模块化，将前端分解为两个大小适中的组件。

2）一个语言的词法规则通常很简单，我们不需要使用像文法这样的功能强大的表示方法来描述这些规则。

3）和文法相比，正则表达式通常提供了更加简洁且易于理解的表示词法单元的方法。

4）根据正则表达式自动构造得到的词法分析器的效率要高于根据任意文法自动构造得到的分析器。

并不存在一个严格的指导方针来规定哪些东西应该放到（和语法规则相对的）词法规则中。正则表达式最适合描述诸如标识符、常量、关键字、空白这样的语言构造的结构。另一方面，文法最适合描述嵌套结构，比如对称的括号对，匹配的 begin-end，相互对应的 if-then-else 等。这些嵌套结构不能使用正则表达式描述。

4.3.2 消除二义性

有时，一个二义性文法可以被改写为无二义性的文法。例如，我们将消除下面的“悬空-else”文法中的二义性：

$$\begin{aligned} stmt &\rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt \\ &\mid \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\ &\mid \textbf{other} \end{aligned} \tag{4.14}$$

这里“**other**”表示任何其他语句。根据这个文法，下面的复合条件语句

$$\textbf{if}\ E_1\ \textbf{then}\ S_1\ \textbf{else if}\ E_2\ \textbf{then}\ S_2\ \textbf{else}\ S_3$$

的语法分析树如图 4-8 所示[⊖]。文法(4.14)是二义性的，因为串

$$\textbf{if}\ E_1\ \textbf{then if}\ E_2\ \textbf{then}\ S_1\ \textbf{else}\ S_2 \tag{4.15}$$

具有图 4-9 所示的两棵语法分析树。

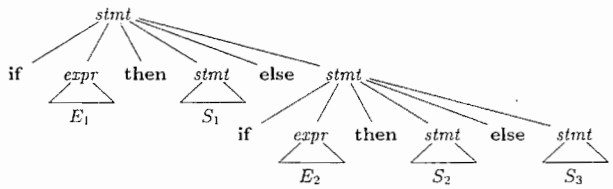

图 4-8 一个条件语句的语法分析树

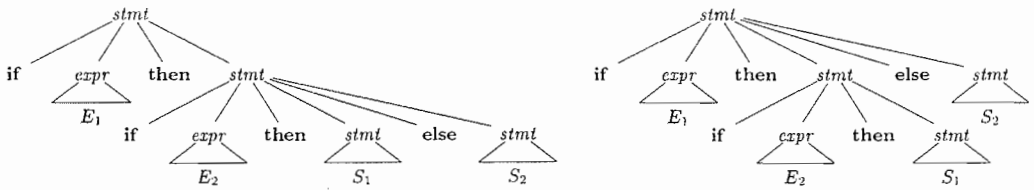

图 4-9 一个二义性句子的两颗语法分析树

在所有包含这种形式的条件语句的程序设计语言中，总是会选择第一棵语法分析树。通用的规则是“每个 **else** 和最近的尚未匹配的 **then** 匹配。”[⊜]从理论上讲，这个消除二义性规则可以用一个文法直接表示，但是在实践中很少用产生式来表示该规则。

例 4.16 我们可以将悬空-else 文法(4.14)改写成如下的无二义性文法。基本思想是在一个 **then** 和一个 **else** 之间出现的语句必须是“已匹配的”。也就是说，中间的语句不能以一个尚未匹

⊖ E 和 S 的下标仅用于区分同一个非终结符号的不同出现，并不表示不同的非终结符号。

⊜ 我们应该注意到，C 语言和它的派生语言也属于这一类语言。虽然 C 系列的语言不使用关键字 **then**，但 **then** 的作用是由 **if** 之后的条件表达式的括号对来承担的。

配的(或者说开放的) **then** 结尾。一个已匹配的语句要么是一个不包含开放语句的 **if-then-else** 语句，要么是一个非条件语句。因此我们可以使用图 4-10 中的文法。这个文法和悬空-else文法(4.14)生成同样的串集合，但是它只允许对串(4.15)进行一种语法分析，也就是将每个 **else** 和前面最近的尚未匹配的 **then** 匹配。 □

stmt	→	*matched_stmt*
	|	*open_stmt*
matched_stmt	→	**if** *expr* **then** *matched_stmt* **else** *matched_stmt*
	|	**other**
open_stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *matched_stmt* **else** *open_stmt*

图 4-10 **if-then-else** 语句的无二义性方法

4.3.3 左递归的消除

如果一个文法中有一个非终结符号 A 使得对某个串 α 存在一个推导 $A \Rightarrow^{+} A\alpha$，那么这个文法就是*左递归的*(left recursive)。自顶向下语法分析方法不能处理左递归的文法，因此需要一个转换方法来消除左递归。在 2.4.5 节中，我们讨论了*立即左递归*，即存在形如 $A \rightarrow A\alpha$ 的产生式的情况。这里我们研究一般性的情形。在 2.4.5 节中，我们说明了如何把左递归的产生式对 $A \rightarrow A\alpha \mid \beta$ 替换为非左递归的产生式：

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

这样的替换不会改变可从 A 推导得到的串的集合。这个规则本身已经足以用来处理很多文法。

例 4.17 这里重复一下非左递归的表达式文法(4.2)：

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +\ T\ E' \mid \epsilon \\ T &\rightarrow F\ T' \\ T' &\rightarrow +\ F\ T' \mid \epsilon \\ F &\rightarrow (\ E\ ) \mid \mathbf{id} \end{aligned}$$

它是通过消除表达式文法(4.1)中的立即左递归而得到的。左递归的产生式对 $E \rightarrow E + T \mid T$ 被替换为 $E \rightarrow T\ E'$ 和 $E' \rightarrow +\ T\ E' \mid \epsilon$。类似地，$T$ 和 T' 的新产生式也是通过消除立即左递归而得到的。 □

立即左递归可以使用下面的技术消除，该技术可以处理任意数量的 A 产生式。首先将 A 的全部产生式分组如下：

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

其中 β_i 都不以 A 开头。然后，将这些 A 产生式替换为：

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

非终结符号 A 生成的串和替换之前生成的串一样，但不再是左递归的。这个过程消除了所有和 A 和 A' 的产生式相关的左递归(前提是 α_i 都不是 ϵ)，但是它没有消除那些因为两步或多步推导而产生的左递归。比如，考虑文法

$$S \rightarrow A\ a \mid b$$
$$A \rightarrow A\ c \mid S\ d \mid \epsilon \qquad (4.18)$$

因为 $S \Rightarrow Aa \Rightarrow Sda$，所以非终结符号 S 是左递归的，但它不是立即左递归的。

下面的算法 4.19 系统地消除了文法中的左递归。如果文法中不存在环（即形如 $A \overset{+}{\Rightarrow} A$ 的推导）或 ϵ 产生式（即形如 $A \to \epsilon$ 的产生式），就保证能够消除左递归。环和 ϵ 产生式都可以从文法中系统地消除（见练习 4.4.6 和练习 4.4.7）。

算法 4.19 消除左递归。

输入： 没有环或 ϵ 产生式的文法 G。

输出： 一个等价的无左递归文法。

方法： 对 G 应用图 4-11 中的算法。请注意，得到的非左递归文法可能具有 ϵ 产生式。 □

图 4-11 中的过程的工作原理如下。在 $i=1$ 的第一次迭代中，第 2～7 行的外层循环消除了 A_1 产生式之间的所有立即左递归。因此，余下的所有形如 $A_1 \to A_l\alpha$ 的产生式都一定满足 $l>1$。在外层循环的第 $i-1$ 次迭代之后，所有的非终结符号 A_k（$k<i$）都被“清洗”过了。也就是说，任何产生式 $A_k \to A_l\alpha$ 都必然满足 $l>k$。结果，在第 i 次迭代中，第 3～5 行的内层循环不断提高所有形如 $A_i \to A_m\alpha$ 的产生式中 m 的下界，直到 $m \geqslant i$ 成立为止。然后，第 6 行消除了 A_i 产生式中的立即左递归，保证 $m>i$ 成立。

```
1)  按照某个顺序将非终结符号排序为 A1, A2, ..., An.
2)  for ( 从 1 到 n的每个 i ) {
3)      for ( 从1 到 i − 1的每个 j ) {
4)          将每个形如 Ai → Ajγ 的产生式替换为产生式组 Ai → δ1γ | δ2γ | ··· | δkγ,
            其中 Aj → δ1 | δ2 | ··· | δk 是所有的Aj产生式
5)      }
6)      消除 Ai 产生式之间的立即左递归
7)  }
```

图 4-11 消除文法中的左递归的算法

例 4.20 我们将算法 4.19 应用于文法(4.18)。从技术上讲，因为该算法有 ϵ 产生式，所以这个算法不一定能得到正确结果。但在这个例子中，最终会证明产生式 $A \to \epsilon$ 是无害的。

我们将非终结符号排序为 S，A。在 S 产生式之间没有立即左递归，因此在 $i=1$ 的外层循环中不进行任何处理。当 $i=2$ 时，我们替换 $A \to Sd$ 中的 S，得到如下的 A 产生式。

$$A \to A\ c \mid A\ a\ d \mid b\ d \mid \epsilon$$

消除这些 A 产生式之间的立即左递归，得到如下的文法：

$$S \to A\ a \mid b$$

$$A \to b\ d\ A' \mid A'$$

$$A' \to c\ A' \mid a\ d\ A' \mid \epsilon$$

□

4.3.4 提取左公因子

提取左公因子是一种文法转换方法，它可以产生适用于预测分析技术或自顶向下分析技术的文法。当不清楚应该在两个 A 产生式中如何选择时，我们可以通过改写产生式来推后这个决定，等我们读入了足够多的输入，获得足够信息后再做出正确选择。

比如，如果我们有两个产生式

$$stmt \to \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt$$

$$\mid\ \textbf{if}\ expr\ \textbf{then}\ stmt$$

在看到输入 **if** 的时候，我们不能立刻指出应该选择哪个产生式来展开 *stmt*。一般来说，如果 $A\rightarrow\alpha\beta_1 \mid \alpha\beta_2$ 是两个 A 产生式，并且输入的开头是从 α 推导得到的一个非空串，那么我们就不知道应该将 A 展开为 $\alpha\beta_1$ 还是 $\alpha\beta_2$。然而，我们可以将 A 展开为 $\alpha A'$，从而将做出决定的时间往后延。在读入了从 α 推导得到的输入前缀之后，我们再决定将 A' 展开为 β_1 或 β_2。也就是说，经过提取左公因子，原来的产生式变成了

$$A\rightarrow\alpha A'$$
$$A'\rightarrow\beta_1 \mid \beta_2$$

算法 4.21 对一个文法提取左公因子。

输入：文法 G。

输出：一个等价的提取了左公因子的文法。

方法：对于每个非终结符号 A，找出它的两个或多个选项之间的最长公共前缀 α。如果 $\alpha\neq\epsilon$，即存在一个非平凡的公共前缀，那么将所有 A 产生式 $A\rightarrow\alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$，替换为

$$A\rightarrow\alpha A' \mid \gamma$$
$$A'\rightarrow\beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

其中，γ 表示所有不以 α 开头的产生式体；A' 是一个新的非终结符号。不断应用这个转换，直到每个非终结符号的任意两个产生式体都没有公共前缀为止。 □

例 4.22 下面的文法抽象表达了“悬空-else”问题：

$$S\rightarrow i\ E\ t\ S \mid i\ E\ t\ S\ e\ S \mid a \tag{4.23}$$
$$E\rightarrow b$$

这里 i、t 和 e 代表 **if**、**then** 和 **else**；E 和 S 表示“条件表达式”和“语句”。提取左公因子后，这个文法变为：

$$S\rightarrow i\ E\ t\ S\ S' \mid a$$
$$S'\rightarrow e\ S \mid \epsilon \tag{4.24}$$
$$E\rightarrow b$$

这样，我们可以在输入为 i 时将 S 展开为 $iEtSS'$，并在处理 $iEtS$ 之后才决定将 S' 展开为 eS 还是 ϵ。当然，上面的两个文法都是二义性的，当输入为 e 时不能够确定应该选择 S' 的哪个产生式。例子 4.33 将讨论一个可以摆脱这个困境的方法。 □

4.3.5 非上下文无关语言的构造

在常见的程序设计语言中，可以找到少量不能仅用文法描述的语法构造。这里，我们考虑其中的两种构造，并使用简单的抽象语言来说明其困难之处。

例 4.25 这个例子中的语言抽象地表示了检查标识符在程序中先声明后使用的问题。这个语言由形如 wcw 的串组成，其中第一个 w 表示某个标识符 w 的声明，c 表示中间的程序片段，第二个 w 表示对这个标识符的使用。

这个抽象语言是 $L_1=\{wcw \mid w\text{ 在 }(a\mid b)^*\text{ 中}\}$。$L_1$ 包含了所有符合以下要求的字，字中包含两个相同的由 a，b 所组成串，且中间以 c 隔开，比如 $aabcaab$。这个 L_1 不是上下文无关的，虽然证明这一点已经超出了本书的范围。L_1 的非上下文无关性表明了像 C 或 Java 这样的语言不是上下文无关的，因为这些语言都要求标识符要先声明后使用，并且支持任意长度的标识符。

出于这个原因，C 或者 Java 的文法不区分由不同字符串组成的标识符。所有的标识符在文法中都被表示为像 **id** 这样的词法单元。在这些语言的编译器中，标识符是否先声明后使用是在语义分析阶段检查的。 □

例 4.26 这个例子中的非上下文无关语言抽象地表示了参数个数检查的问题。它检查一个函数声明中的形式参数个数是否等于该函数的某次使用中的实在参数个数。这个语言由形如 $a^nb^mc^nd^m$ 的串组成(记住,a^n 表示 n 个 a)。这里,a^n 和 b^m 可以表示两个分别有 n 和 m 个参数的函数声明的形式参数列表;而 c^n 和 d^m 分别表示对这两个函数的调用中的实在参数列表。

这个抽象语言是 $L_2 = \{a^nb^mc^nd^m \mid n \geq 1 \text{ 且 } m \geq 1\}$。也就是说,$L_2$ 包含的串都在正则表达式 **a** * **b** * **c** * **d** * 所生成的语言中,并且 a 和 c 的个数相同,b 和 d 的个数相同。这个语言不是上下文无关的。

同样,函数声明和使用的常用语法本身并不考虑参数的个数。比如,一个类 C 语言中的函数调用可能被描述为

$$\begin{aligned} stmt &\rightarrow \textbf{id}\ (expr\_list) \\ expr\_list &\rightarrow expr\_list,\ expr \\ &\mid expr \end{aligned}$$

其中 *expr* 另有适当的产生式。检查一次调用中的参数个数是否正确通常是在语义分析阶段完成的。 □

4.3.6 4.3 节的练习

练习 4.3.1:下面是一个只包含符号 a 和 b 的正则表达式的文法。它使用 + 替代表示并运算的字符 |,以避免和文法中作为元符号使用的竖线相混淆:

$$\begin{aligned} &rexpr \rightarrow rexpr\ +\ rterm \mid rterm \\ &rterm \rightarrow rterm\ rfactor \mid rfactor \\ &rfactor \rightarrow rfactor\ * \mid rprimary \\ &rprimary \rightarrow \textbf{a} \mid \textbf{b} \end{aligned}$$

1) 对这个文法提取左公因子。

2) 提取左公因子的变换能使这个文法适用于自顶向下的语法分析技术吗?

3) 提取左公因子之后,从原文法中消除左递归。

4) 得到的文法适用于自顶向下的语法分析吗?

练习 4.3.2:对下面的文法重复练习 4.3.1:

1) 练习 4.2.1 的文法。

2) 练习 4.2.2(1)的文法。

3) 练习 4.2.2(3)的文法。

4) 练习 4.2.2(5)的文法。

5) 练习 4.2.2(7)的文法。

! **练习 4.3.3**:下面文法的目的是消除 4.3.2 节中讨论的“悬空 - else 二义性”:

$$\begin{aligned} stmt &\rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt \\ &\mid matchedStmt \\ matchedStmt &\rightarrow \textbf{if}\ expr\ \textbf{then}\ matchedStmt\ \textbf{else}\ stmt \\ &\mid \textbf{other} \end{aligned}$$

说明这个文法仍然是二义性的。

4.4 自顶向下的语法分析

自顶向下语法分析可以被看作是为输入串构造语法分析树的问题,它从语法分析树的根结

点开始，按照先根次序(如 2.3.4 节中所讨论的，深度优先地)创建这棵语法分析树的各个结点。自顶向下语法分析也可以被看作寻找输入串的最左推导的过程。

例 4.27 图 4-12 中对应于输入 **id + id * id** 的语法分析树序列是一个根据文法(4.2)进行的最左推导序列。这里重复一下这个文法：

$$
\begin{aligned}
&E \rightarrow T\ E' \\
&E' \rightarrow +\ T\ E' \mid \epsilon \\
&T \rightarrow F\ T' \\
&T' \rightarrow *\ F\ T' \mid \epsilon \\
&F \rightarrow (\ E\ ) \mid \mathbf{id}
\end{aligned}
\tag{4.28}
$$

该语法分析树序列对应于这个输入的一个最左推导。 □

在一个自顶向下语法分析的每一步中，关键问题是确定对一个非终结符号(比如 A)应用哪个产生式。一旦选择了某个 A 产生式，语法分析过程的其余部分负责将相应产生式体中的终结符号和输入相匹配。

本节首先给出被称为*递归下降语法分析*的自顶向下语法分析的通用形式，这种方法可能需要进行回溯，以找到要应用的正确 A 产生式。2.4.2 节介绍的预测分析技术是递归下降分析技术的一个特例，它不需要进行回溯。预测分析技术通过在输入中向前看固定多个符号来选择正确的 A 产生式。通常情况下我们只需要向前看一个符号(即只看下一个输入符号)。

比如，考虑图 4-12 中的自顶向下语法分析过程，它构造出了一棵语法分析树，其中有两个标号为 E' 的结点。在(按照前序遍历次序的)第一个 E' 结点上选择的产生式是 $E' \rightarrow +\ T\ E'$；在第二个 E' 结点上选择的产生式是 $E \rightarrow \epsilon$。预测分析器通过查看下一个输入符号就可以在两个 E' 产生式中选择正确的产生式。

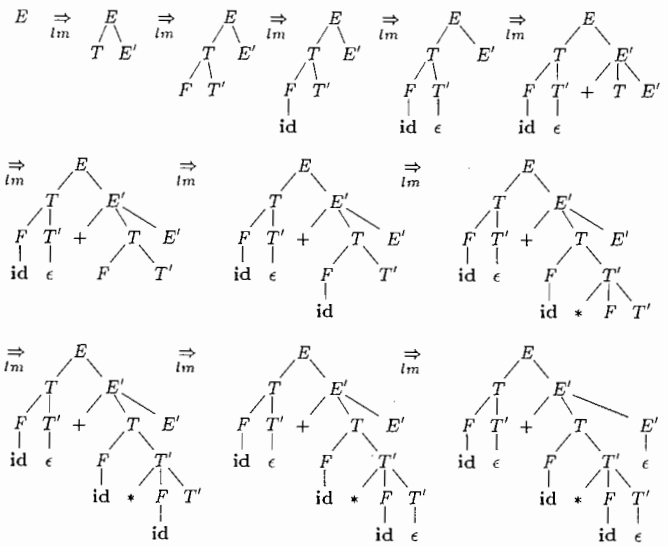

图 4-12 **id + id * id** 的自顶向下分析

对于有些文法，我们可以构造出向前看 k 个输入符号的预测分析器，这一类文法有时也称为 LL(k)文法类。我们在 4.4.3 节中将讨论 LL(1)文法类，但是在介绍预备知识的 4.4.2 节中将介绍一些计算 FIRST 和 FOLLOW 集合的方法。根据一个文法的 FIRST 和 FOLLOW 集合，我们将构

造出“预测分析表”，它说明了如何在自顶向下语法分析过程中选择产生式。这些集合也可以用于自底向上语法分析。

在4.4.4节中，我们给出了一个非递归的语法分析算法，它显式地维护了一个栈，而不是通过递归调用隐式地维护一个栈。最后，我们将在4.4.5节中讨论自顶向下语法分析过程中的错误恢复问题。

4.4.1 递归下降的语法分析

一个递归下降语法分析程序由一组过程组成，每个非终结符号有一个对应的过程。程序的执行从开始符号对应的过程开始，如果这个过程的过程体扫描了整个输入串，它就停止执行并宣布语法分析成功完成。图4-13显示了对应于某个非终结符号的典型过程的伪代码。请注意，这个伪代码是不确定的，因为它没有描述如何在开始时刻选择 A 产生式。

通用的递归下降分析技术可能需要回溯。也就是说，它可能需要重复扫描输入。然而，在对程序设计语言的构造进行语法分析时很少需要回溯，因此需要回溯的语法分析器并不常见。即使在自然语言语法分析这样的场合，回溯也不是很高效，因此人们更加倾向于基于表格的方法，比如练习4.4.9中的动态程序规划算法或者Earley方法(参见参考文献)。

要支持回溯，就需要修改图4-13的代码。首先，因为我们不能在第1行选定唯一的 A 产生式，我们必须按照某个顺序逐个尝试这些产生式。那么，第7行上的失败并不意味着最终失败，而仅仅是建议我们返回到第1行并尝试另一个 A 产生式。只有当再也没有 A 产生式可尝试时，我们才会宣称找到了一个输入错误。为了尝试另一个 A 产生式，我们需要把输入指针重新设置到我们第一次到达第1行时的位置。因此，需要一个局部变量来保存这个输入指针，以供将来回溯时使用。

```
    void A() {
1)      选择一个 A 产生式，A → X1X2⋯Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi 是一个非终结符号 )
4)              调用过程 Xi();
5)          else if ( Xi 等于当前的输入符号 a )
6)              读入下一个输入符号;
7)          else /* 发生了一个错误 */;
        }
    }
```

图4-13 在自顶向下语法分析器中一个非终结符号对应的典型过程

例4.29 考虑文法

$$S \rightarrow c\ A\ d$$
$$A \rightarrow a\ b \mid a$$

在自顶向下地构造输入串 $w = cad$ 的语法分析树时，初始的语法分析树只包含一个标号为 S 的结点，输入指针指向 c，即 w 的第一个符号。S 只有一个产生式，因此我们用它来展开 S，得到图4-14a中的树。最左边的叶子结点的标号为 c，它和输入 w 的第一个符号匹配，因此我们将输入指针推进到 a，即 w 的第二个符号，并考虑下一个标号为 A 的叶子结点。

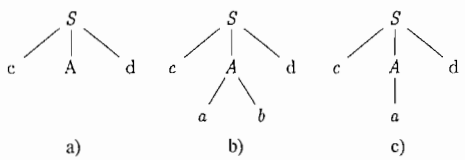

图4-14 一个自顶向下语法分析过程的步骤

现在我们使用第一个 A 产生式 $A \rightarrow ab$ 来展开 A，得到图4-14b所示的树。第二个输入符号 a 得到匹配，因此我们将输入指针推进到 d，即第三个输入符号，并将 d 和下一个叶子结点(标号为 b)比较。因为 b 和 d 不匹配，我们报告失败，并回到 A，查看是否还有尚未尝试过、但有可能匹配的其他 A 产生式。

在回到 A 时，我们必须把输入指针重新设置到位置2，即我们第一次尝试展开 A 时该指针指向的位置。这意味着 A 的过程必须将输入指针存放在一个局部变量中。

A 的第二个选项产生了图4-11c所示的树。叶子结点 a 和 w 的第二个符号匹配，叶子结点 d

和第三个符号相匹配。因为我们已经产生了一棵 w 的语法分析树，所以我们停止分析并宣称已成功完成了语法分析。 □

一个左递归的文法会使它的递归下降语法分析器进入一个无限循环。即使是带回溯的语法分析器也是如此。也就是说，当我们试图展开一个非终结符号 A 的时候，我们可能会没有读入任何输入符号就再次试图展开 A。

4.4.2 FIRST 和 FOLLOW

自顶向下和自底向上语法分析器的构造可以使用和文法 G 相关的两个函数 FIRST 和 FOLLOW 来实现。在自顶向下语法分析过程中，FIRST 和 FOLLOW 使得我们可以根据下一个输入符号来选择应用哪个产生式。在恐慌模式的错误恢复中，由 FOLLOW 产生的词法单元集合可以作为同步词法单元。

FIRST(α)被定义为可从 α 推导得到的串的首符号的集合，其中 α 是任意的文法符号串。如果 $\alpha \stackrel{*}{\Rightarrow} \epsilon$，那么 ϵ 也在 FIRST(α)中。比如在图 4-15 中，$A \stackrel{*}{\Rightarrow} c\gamma$，因此 c 在 FIRST(A)中。

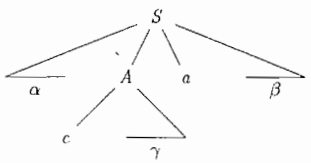

图 4-15 终结符号 c 在 FIRST(A)中且 a 在 FOLLOW(A)中

我们先简单介绍一下如何在预测分析中使用 FIRST。考虑两个 A 产生式 $A \rightarrow \alpha \mid \beta$，其中 FIRST($\alpha$)和 FIRST($\beta$)是不相交的集合。那么我们只需要查看下一个输入符号 a，就可以在这两个 A 产生式中进行选择。因为 a 只能出现在 FIRST(α)或 FIRST(β)中，但不能同时出现在两个集合中。比如，如果 a 在 FIRST(β)中，就选择 $A \rightarrow \beta$。在 4.4.3 中定义 LL(1)文法时将深入研究这个思想。

对于非终结符号 A，FOLLOW(A)被定义为可能在某些句型中紧跟在 A 右边的终结符号的集合。也就是说，如果存在如图 4-15 所示形如 $S \stackrel{*}{\Rightarrow} \alpha A a \beta$ 的推导，终结符号 a 就在 FOLLOW(A)中，其中 α 和 β 是文法符号串。请注意，在这个推导的某个阶段，A 和 a 之间可能存在一些文法符号。但如果这样，这些符号会推导得到 ϵ 并消失。另外，如果 A 是某些句型的最右符号，那么 \$ 也在 FOLLOW(A)中。回忆一下，\$ 是一个特殊的“结束标记”符号，我们假设它不是任何文法的符号。

计算各个文法符号 X 的 FIRST(X)时，不断应用下列规则，直到再没有新的终结符号或 ϵ 可以被加入到任何 FIRST 集合中为止。

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 $X \rightarrow Y_1 Y_2 \cdots Y_k$ 是一个产生式，其中 $k \geqslant 1$，那么如果对于某个 i，a 在 FIRST(Y_i)中且 ϵ 在所有的 FIRST(Y_1)、FIRST(Y_2)、…、FIRST(Y_{i-1})中，就把 a 加入到 FIRST(X)中。也就是说，$Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$。如果对于所有的 $j = 1, 2, \cdots, k$，ϵ 在 FIRST(Y_j)中，那么将 ϵ 加入到 FIRST(X)中。比如，FIRST(Y_1)中的所有符号一定在 FIRST(X)中。如果 Y_1 不能推导出 ϵ，那么我们就不会再向 FIRST(X)中加入任何符号，但是如果 $Y_1 \stackrel{*}{\Rightarrow} \epsilon$，那么我们就加上 FIRST($Y_2$)，依此类推。

3) 如果 $X \rightarrow \epsilon$ 是一个产生式，那么将 ϵ 加入到 FIRST(X)中。

现在，我们可以按照如下方式计算任何串 $X_1 X_2 \cdots X_n$ 的 FIRST 集合。向 FIRST($X_1 X_2 \cdots X_n$)加入 $F(X_1)$中所有的非 ϵ 符号。如果 ϵ 在 FIRST(X_1)中，再加入 FIRST(X_2)中的所有非 ϵ 符号；如果 ϵ 在 FIRST(X_1)和 FIRST(X_2)中，加入 FIRST(X_3)中的所有非 ϵ 符号，依此类推。最后，如果对所有的 i，ϵ 都在 FIRST(X_i)中，那么将 ϵ 加入到 FIRST($X_1 X_2 \cdots X_n$)中。

计算所有非终结符号 A 的 FOLLOW(A)集合时，不断应用下面的规则，直到再没有新的终结

符号可以被加入到任意 FOLLOW 集合中为止。

1）将 $ 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 ϵ 之外的所有符号都在 FOLLOW(B)中。

3）如果存在一个产生式 $A \rightarrow \alpha B$，或存在产生式 $A \rightarrow \alpha B\beta$ 且 FIRST(β)包含 ϵ，那么 FOLLOW(A)中的所有符号都在 FOLLOW(B)中。

例 4.30 再次考虑非左递归的文法(4.28)。那么：

1）FIRST(F) = FIRST(T) = FIRST(E) = {(, **id**}。要知道为什么，请注意 F 的两个产生式的体以终结符号 **id** 和左括号开头。T 只有一个产生式，而该产生式的体以 F 开头。又因为 F 不能推导出 ϵ，所以 FIRST(T)必然和 FIRST(F)相同。对于 FIRST(E)也可以做同样的论证。

2）FIRST(E') = {+, ϵ}。理由是 E'的两个产生式中，一个产生式的体以终结符号 + 开头，且另一个产生式的体为 ϵ。只要一个非终结符号推导出 ϵ，我们就会把 ϵ 放到该终结符号的 FIRST 集合中。

3）FIRST(T') = {*, ϵ}。它的论证过程和 FIRST(E')的论证过程类似。

4）FOLLOW(E) = FOLLOW(E') = {), $}。因为 E 是开始符号，FOLLOW(E)一定包含 $。产生式体（$E$）说明了右括号为什么在 FOLLOW($E$)中。对于 E'，请注意这个非终结符号只出现在 E 产生式的体的尾部，因此 FOLLOW(E')必然和 FOLLOW(E)相同。

5）FOLLOW(T) = FOLLOW(T') = {+,), $}。请注意，$T$ 在产生式体中出现时只有 E'跟在后面。因此，FIRST(E')中除 ϵ 之外的所有符号一定都在 FOLLOW(T)中。这解释了 + 出现在 FOLLOW(T)中的原因。然而，因为 FIRST(E')包含 ϵ（即 $E' \Rightarrow \epsilon$），且 E'就是在 E 产生式的体中跟在 T 后面的全部符号，因此 FOLLOW(E) 中的所有符号都在 FOLLOW(T)中。这解释了符号 $ 和右括号出现在 FOLLOW(T)中的原因。至于 T'，因为它只出现在 T 产生式的尾部，因此必然有 FOLLOW(T') = FOLLOW(T)。

6）FOLLOW(F) = {+, *,), $}。论证过程和第 5 点中对 T 的论证过程类似。 □

4.4.3 LL(1)文法

对于称为 LL(1)的文法，我们可以构造出预测分析器，即不需要回溯的递归下降语法分析器。LL(1)中的第一个“L”表示从左向右扫描输入，第二个“L”表示产生最左推导，而“1”则表示在每一步中只需要向前看一个输入符号来决定语法分析动作。

预测分析器的转换图

转换图有助于将预测分析器可视化。比如，图 4-16a 中显示了文法(4.28)中非终结符号 E 和 E'的转换图。要构造一个文法的转换图，首先要消除左递归，然后对文法提取左公因子。然后对每个非终结符号 A：

1）创建一个初始状态和一个结束(返回)状态。

2）对于每个产生式 $A \rightarrow X_1X_2\cdots X_n$，创建一个从初始状态到结束状态的路径，路径中各条边的标号为 X_1、X_2、…、X_n。如果 $A \rightarrow \epsilon$，那么这条路径就是一条标号为 ϵ 的边。

预测分析器的转换图和词法分析器的转换图是不同的。分析器的转换图对每个非终结符号都有一个图。图中边的标号可以是词法单元，也可以是非终结符号。词法单元上的转换表示当该词法单元是下一个输入符号时我们应该执行这个转换。非终结符号 A 上的转换表示对

A 的过程的一次调用。

对于一个 LL(1)文法,将 ϵ 边作为默认选择可以解决是否选择一个 ϵ 边的二义性问题。

转换图可以化简,前提是各条路径上的文法符号序列必须保持不变。我们也可以将一条标号为非终结符号 A 的边替换为 A 的转换图。图 4-16a 和图 4-16b 中的转换图是等价的:如果我们跟踪从 E 到结束状态的路径,并替换 E',那么在这两组图中,沿着这些路径的文法符号都组成了形如 $T+T+\cdots+T$ 的串。图 4-16b 中的图可以从图 4-16a 通过转换而得到。转换的方法类似于 2.5.4 节所述的方法。在该节中,我们使用尾递归消除和过程体替代的方法来优化一个非终结符号的相应过程。

LL(1)文法已经足以描述大部分程序设计语言构造,虽然在为源语言设计适当的文法时需要多加小心。比如,左递归的文法和二义性的文法都不可能是 LL(1)的。

一个文法 G 是 LL(1)的,当且仅当 G 的任意两个不同的产生式 $A\rightarrow\alpha \mid \beta$ 满足下面的条件:

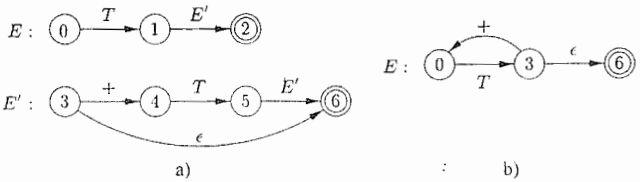

图 4-16 文法 4.28 的非终结符号 E 和 E' 的转换图

1) 不存在终结符号 a 使得 α 和 β 都能够推导出以 a 开头的串。

2) α 和 β 中最多只有一个可以推导出空串。

3) 如果 $\beta \stackrel{*}{\Rightarrow}\epsilon$,那么 α 不能推导出任何以 FOLLOW(A)中某个终结符号开头的串。类似地,如果 $\alpha \stackrel{*}{\Rightarrow}\epsilon$,那么 β 不能推导出任何以 FOLLOW(A)中某个终结符号开头的串。

前两个条件等价于说 FIRST(α)和 FIRST(β)是不相交的集合。第三个条件等价于说如果 ϵ 在 FIRST(β)中,那么 FIRST(α)和 FOLLOW(A)是不相交的集合,并且当 ϵ 在 FIRST(α)中时类似结论成立。

之所以能够为 LL(1)文法构造预测分析器,原因是只需要检查当前输入符号就可以为一个非终结符号选择正确的产生式。因为有关控制流的各个语言构造带有不同的关键字,它们通常满足 LL(1)的约束。比如,如果我们有如下产生式

$$\begin{aligned} stmt \rightarrow & \textbf{if}\ (expr)\ stmt\ \textbf{else}\ stmt \\ \mid & \textbf{while}\ (expr)\ stmt \\ \mid & \{stmt\_list\} \end{aligned}$$

那么关键字 **if**、**while** 和符号{告诉我们:如果在输入中找到一个语句,哪个产生式是唯一可能匹配成功的。

接下来给出的算法把 FIRST 和 FOLLOW 集合中的信息放到一个预测分析表 $M[A, a]$ 中。这是一个二维数组,其中 A 是一个非终结符号,a 是一个终结符号或特殊符号 \$,即输入的结束标记。该算法基于如下的思想:只有当下一个输入符号 a 在 FIRST(α)中时才选择产生式 $A\rightarrow\alpha$。只有当 $\alpha=\epsilon$ 时,或更加一般化的 $\alpha \stackrel{*}{\Rightarrow}\epsilon$ 时,情况才有些复杂。在这种情况下,如果当前输入符号在 FOLLOW(A)中,或者已经到达输入中的 \$ 符号且 \$ 在 FOLLOW(A)中,那么我们仍应该选择 $A\rightarrow\alpha$。

算法 4.31 构造一个预测分析表。

输入: 文法 G。

输出: 预测分析表 M。

方法：对于文法 G 的每个产生式 $A\to\alpha$，进行如下处理：

1）对于 FIRST(α)中的每个终结符号 a，将 $A\to\alpha$ 加入到$M[A, a]$中。

2）如果 ϵ 在 FIRST(α)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(α)中，且 \$ 在 FOLLOW(A)中，也将 $A\to\alpha$ 加入到 $M[A, \$]$中。

在完成上面的操作之后，如果 $M[A, a]$中没有产生式，那么将 $M[A, a]$设置为 **error**（我们通常在表中用一个空条目表示）。 □

例 4.32 对于表达式文法(4.28)，算法 4.31 生成了图 4-17 中的预测分析表。空白条目表示错误条目；非空白的条目中指明了应该用其中的产生式来扩展相应的非终结符号。

非终结符号	输入符号					
	id	+	*	(	)	\$
E	$E\to TE'$			$E\to TE'$		
E'		$E'\to +TE'$			$E'\to\epsilon$	$E'\to\epsilon$
T	$T\to FT'$			$T\to FT'$		
T'		$T'\to\epsilon$	$T'\to *FT'$		$T'\to\epsilon$	$T'\to\epsilon$
F	$F\to$ id			$F\to(E)$		

图 4-17 例 4.32 的预测分析表 M

考虑产生式 $E\to TE'$。因为

$$\text{FIRST}(TE') = \text{FIRST}(T) = \{(, \textbf{id}\}$$

这个产生式被加到 $M[E, (]$和 $M[E, \textbf{id}]$中。因为 FIRST($+TE'$) = { + }，产生式$E'\to +\ TE'$被加入到 $M[E', +]$中。因为 FOLLOW(E') = {), \$ }，产生式 $E'\to\epsilon$ 被加入到 $M[E',)]$和 $M[E', \$]$中。 □

算法 4.31 可以应用于任何文法 G，生成该文法的语法分析表 M。对于每个 LL(1)文法，分析表中的每个条目都唯一地指定了一个产生式，或者标明一个语法错误。然而，对于某些文法，M 中可能会有一些多重定义的条目。比如，如果 G 是左递归的或二义性的，那么 M 至少会包含一个多重定义的条目。虽然可以轻松对其进行消除左递归和提取左公因子的操作，但是仍然存在一些这样的文法，它们不存在等价的 LL(1)文法。

下面例子中的语言根本没有相应的 LL(1)文法。

例 4.33 下面重复一下例子 4.22 中的文法。该文法抽象地表示了悬空 - else 的问题。

$$S\to iEtSS' \mid a$$
$$S'\to eS \mid \epsilon$$
$$E\to b$$

这个文法的语法分析表显示在图 4-18 中。$M[S', e]$的条目同时包含了 $S'\to eS$ 和 $S'\to\epsilon$。

非终结符号	输入符号					
	a	b	e	i	t	\$
S	$S\to a$			$S\to iEtSS'$		
S'			$S'\to\epsilon$ $S'\to eS$			$S'\to\epsilon$
E		$E\to b$				

图 4-18 例 4.33 的分析表 M

这个文法是二义性的。当在输入中看到 e(代表 **else**)时，解决选择使用哪个产生式的问题就会显露出此文法的二义性。解决这个二义性问题时，我们可以选择产生式 $S'\rightarrow eS$。这个选择就相当于把 **else** 和前面最近的 **then** 关联起来。请注意，选择 $S'\rightarrow\epsilon$ 将使得 e 永远不可能被放到栈中或者从输入中被消除，因此选择这个产生式一定是错误的。 □

4.4.4 非递归的预测分析

我们可以构造出一个非递归的预测分析器，它显式地维护一个栈结构，而不是通过递归调用的方式隐式地维护栈。这样的语法分析器可以模拟最左推导的过程。如果 w 是至今为止已经匹配完成的输入部分，那么栈中保存的文法符号序列 α 满足

$$S \underset{lm}{\overset{*}{\Rightarrow}} w\alpha$$

图 4-19 中的由分析表驱动的语法分析器有一个输入缓冲区，一个包含了文法符号序列的栈，一个由算法 4.31 构造得到的分析表，以及一个输出流。它的输入缓冲区中包含要进行语法分析的串，串后面跟有结束标记 \$。我们复用符号 \$ 来标记栈底。在开始时刻，栈中 \$ 的上方是开始符号 S。

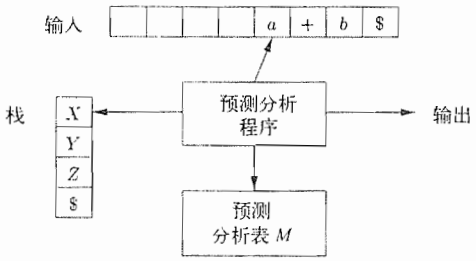

图 4-19 一个分析表驱动的预测分析器的模型

语法分析器由一个程序控制。该程序考虑栈顶符号 X 和当前输入符号 a。如果 X 是一个非终结符号，该分析器查询分析表 M 中的条目 $M[X, a]$来选择一个 X 产生式。(这里可以执行一些附加的代码，比如构造一个语法分析树结点的代码。)否则，它检查终结符号 X 和当前输入符号 a 是否匹配。

这个语法分析器的行为可以使用它的*格局*(configuration)来描述。格局描述了栈中的内容和余下的输入。下面的算法描述了如何处理格局。

算法 4.34 表驱动的预测语法分析。

输入：一个串 w，文法 G 的预测分析表 M。

输出：如果 w 在 $L(G)$ 中，输出 w 的一个最左推导；否则给出一个错误指示。

方法：最初，语法分析器的格局如下：输入缓冲区中是 w\$，而 G 的开始符号 S 位于栈顶，它的下面是 \$。图 4-20 中的程序使用预测分析表 M 生成了处理这个输入的预测分析过程。 □

```
设置 ip 使它指向 w 的第一个符号，其中 ip 是输入指针;
令 X = 栈顶符号;
while ( X ≠ $ ) { /* 栈非空 */
    if ( X 等于 ip 所指向的符号 a ) 执行栈的弹出操作，将 ip 向前移动一个位置;
    else if ( X 是一个终结符号 ) error();
    else if ( M[X, a] 是一个报错条目 ) error();
    else if ( M[X, a] = X → Y1Y2 ··· Yk ) {
        输出产生式 X → Y1Y2 ··· Yk;
        弹出栈顶符号;
        将 Yk, Yk-1, ..., Y1 压入栈中，其中 Y1 位于栈顶。
    }
    令 X = 栈顶符号;
}
```

图 4-20 预测分析算法

例 4.35 考虑文法(4.28)。我们已经在图 4-17 中看到了它的预测分析表。处理输入**id + id * id**

时，算法 4.34 的非递归预测分析器顺序执行图 4-21 中显示的各个步骤。这些步骤对应于一个最左推导（完整的推导过程见图 4-12）：

已匹配	栈	输入	动作
	$E\$$	$\mathrm{id}+\mathrm{id}*\mathrm{id}\$$	
	$TE'\$$	$\mathrm{id}+\mathrm{id}*\mathrm{id}\$$	输出 $E \to TE'$
	$FT'E'\$$	$\mathrm{id}+\mathrm{id}*\mathrm{id}\$$	输出 $T \to FT'$
	$\mathrm{id}\ T'E'\$$	$\mathrm{id}+\mathrm{id}*\mathrm{id}\$$	输出 $F \to \mathrm{id}$
id	$T'E'\$$	$+\,\mathrm{id}*\mathrm{id}\$$	匹配 id
id	$E'\$$	$+\,\mathrm{id}*\mathrm{id}\$$	输出 $T' \to \epsilon$
id	$+\,TE'\$$	$+\,\mathrm{id}*\mathrm{id}\$$	输出 $E' \to +\,TE'$
id +	$TE'\$$	$\mathrm{id}*\mathrm{id}\$$	匹配 +
id +	$FT'E'\$$	$\mathrm{id}*\mathrm{id}\$$	输出 $T \to FT'$
id +	$\mathrm{id}\ T'E'\$$	$\mathrm{id}*\mathrm{id}\$$	输出 $F \to \mathrm{id}$
id + id	$T'E'\$$	$*\,\mathrm{id}\$$	匹配 id
id + id	$*\,FT'E'\$$	$*\,\mathrm{id}\$$	输出 $T' \to *\,FT'$
id + id *	$FT'E'\$$	$\mathrm{id}\$$	匹配 *
id + id *	$\mathrm{id}\ T'E'\$$	$\mathrm{id}\$$	输出 $F \to \mathrm{id}$
id + id * id	$T'E'\$$	$\$$	匹配 id
id + id * id	$E'\$$	$\$$	输出 $T' \to \epsilon$
id + id * id	$\$$	$\$$	输出 $E' \to \epsilon$

图 4-21　对输入 **id** + **id** * **id** 进行预测分析时执行的步骤

$$E \underset{lm}{\Rightarrow} T\,E' \underset{lm}{\Rightarrow} F\,T'E' \underset{lm}{\Rightarrow} \mathbf{id}\,T'\,E' \underset{lm}{\Rightarrow} \mathbf{id}\,E' \underset{lm}{\Rightarrow} \mathbf{id}\ +\ T\,E' \underset{lm}{\Rightarrow} \cdots$$

请注意，这个推导中的各个句型对应于已经被匹配的输入部分（见图中的已匹配列）加上栈中的内容。我们显示已匹配输入就是为了强调这种对应关系。因为同样的原因，在图中将栈顶显示在左边。当我们考虑自底向上语法分析时，将栈顶显示在右边会更加自然。分析器的输入指针指向“输入”列中的串的最左边的符号。 □

4.4.5　预测分析中的错误恢复

在讨论错误恢复时要考虑一个由分析表驱动的预测分析器的栈，因为这个栈明确地显示了语法分析器期望用哪些终结符号及非终结符号来匹配余下的输入。这个技术也可以在递归下降语法分析过程中使用。

当栈顶的终结符号和下一个输入符号不匹配时，或者当非终结符号 A 处于栈顶，a 是下一个输入符号，且 $M[A, a]$ 为 **error**（即相应的语法分析表条目为空）时，预测语法分析过程就可以检测到语法错误。

恐慌模式

恐慌模式的错误恢复是基于下面的思想。语法分析器忽略输入中的一些符号，直到输入中出现由设计者选定的同步词法单元集合中的某个词法单元。它的有效性依赖于同步集合的选取。选取这个集合的原则是应该使得语法分析器能够从实践中可能遇到的错误中快速恢复。下面是一些启发式规则：

1）首先将 FOLLOW(A) 中的所有符号都放到非终结符号 A 的同步集合中。如果我们不断忽略一些词法单元，直到碰到了 FOLLOW(A) 中的某个元素，然后再将 A 从栈中弹出，那么很可能语法分析过程就能够继续进行。

2）只使用 FOLLOW(A) 作为 A 的同步集合是不够的。比如，C 语言用分号表示一个语句结束，那么语句开头的关键字可能不会出现在代表表达式的非终结符号的 FOLLOW 集合中。因此，在一个赋值语句之后遗漏分号可能会使得语法分析器忽略下一个语句开头的关键字。一个语言的各个构造之间常常存在某个层次结构。比如，表达式出现在语句内部，而语句出现在块内部，

等等。我们可以把较高层构造的开始符号加入到较低层构造的同步集合中去。比如，我们可以把语句开头的关键字加入到生成表达式的非终结符号的同步集合中去。

3）如果我们把 FIRST(A)中的符号加入到非终结符号 A 的同步集合中，那么当 FIRST(A)中的某个符号出现在输入中时，我们就有可能可以根据 A 继续进行语法分析。

4）如果一个非终结符号可以生成空串，那么可以把推导出 ϵ 的产生式当作默认值使用。这么做可能会延迟对某些错误的检测，但是不会使错误被漏检。这个方法可以减少我们在处理错误恢复时需要考虑的非终结符号的数量。

5）如果栈顶的一个终结符号不能和输入匹配，一个简单的想法是将该终结符号弹出栈，并发出一个消息称已经插入了这个终结符号，同时继续进行语法分析。从效果上看，这个方法是将所有其他词法单元的集合作为一个词法单元的同步集合。

例 4.36 当按照常用的表达式文法(4.28)对表达式进行语法分析时，使用 FIRST 和 FOLLOW 符号作为同步集合就能够很好地完成任务。图 4-17 中此文法的语法分析表在图 4-22 中再次给出。图 4-22 中使用“synch”来表示根据相应非终结符号的 FOLLOW 集合得到的同步词法单元。各个非终结符号的 FOLLOW 集合是从例子 4.30 中得到的。

图 4-22 中的分析表将按照如下方式使用。如果语法分析器查看 $M[A, a]$ 并发现它是空的，那么输入符号 a 就被忽略。如果该条目是“synch”，那么在试图继续分析时，栈顶的非终结符号被弹出。如果栈顶的词法单元和输入符号不匹配，那么我们就按上述方式从栈中弹出这个单元。

非终结符号	输入符号					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ id	synch	synch	$F \to (E)$	synch	synch

图 4-22 加入到图 4-17 的预测分析表中的同步词法单元

对于错误输入 + **id** * + **id**，语法分析器以及图 4-22 中的错误恢复机制的工作过程如图4-23所示。□

栈	输入	说明
E \$	+ id * + id \$	错误，略过）
E \$	id * + id \$	id 在 FIRST(E)中
TE' \$	id * + id \$	
$FT'E'$ \$	id * + id \$	
id $T'E'$ \$	id * + id \$	
$T'E'$ \$	* + id \$	
* $FT'E'$ \$	* + id \$	
$FT'E'$ \$	+ id \$	错误，$M[F, +]$ = synch
$T'E'$ \$	+ id \$	F 已经被弹出栈
E' \$	+ id \$	
+ TE' \$	+ id \$	
TE' \$	id \$	
$FT'E'$ \$	id \$	
id $T'E'$ \$	id \$	
$T'E'$ \$	\$	
E' \$	\$	
\$	\$	

图 4-23 一个预测分析器所做的语法分析和错误恢复步骤

上面的关于恐慌模式错误恢复的讨论没有考虑有关错误消息的重要问题。编译器的设计者必须提供足够的包含有用信息的错误消息，它不仅描述相应的错误，还必须引导人们注意错误被发现的地方。

短语层次的恢复

短语层次错误恢复的实现方法是在预测语法分析表的空白条目中填写指向处理例程的指针。这些例程可以改变、插入或删除输入中的符号，并发出适当的错误消息。它们也可能执行一些出栈操作。改变栈中符号或将新符号压入栈中可能会引起一些问题，其原因有多个。首先，由语法

分析器执行的动作可能根本不对应于语言中任何句子的推导过程。第二，我们必须保证分析器不会陷入无限循环。防止出现无限循环的一个好办法是保证任何恢复动作最终都会消耗掉某个输入符号(当到达输入结尾处时，则需要保证栈中的内容会变少)。

4.4.6 4.4 节的练习

练习 4.4.1：为下面的每一个文法设计一个预测分析器，并给出预测分析表。你可能先要对文法进行提取左公因子或消除左递归的操作。

1) 练习 4.2.2(1)中的文法。

2) 练习 4.2.2(2)中的文法。

3) 练习 4.2.2(3)中的文法。

4) 练习 4.2.2(4)中的文法。

5) 练习 4.2.2(5)中的文法。

6) 练习 4.2.2(7)中的文法。

!! **练习 4.4.2**：有没有可能通过某种方法修改练习 4.2.1 中的文法，构造出一个与该练习中的语言(运算分量为 a 的后缀表达式)对应的预测分析器？

练习 4.4.3：计算练习 4.2.1 的文法的 FIRST 和 FOLLOW 集合。

练习 4.4.4：计算练习 4.2.2 中各个文法的 FIRST 和 FOLLOW 集合。

练习 4.4.5：文法 $S \rightarrow aSa \mid aa$ 生成了所有由 a 组成的长度为偶数的串。我们可以为这个文法设计一个带回溯的递归下降分析器。如果我们选择先用产生式 $S \rightarrow aa$ 展开，那么我们只能识别到串 aa。因此，任何合理的递归下降分析器将首先尝试 $S \rightarrow aSa$。

1) 说明这个递归下降分析器识别输入 aa、$aaaa$ 和 $aaaaaaaa$，但是识别不了 $aaaaaa$。

!! 2)这个递归下降分析器识别什么样的语言？

下面的练习是构造任意文法的“Chomsky 范式”的有用步骤。Chomsky 范式将在练习 4.4.8 中定义。

! **练习 4.4.6**：如果一个文法没有产生式体为 ϵ 的产生式(称为 ϵ *产生式*)，那么这个文法就是*无 ϵ 产生式*的。

1) 给出一个算法，它的功能是把任何文法转变成一个无 ϵ 产生式的生成相同语言的文法(唯一可能的例外是空串——没有哪个无 ϵ 产生式的文法能生成 ϵ)。*提示*：首先找出所有可能为空的非终结符号。非终结符号可能为空是指它(可能通过很长的推导)生成 ϵ。

2) 将你的算法应用于文法 $S \rightarrow aSbS \mid bSaS \mid \epsilon$。

! **练习 4.4.7**：*单产生式*(single production)是指其产生式体为单个非终结符号的产生式，即形如 $A \rightarrow B$ 的产生式，其中 A、B 为任意的非终结符号。

1) 给出一个算法，它可以把任何文法转变成一个生成相同语言(唯一可能的例外是空串)的、无 ϵ 产生式、无单产生式的文法。*提示*：首先消除 ϵ-产生式，然后找出所有满足下列条件的非终结符号对 A 和 B：存在一系列单产生式使得 $A \Rightarrow B$。

2) 将你的算法应用于 4.1.2 节的算法(4.1)。

3) 说明作为(1)的一个结果，我们可以把一个文法转变为一个没有环(即对某个非终结符号 A 存在一步或多步的推导 $A \Rightarrow A$)的等价文法。

!! **练习 4.4.8**：如果一个文法的每个产生式要么形如 $A \rightarrow BC$，要么形如 $A \rightarrow a$，其中 A、B 和 C 是非终结符号，而 a 是终结符号，那么这个文法就称为 Chomsky *范式*(Chomsky Normal Form，CNF)文法。说明如何将任意文法转变成一个生成相同语言(唯一可能的例外是空串——没有 CNF 文法可以生成 ϵ)的 CNF 文法。

! 练习 4.4.9：对于每个具有上下文无关文法的语言，其长度为 n 的串可以在 $O(n^3)$ 的时间内完成识别。完成这种识别工作的一个简单方法称为 *Cocke-Younger-Kasami*（CYK）算法。该算法基于动态规划技术。也就是说，给定一个串 $a_1a_2\cdots a_n$，我们构造出一个 $n \times n$ 的表 T 使得 T_{ij} 是可以生成子串 $a_ia_{i+1}\cdots a_j$ 的非终结符号的集合。如果基础文法是 CNF 的（见练习 4.4.8），那么只要我们按照正确的顺序来填表：先填 $j-i$ 值最小的条目，则表中的每一个条目都可以在 $O(n)$ 时间内填写完毕。给出一个能够正确填写这个表的条目的算法，并说明你的算法的时间复杂度为 $O(n^3)$。填完这个表之后，你如何判断 $a_1a_2\cdots a_n$ 是否在这个语言中？

! 练习 4.4.10：说明我们如何能够在填好练习 4.4.9 中的表之后，在 $O(n)$ 时间内获得 $a_1a_2\cdots a_n$ 对应的一棵语法分析树？提示：修改练习 4.4.9 中的表 T，使得对于表的每个条目 T_{ij} 中的每个非终结符号 A，这个表同时记录了其他条目中的哪两个非终结符号组成的对偶使得我们将 A 放到 T_{ij} 中。

! 练习 4.4.11：修改练习 4.4.9 中的算法，使得对于任意符号串，它可以找出至少需要执行多少次插入、删除和修改错误（每个错误是一个字符）的操作才能将这个串变成基础文法的语言的句子。

! 练习 4.4.12：图 4-24 中给出了对应于某些语句的文法。你可以将 e 和 s 当作分别代表条件表达式和“其他语句”的终结符号。如果我们按照下列方法来解决因为展开可选“else”（非终结符号 *stmtTail*）而引起的冲突：当我们从输入中看到一个 **else** 时就选择消耗掉这个 **else**。使用 4.4.5 节中描述的同步符号的思想：

1）为这个文法构造一个带有错误纠正信息的预测分析表。

2）给出你的语法分析器在处理下列输入时的行为：

ⓐ **if** e **then** s ; **if** e **then** s **end**

ⓑ **while** e **do begin** s ; **if** e **then** s ; **end**

stmt	→	**if** e **then** *stmt stmtTail*
	|	**while** e **do** *stmt*
	|	**begin** *list* **end**
	|	s
stmtTail	→	**else** *stmt*
	|	ϵ
list	→	*stmt listTail*
listTail	→	; *list*
	|	ϵ

图 4-24 某种类型语句的文法

4.5 自底向上的语法分析

一个自底向上的语法分析过程对应于为一个输入串构造语法分析树的过程，它从叶子结点（底部）开始逐渐向上到达根结点（顶部）。将语法分析描述为语法分析树的构造过程会比较方便，虽然编译器前端实际上不会显式地构造出语法分析树，而是直接进行翻译。图 4-25 中显示的分析树的快照序列演示了按照表达式文法(4.1)对词法单元序列 **id** * **id** 进行的自底向上语法分析的过程。

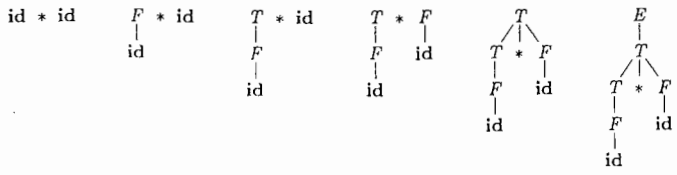

图 4-25 **id** * **id** 的自底向上分析过程

本节将介绍一个被称为移入－归约语法分析的自底向上语法分析的通用框架。我们将在 4.6 节和 4.7 节中讨论 LR 文法类，它是最大的、可以构造出相应移入－归约语法分析器的文法类。虽然手工构造一个 LR 语法分析器的工作量非常大，但借助语法分析器自动生成工具可以使人们

轻松地根据适当的文法构造出高效的 LR 分析器。本节中的概念有助于写出合适的文法，从而有效利用 LR 分析器生成工具。实现语法分析器生成工具的算法将在 4.7 节中给出。

4.5.1 归约

我们可以将自底向上语法分析过程看成将一个串 w“归约”为文法开始符号的过程。在每个归约(reduction)步骤中，一个与某产生式体相匹配的特定子串被替换为该产生式头部的非终结符号。

在自底向上语法分析过程中，关键问题是何时进行归约以及应用哪个产生式进行归约。

例 4.37 图 4-25 中的快照演示了一个归约序列，相应的文法是表达式文法(4.1)。我们将使用如下的符号串序列来讨论这个归约过程：

$$\mathbf{id} * \mathbf{id},\ F * \mathbf{id},\ T * \mathbf{id},\ T * F,\ T,\ E$$

这个序列中的符号串由快照中各相应子树的根结点组成。这个序列从输入串 **id** * **id** 开始。第一次归约使用产生式 $F \rightarrow \mathbf{id}$，将最左边的 **id** 归约为 F，得到串 F * **id**。第二次归约将 F 归约为 T，生成 T * **id**。

现在我们可以选择是对串 T 还是对由第二个 **id** 组成的串进行归约，其中 T 是 $E \rightarrow T$ 的体，而第二个 **id** 是 $F \rightarrow \mathbf{id}$ 的体。我们没有将 T 归约为 E，而是将第二个 **id** 归约为 F，得到串 $T * F$。然后这个串被归约为 T。最后将 T 归约为开始符号 E，从而结束整个语法分析过程。 □

根据定义，一次归约是一个推导步骤的反向操作(回顾一下，一次推导步骤将句型中的一个非终结符号替换为该符号的某个产生式的体)。因此，自底向上语法分析的目标是反向构造一个推导过程。下面的推导对应于图 4-25 中的分析过程：

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

这个推导过程实际上是一个最右推导。

4.5.2 句柄剪枝

对输入进行从左到右的扫描，并在扫描过程中进行自底向上语法分析，就可以反向构造出一个最右推导。非正式地讲，“句柄”是和某个产生式体匹配的子串，对它的归约代表了相应的最右推导中的一个反向步骤。

比如，在按照表达式文法(4.1)对 $\mathbf{id}_1 * \mathbf{id}_2$ 进行语法分析时，各个句柄如图 4-26 所示。为了表示得更清楚，我们为其中的词法单元 **id** 加上了下标。虽然 T 是产生式 $E \rightarrow T$ 的体，但符号 T 并不是句型 $T * \mathbf{id}_2$ 的一个句柄。假如 T 真的被替换为 E，我们将得到串 $E * \mathbf{id}_2$，而这个串不能从开始符号 E 推导得到。因此，和某个产生式体匹配的最左子串不一定是句柄。

正式地讲，如果有 $S \underset{rm}{\Rightarrow^*} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$(如图 4-27所示)，那么紧跟 α 的产生式 $A \rightarrow \beta$ 是 $\alpha\beta w$ 的一个句柄(handle)。换句话说，最右句型 γ 的一个句柄是满足下述条件的产生式 $A \rightarrow \beta$ 及串 β 在 γ 中出现的位置：将这个位置上的 β 替换为 A 之后得到的串是 γ 的某个最右推导序列中出现在位于 γ 之前的最右句型。

最右句型	句柄	归约用的产生式
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \rightarrow F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$T \rightarrow T * F$
T	T	$E \rightarrow T$

图 4-26 $\mathbf{id}_1 * \mathbf{id}_2$ 的语法分析过程中出现的句柄

请注意，句柄右边的串 w 一定只包含终结符号。为方便起见，我们把产生式体 β(而不是 $A \rightarrow \beta$)称为一个句柄。注意，我们说的是“一个句柄”，而不是“唯一句柄”。这是因为文法可能是二义性的，$\alpha\beta w$ 可能存在多个最右推导。如果一个文法是无二义性的，那么该文法的每个右句型都有且只有一个句柄。

通过“句柄剪枝”可以得到一个反向的最右推导。也就是说，我们从被分析的终结符号串 w

开始。如果 w 是当前文法的句子，那么令 $w=\gamma_n$，其中 γ_n 是某个未知最右推导的第 n 个最右句型。

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

为了以相反顺序重构这个推导，我们在 γ_n 中寻找句柄 β_n，并将 β_n 替换为相关产生式 $A_n \rightarrow \beta_n$ 的头部，得到前一个最右句型 γ_{n-1}。请注意，我们现在还不知道如何发现句柄，但是我们很快就会介绍多个寻找句柄的方法。

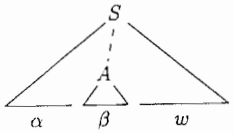

图 4-27 $\alpha\beta w$ 的语法分析树中的一个句柄 $A \rightarrow \beta$

然后我们重复这个过程。也就是说，我们在 γ_{n-1} 中寻找句柄 β_{n-1}，并对这个句柄进行归约，得到最右句型 γ_{n-2}。如果我们按照这个过程得到了一个只包含开始符号 S 的最右句型，那么就可以停止分析并宣称语法分析过程成功完成。将归约过程中用到的产生式反向排序，就得到了输入串的一个最右推导过程。

4.5.3 移入－归约语法分析技术

移入－归约语法分析是自底向上语法分析的一种形式。它使用一个栈来保存文法符号，并用一个输入缓冲区来存放将要进行语法分析的其余符号。我们将看到，句柄在被识别之前，总是出现在栈的顶部。

我们使用 \$ 来标记栈的底部以及输入的右端。按照惯例，在讨论自底向上语法分析的时候，我们将栈顶显示在右侧，而不是像在自顶向下语法分析中那样显示在左侧。如下所示，开始的时候栈是空的，并且输入串 w 存放在输入缓冲区中。

栈	输入
\$	w \$

在对输入串的一次从左到右扫描过程中，语法分析器将零个或多个输入符号移到栈的顶端，直到它可以对栈顶的一个文法符号串 β 进行归约为止。它将 β 归约为某个产生式的头。语法分析器不断地重复这个循环，直到它检测到一个语法错误，或者栈中包含了开始符号且输入缓冲区为空为止：

栈	输入
\$$S$	\$

当进入这样的格局时，语法分析器停止运行，并宣称成功完成了语法分析。图 4-28 显示了一个移入－归约语法分析器在按照表达式文法(4.1)对输入串 $\mathbf{id}_1 * \mathbf{id}_2$ 进行语法分析时可能采取的动作。

虽然主要的语法分析操作是移入和归约，但实际上一个移入－归约语法分析器可采取如下四种可能的动作：①移入，②归约，③接受，④报错。

1）移入(shift)：将下一个输入符号移到栈的顶端。

2）归约(reduce)：被归约的符号串的右端必然是栈顶。语法分析器在栈中确定这个串的左端，并决定用哪个非终结符号来替换这个串。

3）接受(accept)：宣布语法分析过程成功完成。

4）报错(error)：发现一个语法错误，并调用一个错误恢复子例程。

我们之所以能够在移入－归约语法分析中使用

栈	输入	动作
\$	$\mathbf{id}_1 * \mathbf{id}_2$ \$	移入
\$$\mathbf{id}_1$	$* \mathbf{id}_2$ \$	按照 $F \rightarrow \mathbf{id}$ 归约
\$$F$	$* \mathbf{id}_2$ \$	按照 $T \rightarrow F$ 归约
\$$T$	$* \mathbf{id}_2$ \$	移入
\$$T*$	$\mathbf{id}_2$ \$	移入
\$$T*\mathbf{id}_2$	\$	按照 $F \rightarrow \mathbf{id}$ 归约
\$$T*F$	\$	按照 $T \rightarrow T*F$ 归约
\$$T$	\$	按照 $E \rightarrow T$ 归约
\$$E$	\$	接受

图 4-28 一个移入－归约语法分析器在处理输入 $\mathbf{id}_1 * \mathbf{id}_2$ 时经历的格局

栈，是因为这个分析过程具有如下重要性质：句柄总是出现在栈的顶端，绝不会出现在栈的中间。要证明这个性质，我们只需要考虑任意最右推导中的两个连续步骤可能具有的形式。图4-29演示了两种可能的情况。在情况(1)中，A 被替换为 βBy，然后产生式体 βBy 中最右非终结符号 B 被替换为 γ。在情况(2)中，A 仍然首先被展开，但这次使用的产生式体 y 中只包含终结符号。下一个最右非终结符号 B 将位于 y 左侧的某个地方。

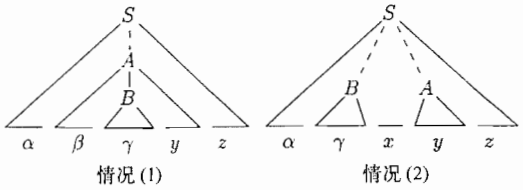

图 4-29 一个最右推导中两个连续步骤的两种情况

换句话说：

1) $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha Az \underset{rm}{\Rightarrow} \alpha\beta Byz \underset{rm}{\Rightarrow} \alpha\beta\gamma yz$

2) $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha BxAz \underset{rm}{\Rightarrow} \alpha Bxyz \underset{rm}{\Rightarrow} \alpha\gamma xyz$

反向考虑情况(1)，即一个移入－归约语法分析器刚刚到达如下格局的情况：

栈	输入
$\$\alpha\beta\gamma$	$yz\ \$$

语法分析器将句柄 γ 归约为 B，从而到达如下格局：

$\$\alpha\beta B$	$yz\ \$$

现在，语法分析器可以通过零次或多次移入动作，把串 y 移入到栈的上方，得到如下格局：

$\$\alpha\beta By$	$z\ \$$

其中，句柄 βBy 位于栈顶，它将被归约为 A。

现在考虑情况(2)。在格局

$\$\alpha\gamma$	$xyz\ \$$

中，句柄 γ 位于栈顶。将句柄 γ 归约为 B 之后，语法分析器可以把串 xy 移入栈中，得到位于栈顶的下一个句柄 y。该句柄可以被归约为 A：

$\$\alpha Bxy$	$z\ \$$

在这两种情况下，语法分析器在进行一次归约之后，都必须接着移入零个或多个符号才能在栈顶找到下一个句柄。因此它从不需要到栈中间去寻找句柄。

4.5.4 移入－归约语法分析中的冲突

有些上下文无关文法不能使用移入－归约语法分析技术。对于这样的文法，每个移入－归约语法分析器都会得到如下的格局：即使知道了栈中的所有内容以及接下来的 k 个输入符号，我们仍然无法判断应该进行移入还是归约操作(*移入/归约冲突*)，或者无法在多个可能的归约方法中选择正确的归约动作(*归约/归约冲突*)。现在我们给出一些语法构造的例子，这些构造的文法可能会出现这样的冲突。从技术上来讲，这些文法不在4.7节定义的 LR(k)文法类中，我们把它们称为非 LR 文法。LR(k)中的 k 表示在输入中向前看 k 个符号。在编译中使用的文法通常属于 LR(1)文法类，即最多只需要向前看一个符号。

例 4.38 一个二义性文法不可能是 LR 的。比如，考虑4.3节中的悬空-else 文法(4.14)：

$$\begin{aligned} stmt \rightarrow\ & \textbf{if}\ expr\ \textbf{then}\ stmt \\ & \mid \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\ & \mid \textbf{other} \end{aligned}$$

如果我们有一个移入－归约语法分析器处于格局

栈	输入
… **if** *expr* **then** *stmt*	**else** … $

中，那么不管栈中 **if** *expr* **then** *stmt* 之下是什么内容，我们都不能确定它是否是句柄。这里就出现了一个移入/归约冲突。根据输入中 **else** 之后的内容的不同，可能应该将 **if** *expr* **then** *stmt* 归约为 *stmt*，也可能应该将 **else** 移入然后再寻找另一个 *stmt*，从而找到完整的 *stmt* 产生式体 **if** *expr* **then** *stmt* **else** *stmt*。

请注意，经过修正的移入－归约语法分析技术可以对某些二义性文法进行语法分析，比如上面的 if-then-else 文法。如果我们在碰到 **else** 时选择移入来解决移入/归约冲突，语法分析器就会按照我们的期望运行，也就是将每个 **else** 和前一个尚未匹配的 **then** 相关联。我们将在 4.8 节讨论能够处理这种二义性文法的语法分析器。 □

另一个常见的冲突情况发生在我们确认已经找到句柄的时候。在这种情况下我们不能够根据栈中内容和下一个输入符号确定应该使用哪个产生式进行归约。下面的例子说明了这种情况。

例 4.39 假设我们有这样一个词法分析器，它不考虑各个名字的类型，而是对所有的名字都返回词法单元名 **id**。假设我们的语言在调用过程时会给出过程名字，并把调用参数放在括号内。并且假设引用数组的语法与此相同。因为在数组引用中对下标的翻译不同于过程调用中对参数的翻译，我们希望使用不同的产生式分别生成实在参数列表和下标列表。因此，我们的文法包含了图 4-30 中所示的产生式（还包含其他产生式）。

(1)	*stmt*	→	**id** (*parameter_list*)
(2)	*stmt*	→	*expr* := *expr*
(3)	*parameter_list*	→	*parameter_list* , *parameter*
(4)	*parameter_list*	→	*parameter*
(5)	*parameter*	→	**id**
(6)	*expr*	→	**id** (*expr_list*)
(7)	*expr*	→	**id**
(8)	*expr_list*	→	*expr_list* , *expr*
(9)	*expr_list*	→	*expr*

图 4-30 有关过程调用和数组引用的产生式

一个以 p(i, j)开头的语句将以词法单元流 **id**(**id**, **id**)的方式输入到语法分析器中。在将前三个词法单元移入到栈中后，移入－归约语法分析器将处于如下格局中：

栈	输入
… **id** (**id**	, **id**) …

显然，栈顶的 **id** 必须被归约，但使用哪个产生式呢？如果 p 是一个过程，那么正确的选择是产生式(5)；但如果 p 是一个数组，就该选择产生式(7)。栈中的内容并没有指出 p 是什么，必须使用从 p 的声明中获得的符号表中的信息来确定。

解决方法之一是将产生式(1)中的词法单元 **id** 改成 **procid**，并使用一个更加复杂的词法分析器。该词法分析器在识别到一个过程名字的词素时返回词法单元名 **procid**。这就要求词法分析器在返回一个词法单元之前先查询符号表。

如果我们做了这样的修改，那么在处理 p(i, j)的时候，语法分析器要么进入格局

栈	输入
… **procid** (**id**	, **id**) …

要么进入前面描述的格局。在前一种情况下，我们选择产生式(5)进行归约；在后一种情况下，则选择产生式(7)进行归约。请注意，在这个例子里，栈顶之下的第三个符号决定了应该执行什么归约，虽然它本身并没有被归约。移入－归约的语法分析技术可以使用栈中离栈顶很远的信息来引导语法分析过程。 □

4.5.5 4.5 节的练习

练习 4.5.1：对于练习 4.2.2(a)中的文法 $S \rightarrow 0\ S\ 1 \mid 0\ 1$，指出下面各个最右句型的句柄：

1) 000111

2) 00S11

练习 4.5.2：对于练习 4.2.1 的文法 $S \rightarrow S\ S\ +\ |\ S\ S\ *\ |\ a$ 和下面各个最右句型，重复练习 4.5.1。

1) $SSS+a*+$

2) $SS+a*a+$

3) $aaa*a++$

练习 4.5.3：对于下面的输入符号串和文法，说明相应的自底向上语法分析过程。

1) 练习 4.5.1 的文法的串 000111。

2) 练习 4.5.2 的文法的串 $aaa*a++$。

4.6 LR 语法分析技术介绍：简单 LR 技术

目前最流行的自底向上语法分析器都基于所谓的 LR(k)语法分析的概念。其中，“L”表示对输入进行从左到右的扫描，“R”表示反向构造出一个最右推导序列，而 k 表示在做出语法分析决定时向前看 k 个输入符号。$k=0$ 和 $k=1$ 这两种情况具有实践意义，因此这里我们将只考虑 $k \leqslant 1$ 的情况。当省略(k)时，我们假设 $k=1$。

本节将介绍 LR 语法分析的基本概念，同时还将介绍最简单的构造移入－归约语法分析器的方法。这个方法称为“简单 LR 技术”（或简称为 SLR）。虽然 LR 语法分析器本身是使用语法分析器自动生成工具构造得到的，但对基本概念有所了解仍然是有益的。我们首先介绍“项”和“语法分析器状态”的概念，一个 LR 语法分析器生成工具给出的诊断信息通常会包含语法分析器状态。我们可以使用这些状态分离出语法分析冲突的源头。

4.7 节将介绍两个更加复杂的方法——规范 LR 和 LALR。它们被用于大多数的 LR 语法分析器中。

4.6.1 为什么使用 LR 语法分析器

LR 语法分析器是表格驱动的，在这一点上它和 4.4.4 节中提到的非递归 LL 语法分析器很相似。如果我们可以使用本节和下一节中的某个方法为一个文法构造出语法分析表，那么这个文法就称为 LR 文法(LR grammar)。直观地讲，只要存在这样一个从左到右扫描的移入－归约语法分析器，它总是能够在某文法的最右句型的句柄出现在栈顶时识别出这个句柄，那么这个文法就是 LR 的。

LR 语法分析技术很有吸引力，原因如下：

- 对于几乎所有的程序设计语言构造，只要能够写出该构造的上下文无关文法，就能够构造出识别该构造的 LR 语法分析器。确实存在非 LR 的上下文无关文法，但一般来说，常见的程序设计语言构造都可以避免使用这样的文法。
- LR 语法分析方法是已知的最通用的无回溯移入－归约分析技术，并且它的实现可以和其他更原始的移入－归约方法(见参考文献)一样高效。
- 一个 LR 语法分析器可以在对输入进行从左到右扫描时尽可能早地检测到错误。
- 可以使用 LR 方法进行语法分析的文法类是可以使用预测方法或 LL 方法进行语法分析的文法类的真超集。一个文法是 LR(k)的条件是当我们在一个最右句型中看到某个产生式的右部时，我们再向前看 k 个符号就可以决定是否使用这个产生式进行归约。这个要求比 LL(k)文法的要求宽松很多。对于 LL(k)文法，我们在决定是否使用某个产生式时，只能向前看该产生式右部推导出的串的前 k 个符号。因此，LR 文法能够比 LL 文法描述

更多的语言就一点也不奇怪了。

LR 方法的主要缺点是为一个典型的程序设计语言文法手工构造 LR 分析器的工作量非常大。我们需要一个特殊的工具，即一个 LR 语法分析器生成工具。幸运的是，有很多这样的生成工具可用，我们将在 4.9 节讨论其中最常用的工具 Yacc。这种生成工具将一个上下文无关文法作为输入，自动生成一个该文法的语法分析器。如果该文法含有二义性的构造，或者含有其他难以在从左到右扫描时进行语法分析的构造，那么语法分析器生成工具将对这些构造进行定位，并给出详细的诊断消息。

4.6.2 项和 LR(0) 自动机

一个移入 - 归约语法分析器怎么知道何时进行移入、何时进行归约呢？比如，当图 4-28 中栈的内容为 $ T 而下一个输入符号是 * 时，语法分析器是怎么知道位于栈顶的 T 不是句柄，因此正确的动作是移入而不是将 T 归约到 E 呢？

一个 LR 语法分析器通过维护一些状态，用这些状态来表明我们在语法分析过程中所处的位置，从而做出移入 - 归约决定。这些状态代表了“项”（item）的集合。一个文法 G 的一个 LR(0) 项（简称为项）是 G 的一个产生式再加上一个位于它的体中某处的点。因此，产生式 $A \rightarrow XYZ$ 产生了四个项：

$$A \rightarrow \cdot XYZ$$
$$A \rightarrow X \cdot YZ$$
$$A \rightarrow XY \cdot Z$$
$$A \rightarrow XYZ \cdot$$

产生式 $A \rightarrow \epsilon$ 只生成一个项 $A \rightarrow \cdot$。

项集的表示

一个生成自底向上语法分析器的生成工具可能需要便利地表示项和项集。请注意，一个项可以表示为一对整数，第一个整数是基础文法的产生式编号，第二个整数是点的位置。项集可以用这些数对的列表来表示。然而，如我们将看到的，需要用到的项集通常包含“闭包”项，这些项的点位于产生式体的开始处。这些项总是可以根据项集中的其他项重新构造出来，因此我们不必将它们包含在这个列表中。

直观地讲，项指明了在语法分析过程中的给定点上，我们已经看到了一个产生式的哪些部分。比如，项 $A \rightarrow \cdot XYZ$ 表明我们希望接下来在输入中看到一个从 XYZ 推导得到的串。项 $A \rightarrow X \cdot YZ$ 说明我们刚刚在输入中看到了一个可以由 X 推导得到的串，并且我们希望接下来看到一个能从 YZ 推导得到的串。项 $A \rightarrow XYZ \cdot$ 表示我们已经看到了产生式体 XYZ，已经是时候把 XYZ 归约为 A 了。

一个称为规范 LR(0) 项集族（canonical LR(0) collection）的一组项集提供了构建一个确定有穷自动机的基础。该自动机可用于做出语法分析决定。这样的有穷自动机称为 LR(0) 自动机[⊖]。更明确地说，这个 LR(0) 自动机的每个状态代表了规范 LR(0) 项集族中的一个项集。表达式文法 (4.1) 的对应的自动机显示在图 4-31 中。我们将把它用做讨论规范 LR(0) 项集族的连续使用的例子。

为了构造一个文法的规范 LR(0) 项集族，我们定义了一个增广文法（augmented grammar）和

⊖ 从技术上讲，根据 3.6.4 节的定义，这个自动机并不是确定自动机，因为我们没有对应于空项集的死状态。结果是有一些状态 - 输入对没有后继状态。

两个函数：CLOSURE 和 GOTO。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 中加上新开始符号 S' 和产生式 $S'\rightarrow S$ 而得到的文法。引入这个新的开始产生式的目的是告诉语法分析器何时应该停止语法分析并宣称接受输入符号串。也就是说，当且仅当语法分析器要使用规则 $S'\rightarrow S$ 进行归约时，输入符号串被接受。

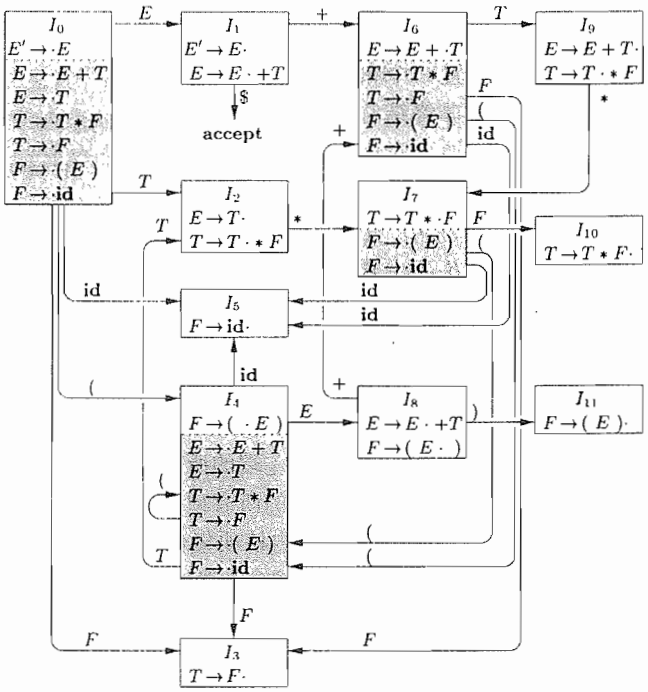

图 4-31　表达式文法(4.1)的 LR(0)自动机

项集的闭包

如果 I 是文法 G 的一个项集，那么 CLOSURE(I) 就是根据下面的两个规则从 I 构造得到的项集：

1）一开始，将 I 中的各个项加入到 CLOSURE(I) 中。

2）如果 $A\rightarrow\alpha\cdot B\beta$ 在 CLOSURE(I) 中，$B\rightarrow\gamma$ 是一个产生式，并且项 $B\rightarrow\cdot\gamma$ 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。

直观地讲，CLOSURE(I) 中的项 $A\rightarrow\alpha\cdot B\beta$ 表明在语法分析过程的某点上，我们认为接下来可能会在输入中看到一个能够从 $B\beta$ 推导得到的子串。这个可从 $B\beta$ 推导得到的子串的某个前缀可以从 B 推导得到，而推导时必然要应用某个 B 产生式。因此我们加入了各个 B 产生式对应的项，也就是说，如果 $B\rightarrow\gamma$ 是一个产生式，那么我们把 $B\rightarrow\cdot\gamma$ 加入到 CLOSURE(I) 中。

例 4.40　考虑增广的表达式文法：

$$E'\rightarrow E$$
$$E\rightarrow E+T\mid T$$
$$T\rightarrow T*F\mid F$$
$$F\rightarrow(E)\mid \mathbf{id}$$

如果 I 是由一个项组成的项集$\{[E'\rightarrow .E]\}$，那么 CLOSURE(I)包含了图 4-31 中的项集 I_0。

下面说明一下如何计算这个闭包。根据规则(1)，$E'\rightarrow \cdot E$ 被放到 CLOSURE(I)中。因为点的右边有一个 E，我们加入如下的 E 产生式，点位于产生式体的左端：$E\rightarrow \cdot E+T$ 和 $E\rightarrow \cdot T$。现在，后一个项中有一个 T 在点的右边，因此我们加入 $T\rightarrow \cdot T*F$ 和 $T\rightarrow \cdot F$。接下来，位于点右边的 F 令我们加入 $F\rightarrow \cdot (E)$ 和 $F\rightarrow \cdot$ **id**，然后就不再需要加入任何新的项。 □

闭包可以按照图 4-32 中的方法计算。实现函数 *closure* 的一个便利方法是设置一个布尔数组 *added*，该数组的下标是 G 的非终结符号。当我们为各个 B 产生式 $B\rightarrow\gamma$ 加入对应的项 $B\rightarrow \cdot\gamma$ 时，*added*[B]被设置为 **true**。

```
SetOfItems CLOSURE(I) {
    J = I;
    repeat
        for (J中的每个项 A → α·Bβ)
            for (G的每个产生式 B → γ)
                if (项 B → ·γ 不在J中)
                    将 B → ·γ 加入J中;
    until 在某一轮中没有新的项被加入到J中;
    return J;
}
```

图 4-32 CLOSURE 的计算

请注意，如果点在最左端的某个 B 产生式被加入到 I 的闭包中，那么所有 B 产生式都会被加入到这个闭包中。因此在某些情况下，不需要真的将那些被 CLOSURE 函数加入到 I 中的项 $B\rightarrow \cdot\gamma$ 列出来，只需要列出这些被加入的产生式的左部非终结符号就足够了。我们将感兴趣的各个项分为如下两类：

1）内核项：包括初始项 $S'\rightarrow \cdot S$ 以及点不在最左端的所有项。

2）非内核项：除了 $S'\rightarrow \cdot S$ 之外的点在最左端的所有项。

不仅如此，我们感兴趣的每一个项集都是某个内核项集合的闭包，当然，在求闭包时加入的项不可能是内核项。因此，如果我们抛弃所有非内核项，就可以用很少的内存来表示真正感兴趣的项的集合，因为我们已知这些非内核项可以通过闭包运算重新生成。在图 4-31 中，非内核项位于表示状态的方框的阴影部分中。

GOTO 函数

第二个有用的函数是 GOTO(I, X)，其中 I 是一个项集而 X 是一个文法符号。GOTO(I, X)被定义为 I 中所有形如 $[A\rightarrow\alpha\cdot X\beta]$ 的项所对应的项 $[A\rightarrow\alpha X\cdot\beta]$ 的集合的闭包。直观地讲，GOTO 函数用于定义一个文法的 LR(0)自动机中的转换。这个自动机的状态对应于项集，而 GOTO(I, X)描述了当输入为 X 时离开状态 I 的转换。

例 4.41 如果 I 是两个项的集合$\{[E'\rightarrow E\cdot\,],\ [E\rightarrow E\cdot\,+T]\}$，那么 GOTO($I$, +)包含如下项：

$$E\rightarrow E+\cdot T$$
$$T\rightarrow \cdot T*F$$
$$T\rightarrow \cdot F$$
$$F\rightarrow \cdot (E)$$
$$F\rightarrow \cdot \textbf{id}$$

我们查找 I 中点的右边紧跟 + 的项，就可以计算得到 GOTO(I, +)。$E'\rightarrow E\cdot$ 不是这样的项，但 $E\rightarrow E\cdot +T$ 是这样的项。我们将点移过 + 号得到 $E\rightarrow E+\cdot T$，然后求出这个单元素集合的闭包。 □

现在我们可以给出构造一个增广文法 G'的规范 LR(0)项集族 C 的算法。这个算法如图4-33所示。

例 4.42 文法(4.1)的规范 LR(0)项集族和 GOTO 函数如图 4-31 所示。其中,GOTO 函数用图中的转换表示。 □

```
void items(G') {
    C = {CLOSURE({[S' → ·S]})};
    repeat
        for (C中的每个项集I)
            for (每个文法符号X)
                if (GOTO(I, X)非空且不在C中)
                    将GOTO(I, X)加入C中;
    until 在某一轮中没有新的项集被加入到C中;
}
```

图 4-33 规范 LR(0)项集族的计算

LR(0)自动机的用法

"简单 LR 语法分析技术"(即 SLR 分析技术)的中心思想是根据文法构造出 LR(0)自动机。这个自动机的状态是规范 LR(0)项集族中的元素,而它的转换由 GOTO 函数给出。表达式文法(4.1)的 LR(0)自动机已经在前面的图 4-31 中显示过了。

这个 LR(0)自动机的开始状态是 CLOSURE({[$S'\rightarrow\cdot S$]}),其中 S'是增广文法的开始符号。所有的状态都是接受状态。我们说的"状态 j"指的是对应于项集 I_j 的状态。

LR(0)自动机是如何帮助做出移入 - 归约决定的呢?移入 - 规约决定可以按照如下方式做出。假设文法符号串 γ 使 LR(0)自动机从开始状态 0 运行到某个状态 j。那么如果下一个输入符号为 a 且状态 j 有一个在 a 上的转换,就移入 a。否则我们就选择归约动作。状态 j 的项将告诉我们使用哪个产生式进行归约。

将在 4.6.3 节中介绍的 LR 语法分析算法用它的栈来跟踪状态及文法符号。实际上,文法符号可以从相应状态中获取,因此它的栈只保存状态。下面的例子将展示如何使用一个 LR(0)自动机和一个状态栈来做出移入 - 归约语法分析决定。

例 4.43 图 4-34 给出了一个使用图 4-31 中的 LR(0)自动机的移入 - 归约语法分析器在分析 **id** * **id**时采取的动作。我们使用一个栈来保存状态。为清晰起见,栈中状态所对应的文法符号显示在"符号"列中。在第 1 行,栈中存放了自动机的开始状态 0,相应的符号是栈底标记 $。

行号	栈	符号	输入	动作
(1)	0	$	id * id $	移入到 5
(2)	0 5	$ id	* id $	按照$F \rightarrow$ id 归约
(3)	0 3	$ F	* id $	按照$T \rightarrow F$归约
(4)	0 2	$ T	* id $	移入到 7
(5)	0 2 7	$ T *	id $	移入到 5
(6)	0 2 7 5	$ T * id	$	按照$F \rightarrow$ id归约
(7)	0 2 7 10	$ T * F	$	按照$T \rightarrow T * F$归约
(8)	0 2	$ T	$	按照$E \rightarrow T$ 归约
(9)	0 1	$ E	$	接受

图 4-34 **id** * **id** 的语法分析

下一个输入符号是 **id**,而状态 0 在 **id** 上有一个到达状态 5 的转换。因此我们选择移入。在第 2 行,状态 5(符号 **id**)已经被压入到栈中。从状态 5 出发没有输入 * 上的转换,因此我们选择归约。根据状态 5 中的项[$F\rightarrow$**id** ·],这次归约应用产生式 $F\rightarrow$**id**。

如果栈中保存的是文法符号,那么归约就是通过将相应产生式的体(在第 2 行中,产生式的体是 **id**)弹出栈并将产生式头(在这个例子中是 F)压入栈中来实现的。现在栈中保存的是状态,我们弹出和符号 **id** 对应的状态 5,使得状态 0 成为栈顶。然后我们寻找一个 F(即该产生式的头部)上的转换。在图 4-31 中,状态 0 有一个 F 上的到达状态 3 的转换,因此我们压入状态 3。这个状态对应的符号是 F,见第 3 行。

我们看另一个例子,考虑第 5 行,状态 7(符号 *)位于栈顶。这个状态有一个 **id** 上的到达状态 5 的转换,因此我们将状态 5(符号 **id**)压入栈中。状态 5 没有转换,因此我们按照 $F\rightarrow$**id** 进行归约。当我们弹出对应于产生式体 **id** 的状态 5 后,状态 7 到达栈顶。因为状态 7 有一个 F 上的转换到达状态 10,我们压入状态 10(符号 F)。 □

4.6.3 LR 语法分析算法

图 4-35 中显示了一个 LR 语法分析器的示意图。它由一个输入、一个输出、一个栈、一个驱动程序和一个语法分析表组成。这个分析表包括两个部分(ACTION 和 GOTO)。所有 LR 语法分析器的驱动程序都是相同的,而语法分析表是随语法分析器的不同而变化的。语法分析器从输入缓冲区逐个读入符号。当一个移入 - 归约语法分析器移入一个符号时,LR 语法分析器移入的是一个对应的状态。每个状态都是对栈中该状态之下的内容所含信息的摘要。

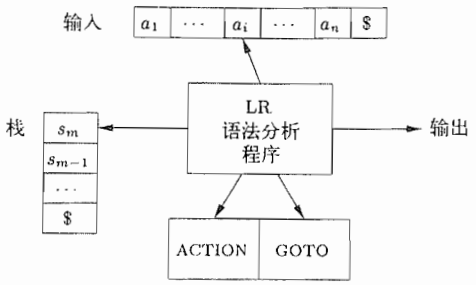

图 4-35 一个 LR 语法分析器的模型

分析器的栈存放了一个状态序列 $s_0 s_1 \cdots s_m$,其中 s_m 位于栈顶。在 SLR 方法中,栈中保存的是 LR(0)自动机中的状态,规范 LR 和 LALR 方法和 SLR 方法类似。根据构造方法,每个状态都有一个对应的文法符号。回顾一下,各个状态都和某个项集对应,并且有一个从状态 i 到状态 j 的转换当且仅当 $\text{GOTO}(I_i, X) = I_j$。所有到达状态 j 的转换一定对应于同一个文法符号 X。因此,除了开始状态 0 之外,每个状态都和唯一的文法符号相关联[⊖]。

LR 语法分析表的结构

语法分析表由两个部分组成:一个语法分析动作函数 ACTION 和一个转换函数 GOTO。

1) ACTION 函数有两个参数:一个是状态 i,另一个是终结符号 a(或者是输入结束标记 $)。ACTION[$i$, a]的取值可以有下列四种形式:

① 移入 j,其中 j 是一个状态。语法分析器采取的动作是把输入符号 a 高效地移入栈中,但是使用状态 j 来代表 a。

② 归约 $A \to \beta$。语法分析器的动作是把栈顶的 β 高效地归约为产生式头 A。

③ 接受。语法分析器接受输入并完成语法分析过程。

④ 报错。语法分析器在它的输入中发现了一个错误并执行某个纠正动作。我们将在 4.8.3 节和 4.9.4 节中进一步讨论这样的错误恢复例程是如何工作的。

2) 我们把定义在项集上的 GOTO 函数扩展为定义在状态集上的函数:如果 $\text{GOTO}[I_i, A] = I_j$,那么 GOTO 也把状态 i 和一个非终结符号 A 映射到状态 j。

LR 语法分析器的格局

描述 LR 语法分析器的行为时,我们需要一个能够表示 LR 语法分析器的完整状态的方法。语法分析器的完整状态包括:它的栈和余下的输入。LR 语法分析器的*格局*(configuration)是一个形如:

$$(s_0 s_1 \cdots s_m,\ a_i a_{i+1} \cdots a_n\ \$)$$

的对。其中,第一个分量是栈中的内容(右侧是栈顶),第二个分量是余下的输入。这个格局表示了如下的最右句型:

$$X_1 X_2 \cdots X_m a_i a_{i+1} \cdots a_n$$

它表示最右句型的方法本质上和一个移入 - 归约语法分析器的表示方法相同。唯一的不同之处

⊖ 其逆命题不一定成立。也就是说,多个状态可能对应于同一个文法符号。例如,图 4-31 中的 LR(0)自动机的状态 1 和 8,进入它们的都是 E 上的转换;而对于状态 2 和 9,它们都是通过 T 上的转换进入。

在于栈中存放的是状态而不是文法符号，从这些状态能够复原出相应的文法符号。也就是说，X_i 是状态 s_i 所代表的文法符号。请注意，s_0（即分析器的开始状态）不代表任何文法符号，它只是作为栈底标记，同时也在语法分析过程中担负了重要的角色。

LR 语法分析器的行为

语法分析器根据上面的格局决定下一个动作时，首先读入当前输入符号 a_i 和栈顶的状态 s_m，然后在分析动作表中查询条目 ACTOIN[s_m, a_i]。对于前面提到的四种动作，每个动作结束之后的格局如下：

1）如果 ACTION[s_m, a_i] = 移入 s，那么语法分析器执行一次移入动作；它将下一个状态 s 移入栈中，进入格局

$$(s_0 s_1 \cdots s_m s,\ a_{i+1} \cdots a_n \$)$$

符号 a_i 不需要存放在栈中，因为在需要时（在实践中从不需要 a_i）可以根据 s 恢复出 a_i。现在，当前的输入符号是 a_{i+1}。

2）如果 ACTION[s_m, a_i] = 规约 $A \rightarrow \beta$，那么语法分析器执行一次归约动作，进入格局

$$(s_0 s_1 \cdots s_{m-r} s,\ a_i a_{i+1} \cdots a_n \$)$$

其中，r 是 β 的长度，且 s = GOTO[s_{m-r}, A]。在这里，语法分析器首先将 r 个状态符号弹出栈，使状态 s_{m-r} 位于栈顶。然后，语法分析器将 s（即条目 GOTO[s_{m-r}, A] 的值）压入栈中。在一个归约动作中，当前的输入符号不会改变。对于我们将构造的 LR 语法分析器，对应于被弹出栈的状态的文法符号序列 $X_{m-r+1} \cdots X_m$ 总是等于 β，即归约使用的产生式的右部。

在一次归约动作之后，LR 语法分析器将执行和归约所用产生式关联的语义动作，生成相应的输出。我们暂时假设输出的内容仅仅包括打印出归约产生式。

3）如果 ACTION[s_m, a_i] = 接受，那么语法分析过程完成。

4）如果 ACTION[s_m, a_i] = 报错，则说明语法分析器发现了一个语法错误，并调用一个错误恢复例程。

LR 语法分析算法总结如下。所有的 LR 语法分析器都按照这个方式执行，两个 LR 语法分析器之间的唯一区别是它们的语法分析表的 ACTION 表项和 GOTO 表项中包含的信息不同。

算法 4.44 LR 语法分析算法。

输入： 一个输入串 w 和一个 LR 语法分析表，这个表描述了文法 G 的 ACTION 函数和 GOTO 函数。

输出： 如果 w 在 $L(G)$ 中，则输出 w 的自底向上语法分析过程中的归约步骤；否则给出一个错误指示。

方法： 最初，语法分析器栈中的内容为初始状态 s_0，输入缓冲区中的内容为 $w\$$。然后，语法分析器执行图 4-36 中的程序。 □

例 4.45 图 4-37 显示了表达式文法(4.1)的一个 LR 语法分析表中的 ACTION 和 GOTO 函数。下面再次给出文法(4.1)，并对它们的产生式进行编号：

(1) $E \rightarrow E + T$　　(4) $T \rightarrow F$

(2) $E \rightarrow T$　　(5) $F \rightarrow (E)$

(3) $T \rightarrow T * F$　　(6) $F \rightarrow$ **id**

各种动作在此图中的编码方法如下：

1）si 表示移入并将状态 i 压栈。

2）rj 表示按照编号为 j 的产生式进行归约。

```
令 a 为w$的第一个符号;
while(1) { /* 永远重复 */
    令 s 是栈顶的状态;
    if ( ACTION[s, a] = 移入 t ) {
        将 t 压入栈中;
        令 a 为下一个输入符号;
    } else if ( ACTION[s, a] = 归约 A → β ) {
        从栈中弹出|β|个符号;
        令 t 为当前的栈顶状态;
        将 GOTO[t, A]压入栈中;
        输出产生式 A → β;
    } else if ( ACTION[s, a] = 接受) break; /* 语法分析完成 */
    else 调用错误恢复例程;
}
```

图 4-36 LR 语法分析程序

3）acc 表示接受。

4）空白表示报错。

请注意，对于终结符号 a，GOTO[s, a]的值在 ACTION 表项中给出，这个值和在输入 a 上对应于状态 s 的移入动作一起给出。GOTO 条目给出了对应于非终结符号 A 的 GOTO[s, A]的值。我们还没有解释图 4-37 的表中各个条目是如何得到的，但很快就会来处理这个问题。

状态	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

图 4-37 表达式文法的语法分析表

在处理输入 **id** * **id** + **id** 时，栈和输入内容的序列显示在图 4-38 中。为清晰起见，图中还显示了与栈中状态对应的文法符号的序列。比如，在第 1 行中，LR 语法分析器位于状态 0 上。这是初始状态，没有对应的文法符号，而第一个输入符号是 **id**。图 4-37 中的动作部分第 0 行、id 列中的动作是 s5，表示应该移入，将状态 5 压栈。在第 2 行，状态符号 5 被压入到栈中，而 **id** 从输入中被删除。

	栈	符号	输入	动作
(1)	0		**id** * **id** + **id** $	移入
(2)	0 5	**id**	* **id** + **id** $	根据 $F \to$ **id** 归约
(3)	0 3	F	* **id** + **id** $	根据 $T \to F$ 归约
(4)	0 2	T	* **id** + **id** $	移入
(5)	0 2 7	T *	**id** + **id** $	移入
(6)	0 2 7 5	T * **id**	+ **id** $	根据$F \to$ **id** 归约
(7)	0 2 7 10	$T * F$	+ **id** $	根据$T \to T * F$归约
(8)	0 2	T	+ **id** $	根据$E \to T$ 归约
(9)	0 1	E	+ **id** $	移入
(10)	0 1 6	E +	**id** $	移入
(11)	0 1 6 5	E + **id**	$	根据 $F \to$ **id** 归约
(12)	0 1 6 3	$E + F$	$	根据 $T \to F$ 归约
(13)	0 1 6 9	$E + T$	$	根据 $E \to E + T$归约
(14)	0 1	E	$	接受

图 4-38 一个 LR 语法分析器处理输入 **id** * **id** + **id** 的各个步骤

然后，*变成了当前的输入符号，而状态5在输入为*时的动作是根据产生式 $F \rightarrow \mathbf{id}$ 进行归约。一个状态符号被弹出栈。然后，状态0成为栈顶。因为状态0对于F的GOTO值是3，因此状态3被压到栈中。现在我们得到第3行中的格局。下面的各个动作的执行方式与此类似。 □

4.6.4 构造SLR语法分析表

构造语法分析表的SLR构造方法是研究LR语法分析技术的很好的起点。我们把使用这种方法构造得到的语法分析表称为SLR语法分析表，并把使用SLR语法分析表的LR语法分析器称为SLR语法分析器。另外两种SLR方法通过向前看信息来增强分析能力。

SLR方法以4.5节介绍的LR(0)项和LR(0)自动机为基础。也就是说，给定一个文法 G，我们通过添加新的开始符号 S' 得到增广文法 G'。我们根据 G' 构造出 G' 的规范项集族 C 以及GOTO函数。

然后，使用下面的算法就可以构造出这个语法分析表中的ACTION和GOTO条目。它要求我们知道输入文法的每个非终结符号 A 的FOLLOW(A)(见4.4节)。

算法4.46 构造一个SLR语法分析表。

输入：一个增广文法 G'。

输出：G' 的SLR语法分析表函数ACTION和GOTO。

方法：

1) 构造 G' 的规范LR(0)项集族 $C = \{I_0, I_1, \cdots, I_n\}$。

2) 根据 I_i 构造得到状态 i。状态 i 的语法分析动作按照下面的方法决定：

① 如果[$A \rightarrow \alpha \cdot a\beta$]在 I_i 中并且GOTO(I_i, a) = I_j，那么将ACTION[i, a]设置为“移入 j”。这里 a 必须是一个终结符号。

② 如果[$A \rightarrow \alpha \cdot$]在 I_i 中，那么对于FOLLOW(A)中的所有 a，将ACTION[i, a]设置为“归约 $A \rightarrow \alpha$”。这里 A 不等于 S'。

③ 如果[$S' \rightarrow S \cdot$]在 I_i 中，那么将ACTION[i, $\$$]设置为“接受”。

如果根据上面的规则生成了任何冲突动作，我们就说这个文法不是SLR(1)的。在这种情况下，这个算法无法生成一个语法分析器。

3) 状态 i 对于各个非终结符号 A 的GOTO转换使用下面的规则构造得到：如果GOTO(I_i, A) = I_j，那么GOTO[i, A] = j。

4) 规则(2)和(3)没有定义的所有条目都设置为“报错”。

5) 语法分析器的初始状态就是根据[$S' \rightarrow \cdot S$]所在项集构造得到的状态。 □

由算法4.46得到的由ACTION函数和GOTO函数组成的语法分析表被称为文法 G 的SLR(1)分析表。使用 G 的SLR(1)分析表的LR语法分析器称为 G 的SLR(1)语法分析器。一个具有SLR(1)语法分析表的文法被称为是SLR(1)的。我们常常省略“SLR”后面的“(1)”，因为我们不会在这里处理向前看多个符号的语法分析器。

例4.47 让我们为增广表达式文法构造SLR分析表。这个文法的规范LR(0)项集族如图4-31所示。首先考虑项集 I_0：

$$
\begin{aligned}
&E' \rightarrow \cdot E\\
&E \rightarrow \cdot E + T\\
&E \rightarrow \cdot T\\
&T \rightarrow \cdot T * F\\
&T \rightarrow \cdot F
\end{aligned}
$$

$$F \rightarrow \cdot (\ E \ )$$
$$F \rightarrow \cdot \ \mathbf{id}$$

其中的项 $F \rightarrow \cdot (\ E \ )$ 使得条目 ACTION[0, (] = 移入 4，项 $F \rightarrow \cdot$ **id** 使得条目 ACTION[0, **id**] = 移入 5。I_0 中的其他项没有生成动作。现在考虑 I_1：

$$E' \rightarrow E \cdot$$
$$E \rightarrow E \cdot \ + \ T$$

第一个项使得 ACTION[1, $] = 接受，第二个项使得 ACTION[1, +] = 移入 6。下一步考虑 I_2：

$$E \rightarrow T \cdot$$
$$T \rightarrow T \cdot \ * \ F$$

因为 FOLLOW(E) = {$, +,)}，第一个项使得

$$\text{ACTION}[2, \$] = \text{ACTION}[2, +] = \text{ACTION}[2,)] = \text{归约 } E \rightarrow T$$

第二个项使得 ACTION[2, *] = 移入 7。按照这个方式继续推导，我们就得到了图 4-37 所示的 ACTION 和 GOTO 表。在该图中，归约动作中的产生式编号和它们在原文法(4.1)中的出现顺序相同。也就是说，$E \rightarrow E + T$ 的编号为 1，$E \rightarrow T$ 的编号为 2，依此类推。 □

例 4.48 每个 SLR(1)文法都是无二义性的，但是存在很多不是 SLR(1)的无二义性文法。考虑包含下列产生式的文法：

$$\begin{aligned} S &\rightarrow L = R \mid R \\ L &\rightarrow * \ R \mid \mathbf{id} \\ R &\rightarrow L \end{aligned} \tag{4.49}$$

将 L 和 R 分别看作代表左值和右值的文法符号，将 * 看作是代表“左值所指向的内容”的运算符[⊖]。文法 4.49 对应的规范 LR(0)项集族显示在图 4-39 中。

I_0: $S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$
$R \rightarrow \cdot L$

I_1: $S' \rightarrow S \cdot$

I_2: $S \rightarrow L \cdot = R$
$R \rightarrow L \cdot$

I_3: $S \rightarrow R \cdot$

I_4: $L \rightarrow * \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_5: $L \rightarrow \mathbf{id} \cdot$

I_6: $S \rightarrow L = \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_7: $L \rightarrow * R \cdot$

I_8: $R \rightarrow L \cdot$

I_9: $S \rightarrow L = R \cdot$

图 4-39 文法(4.49)对应的规范 LR(0)项集族

⊖ 2.8.3 节介绍过，一个左值表示了一个内存位置，而右值是一个可以存放在某个位置上的值。

考虑项集 I_2。这个项集中的第一个项使得 ACTION[2, =]是“移入 6”。因为 FOLLOW(R)包含 =(考虑推导过程 $S \Rightarrow L = R \Rightarrow * R = R$ 即可知原因),第二个项将 ACTION[2, =]设置为“归约 $R \rightarrow L$”。因为在 ACTION[2, =]中既存在移入条目又存在归约条目,所以状态 2 在输入符号 = 上存在移入/归约冲突。

文法(4.49)不是二义性的。产生移入/归约冲突的原因是构造 SLR 分析器的方法功能不够强大,不能记住足够多的上下文信息。因此当它看到一个可归约为 L 的串时,不能确定语法分析器应该对输入 = 采取什么动作。接下来讨论的规范 LR 方法和 LALR 方法将可以成功地处理更大的文法类型,包括文法(4.49)。然而请注意,存在一些无二义性的文法使得每种 LR 语法分析器构造方法都会产生带有语法分析动作冲突的语法分析动作表。幸运的是,在处理程序设计语言时,一般都可以避免使用这样的文法。 □

4.6.5 可行前缀

为什么可以使用 LR(0)自动机来做出移入 - 归约决定?对于一个文法的移入 - 归约语法分析器,该文法的 LR(0)自动机可以刻画出可能出现在分析器栈中的文法符号串。栈中内容一定是某个最右句型的前缀。如果栈中的内容是 α 而余下的输入是 x,那么存在一个将 αx 归约到开始符号 S 的归约序列。用推导的方式表示就是 $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha x$。

然而,不是所有的最右句型的前缀都可以出现在栈中,因为语法分析器在移入时不能越过句柄。比如,假设

$$E \overset{*}{\underset{rm}{\Rightarrow}} F * \mathbf{id} \underset{rm}{\Rightarrow} (E) * \mathbf{id}$$

那么在语法分析的不同时刻,栈中存放的内容可以是(、(E 和(E),但不会是(E) *,因为(E)是句柄,语法分析器必须在移入 * 之前将它归约为 F。

可以出现在一个移入 - 归约语法分析器的栈中的最右句型前缀被称为可行前缀(viable prefix)。它们的定义如下:一个可行前缀是一个最右句型的前缀,并且它没有越过该最右句型的最右句柄的右端。根据这个定义,我们总是可以在一个可行前缀之后增加一些终结符号来得到一个最右句型。

SLR 分析技术基于 LR(0)自动机能够识别可行前缀这一事实。如果存在一个推导过程 $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha \beta_1 \beta_2 w$,我们就说项 $A \rightarrow \beta_1 . \beta_2$ 对于可行前缀 $\alpha\beta_1$ 有效。一般来说,一个项可以对多个可行前缀有效。

项 $A \rightarrow \beta_1 \cdot \beta_2$ 对 $\alpha\beta_1$ 有效的事实可以告诉我们很多信息。当我们在语法分析栈中发现 $\alpha\beta_1$ 时,这些信息可以帮助我们决定是进行归约还是移入。特别是,如果 $\beta_2 \neq \epsilon$,那么它告诉我们句柄还没有被全部移入到栈中,因此我们应该选择移入。如果 $\beta_2 = \epsilon$,那么看起来 $A \rightarrow \beta_1$ 就是句柄,我们应该按照这个产生式进行归约。当然,可能会有两个有效项要求我们对同一个可行前缀做不同的事情。有些这样的冲突可以通过查看下一个输入符号来解决,还有一些冲突可以通过 4.8 节中的方法来解决,但是我们不应该认为将 LR 方法应用于任意文法所产生的语法分析动作冲突都可以得到解决。

对于可能出现在 LR 语法分析栈中的各个可行前缀,我们可以很容易地计算出对应于这些可行前缀的有效项的集合。实际上,LR 语法分析理论的核心定理是:如果我们在某个文法的 LR(0)自动机中从初始状态开始沿着标号为某个可行前缀 γ 的路径到达一个状态,那么该状态对应的项集就是 γ 的有效项集。实质上,有效项集包含了所有能够从栈中收集到的有用信息。我们不会在这里证明这个定理,但我们将给出一个例子。

将项看作一个 NFA 的状态

如果将项本身看作状态，我们就可以构造出一个识别可行前缀的不确定有穷自动机 N。从 $A\rightarrow\alpha\cdot X\beta$ 到 $A\rightarrow\alpha X\cdot\beta$ 有一个标号为 X 的转换，并且从 $A\rightarrow\alpha\cdot B\beta$ 到 $B\rightarrow\cdot\gamma$ 有一个标号为 ϵ 的转换。那么项（N 的状态）的集合 I 的 CLOSURE(I) 恰恰就是 3.7.1 节中定义的一个 NFA 状态集合的 ϵ 闭包。由 NFA N 通过子集构造法可以得到一个 DFA。GOTO(I,X) 给出了这个 DFA 中状态 I 在符号 X 上的转换。从这个角度看，图 4-33 中的过程 items(G') 就是将子集构造方法应用于以项作为状态的 NFA N 并构造出 DFA 的过程。

例 4.50 让我们再次考虑增广表达式文法。该文法的项集和 GOTO 函数如图 4-31 所示。显然，串 $E+T*$ 是该文法的一个可行前缀。图 4-31 中的自动机在读入 $E+T*$ 之后将位于状态 7 上。状态 7 中包含了项

$$T\rightarrow T\ *\ \cdot F$$
$$F\rightarrow\cdot(\ E\ )$$
$$F\rightarrow\cdot\ \mathbf{id}$$

它们恰恰就是 $E+T*$ 的有效项。为了说明原因，考虑如下三个最右推导：

$$E'\underset{rm}{\Rightarrow}E\underset{rm}{\Rightarrow}E+T\underset{rm}{\Rightarrow}E+T*F\underset{rm}{\Rightarrow}E+T*(\ E\ )$$

$$E'\underset{rm}{\Rightarrow}E\underset{rm}{\Rightarrow}E+T\underset{rm}{\Rightarrow}E+T*F\underset{rm}{\Rightarrow}E+T*\mathbf{id}$$

第一个推导说明 $T\rightarrow T\ *\ \cdot\ F$ 是有效的，第二个推导说明 $F\rightarrow\cdot(\ E\ )$ 是有效的，第三个推导说明了 $F\rightarrow\cdot\ \mathbf{id}$ 是有效的。可以证明 $E+T*$ 没有其他的有效项，但我们并不会在这里证明这个事实。 □

4.6.6 4.6 节的练习

练习 4.6.1：描述下列文法的所有可行前缀：

1）练习 4.2.2(1) 的文法 $S\rightarrow 0\ S\ 1\mid 0\ 1$。

!2）练习 4.2.1 的文法 $S\rightarrow S\ S\ +\mid S\ S\ *\mid a$。

!3）练习 4.2.2(3) 的文法 $S\rightarrow S\ (\ S\ )\mid\epsilon$。

练习 4.6.2：为练习 4.2.1 中的（增广）文法构造 SLR 项集。计算这些项集的 GOTO 函数。给出这个文法的语法分析表。这个文法是 SLR 文法吗？

练习 4.6.3：利用练习 4.6.2 得到的语法分析表，给出处理输入 $aa*a+$ 时的各个动作。

练习 4.6.4：对于练习 4.2.2(1) ~ (7) 中的各个（增广）文法：

1）构造 SLR 项集和它们的 GOTO 函数。

2）指出你的项集中的所有动作冲突。

3）如果存在 SLR 语法分析表，构造出这个语法分析表。

练习 4.6.5：说明下面的文法

$$S\rightarrow A\ a\ A\ b\mid B\ b\ B\ a$$
$$A\rightarrow\epsilon$$
$$B\rightarrow\epsilon$$

是 LL(1) 的，但不是 SLR(1) 的。

练习 4.6.6：说明下面的文法

$$S \rightarrow S\ A \mid A$$
$$A \rightarrow a$$

是 SLR(1)的，但不是 LL(1)的。

!! 练习 4.6.7：考虑按照下面方式定义的文法族 G_n：

$$S \rightarrow A_i\ b_i \qquad \text{其中 } 1 \leqslant i \leqslant n$$
$$A_i \rightarrow a_j\ A_i \mid a_j \qquad \text{其中 } 1 \leqslant i, j \leqslant n \text{ 且 } i \neq j$$

说明：

1) G_n 有 $2n^2 - n$ 个产生式。

2) G_n 有 $2^n + n^2 + n$ 个 LR(0)项集。

3) G_n 是 SLR(1)的。

关于 LR 语法分析器的大小，这个分析结果说明了什么？

! 练习 4.6.8：我们说单个项可以看作一个不确定有穷自动机的状态，而有效项的集合就是一个确定有穷自动机的状态（见 4.6.5 节中的“将项看作一个 NFA 的状态”部分）。对于练习 4.2.1 的文法 $S \rightarrow S\ S\ +\ \mid S\ S\ *\ \mid a$：

1) 根据“将项看作一个 NFA 的状态”部分中的规则，画出这个文法的有效项的转换图(NFA)。

2) 将子集构造算法（算法 3.20）应用于在(1)部分构造得到的 NFA。得到的 DFA 和这个文法的 LR(0)项集相比有什么关系？

!! 3) 说明在任何情况下，将子集构造算法应用于一个文法的有效项的 NFA 所得到的就是该文法的 LR(0)项集。

! 练习 4.6.9：下面是一个二义性文法：

$$S \rightarrow A\ S \mid b$$
$$A \rightarrow S\ A \mid a$$

构造出这个文法的规范 LR(0)项集族。如果我们试图为这个文法构造出一个 LR 语法分析表，必然会存在某些冲突动作。都有哪些冲突动作？假设我们使用这个语法分析表，并且在出现冲突时不确定地选择一个可能的动作。给出处理输入 *abab* 时的所有可能的动作序列。

4.7 更强大的 LR 语法分析器

在本节中，我们将扩展前面的 LR 语法分析技术，在输入中向前看一个符号。有两种不同的方法：

1)“规范 LR”方法，或直接称为“LR”方法。它充分地利用了向前看符号。这个方法使用了一个很大的项集，称为 LR(1)项集。

2)“向前看 LR”，或称为“LALR”方法。它基于 LR(0)项集族。和基于 LR(1)项的典型语法分析器相比，它的状态要少很多。通过向 LR(0)项中小心地引入向前看符号，我们使用 LALR 方法处理的文法比使用 SLR 方法时处理的文法更多，同时构造得到的语法分析表却不比 SLR 分析表大。在很多情况下，LALR 方法是最合适的选择。

在介绍了这两种方法之后，我们将在本节的结尾讨论如何在一个内存有限的环境中建立简洁的 LR 语法分析表。

4.7.1 规范 LR(1)项

现在我们将给出最通用的为文法构造 LR 语法分析表的技术。回顾一下，在 SLR 方法中，如

果项集 I_i 包含项$[A\to\alpha\cdot]$，且当前输入符号 a 在 FOLLOW(A)中，那么状态 i 就要按照 $A\to\alpha$ 进行归约。然而在某些情况下，当状态 i 出现在栈顶时，栈中的可行前缀是 $\beta\alpha$ 且在任何最右句型中 a 都不可能跟在 βA 之后，那么当输入为 a 时不应该按照 $A\to\alpha$ 进行归约。

例 4.51 让我们重新考虑例子 4.48，其中的状态 2 包含项 $R\to L\cdot$。这个项对应于上面讨论的 $A\to\alpha$，而和 a 对应的是 FOLLOW(R)中的符号 =。因此，SLR 语法分析器在下一个输入为 = 且状态为 2 时要求按照 $R\to L$ 进行归约(因为状态 2 中还包含项 $S\to L\cdot =R$，它同时还要求执行移入动作)。然而，例 4.48 的文法没有以 $R=\cdots$ 开头的最右句型。因此状态 2 只和可行前缀 L 对应，它实际上不应该执行从 L 到 R 的归约。 □

如果在状态中包含更多的信息，我们就可能排除掉一些这样的不正确的 $A\to\alpha$ 归约。在必要时，我们可以通过分裂某些状态，设法让 LR 语法分析器的每个状态精确地指明哪些输入符号可以跟在句柄 α 的后面，从而使 α 可能被归约成为 A。

将这个额外的信息加入状态中的方法是对项进行精化，使它包含第二个分量，这个分量的值为一个终结符号。项的一般形式变成了$[A\to\alpha\cdot\beta,\ a]$，其中 $A\to\alpha\beta$ 是一个产生式，而 a 是一个终结符号或右端结束标记 $。我们称这样的对象为 $LR(1)$ 项。其中的 1 指的是第二个分量的长度。第二个分量称为这个项的向前看符号[⊖]。在形如$[A\to\alpha\cdot\beta,\ a]$且 $\beta\neq\epsilon$ 的项中，向前看符号没有任何作用，但是一个形如$[A\to\alpha\cdot,\ a]$的项只有在下一个输入符号等于 a 时才要求按照 $A\to\alpha$ 进行归约。因此，只有当栈顶状态中包含一个 LR(1)项$[A\to\alpha\cdot,\ a]$，我们才会在输入为 a 时按照 $A\to\alpha$ 进行归约。这样的 a 的集合总是 FOLLOW(A)的子集，而且如例 4.51 所示，它很可能是一个真子集。

正式地讲，我们说 LR(1)项$[A\to\alpha\cdot\beta,\ a]$对于一个可行前缀 γ 有效的条件是存在一个推导 $S\underset{rm}{\overset{*}{\Rightarrow}}\delta Aw\underset{rm}{\Rightarrow}\delta\alpha\beta w$，其中

1) $\gamma=\delta\alpha$，且

2) 要么 a 是 w 的第一个符号，要么 w 为 ϵ 且 a 等于 $。

例 4.52 让我们考虑文法

$$S\to B\ B$$
$$B\to a\ B\mid b$$

该文法有一个最右推导 $S\underset{rm}{\overset{*}{\Rightarrow}}aaBab\underset{rm}{\Rightarrow}aaaBab$。在上面的定义中，令 $\delta=aa$，$A=B$，$w=ab$，$\alpha=a$ 且 $\beta=B$，我们可知项$[B\to\alpha\cdot B,\ a]$对于可行前缀 $\gamma=aaa$ 是有效的。另外还有一个最右推导 $S\underset{rm}{\overset{*}{\Rightarrow}}BaB\underset{rm}{\Rightarrow}BaaB$。根据这个推导，我们知道项$[B\to\alpha\cdot B,\ \$]$是可行前缀 Baa 的有效项。 □

4.7.2 构造 LR(1)项集

构造有效 LR(1)项集族的方法实质上和构造规范 LR(0)项集族的方法相同。我们只需要修改两个过程：CLOSURE 和 GOTO。

为了理解 CLOSURE 操作的新定义，特别是理解为什么 b 必须在 FIRST(βa)中，我们考虑对某些可行前缀 γ 有效的项集合中的一个形如$[A\to\alpha\cdot B\beta,a]$的项，那么必然存在一个最右推导 $S\underset{rm}{\overset{*}{\Rightarrow}}\delta Aax\underset{rm}{\Rightarrow}\delta\alpha B\beta ax$，其中 $\gamma=\delta\alpha$。假设 βax 推导出终结符号串 by，那么对于某个形如 $B\to\eta$ 的产生式，我们有推导 $S\underset{rm}{\overset{*}{\Rightarrow}}\gamma Bby\underset{rm}{\Rightarrow}\gamma\eta by$。因此，$[B\to\cdot\eta,\ b]$是 γ 的有效项。请注意，$b$ 可能是从 β 推导

⊖ 当然可以使用长度大于 1 的向前看符号串。但是这里我们不考虑这样的向前看符号串。

得到的第一个终结符号，也可能在 $\beta ax \underset{rm}{\Rightarrow} by$ 的推导过程中 β 推导出了 ϵ，因此 b 也可能是 a。总结这两种情况，我们说 b 可以是 FIRST(βax) 中的任意终结符号，其中 FIRST 是在 4.4 节中定义的函数。请注意，x 不可能包含 by 的第一个终结符号，因此 FIRST(βax) = FIRST(βa)。现在我们给出 LR(1) 项集的构造方法。

算法 4.53 LR(1) 项集族的构造方法。

输入： 一个增广文法 G'。

输出： LR(1) 项集族，其中的每个项集对文法 G' 的一个或多个可行前缀有效。

方法： 过程 CLOSURE 和 GOTO，以及用于构造项集的主例程 items 见图 4-40。 □

```
SetOfItems CLOSURE(I) {
        repeat
                for ( I 中的每个项 [A → α·Bβ, a] )
                        for ( G' 中的每个产生式 B → γ )
                                for ( FIRST(βa)中的每个终结符号 b )
                                        将 [B → ·γ, b] 加入到集合 I 中;
        until 不能向 I 中加入更多的项;
        return I;
}

SetOfItems GOTO(I, X) {
        将 J 初始化为空集;
        for ( I 中的每个项 [A → α·Xβ, a])
                将项 [A → αX·β, a]加入到集合 J 中;
        return CLOSURE(J);
}

void items(G') {
        将 C 初始化为{CLOSURE}({[S' → ·S, $]});
        repeat
                for ( C 中的每个项集 I )
                        for ( 每个文法符号 X )
                                if ( GOTO(I, X) 非空且不在 C 中)
                                        将 GOTO(I, X) 加入 C 中;
        until 不再有新的项集加入到 C 中;
}
```

图 4-40 为文法 G' 构造 LR(1) 项集族的算法

例 4.54 考虑下面的增广文法：

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow C\ C \\ C &\rightarrow c\ C \mid d \end{aligned} \tag{4.55}$$

我们首先计算 $\{[S' \rightarrow \cdot S,\ \$]\}$ 的闭包。在求闭包时，我们将项 $[S' \rightarrow \cdot S,\ \$]$ 和过程 CLOSURE 中的项 $[A \rightarrow \alpha \cdot B\beta,\ a]$ 相匹配。也就是说，$A = S'$，$\alpha = \epsilon$，$B = S$，$\beta = \epsilon$ 和 $a = \$$。函数 CLOSURE 告诉我们，对于每个产生式 $B \rightarrow \gamma$ 和 FIRST(βa) 中的终结符号 b，将项 $[B \rightarrow \cdot \gamma,\ b]$ 加入到闭包中。对于当前的文法，$B \rightarrow \gamma$ 就是 $S \rightarrow CC$，并且因为 β 是 ϵ 且 a 是 \$，b 只能是 \$。因此，我们增加 $[S \rightarrow \cdot CC,\ \$]$。

我们继续计算闭包，对于在 FIRST($C\ \$$) 中的 b，加入所有的项 $[C \rightarrow \cdot \gamma,\ b]$。也就是说，将 $[S \rightarrow \cdot CC,\ \$]$ 和 $[A \rightarrow \alpha \cdot B\beta,\ a]$ 相匹配，我们有 $A = S$，$\alpha = \epsilon$，$B = C$，$\beta = C$ 且 $a = \$$。因为 C 不会推导出空串，所以 FIRST($C\ \$$) = FIRST(C)。因为 FIRST(C) 包含终结符号 c 和 d，所以我们

加入项[$C\rightarrow\cdot cC$, c]、[$C\rightarrow\cdot cC$, d]、[$C\rightarrow\cdot d$, c]和[$C\rightarrow\cdot d$, d]。在这些项中，紧靠在点右边的都不是非终结符号，因此我们已经完成了第一个 LR(1)项集。这个初始项集是：

$$\begin{aligned} I_0:\ & S'\rightarrow\cdot S,\ \$\\ & S\rightarrow\cdot CC,\ \$\\ & C\rightarrow\cdot cC,\ c/d\\ & C\rightarrow\cdot d,\ c/d \end{aligned}$$

为表示方便，我们省略了方括号，并且使用[$C\rightarrow\cdot cC$, c/d]作为两个项[$C\rightarrow\cdot cC$, c]和[$C\rightarrow\cdot cC$, d]的缩写。

现在我们对不同的 X 值计算 GOTO(I_0, X)。对于 $X=S$，我们必须求[$S'\rightarrow S\cdot$, \$]的闭包。因为点在最右端，所以无法加入新的项。因此我们得到下一个项集

$$I_1:\ S'\rightarrow S\cdot,\ \$$$

对于 $X=C$，我们求[$S\rightarrow C\cdot C$, \$]闭包。我们以 \$ 作为第二个分量加入 C 产生式，之后不能再加入新的项，得到：

$$\begin{aligned} I_2:\ & S\rightarrow C\cdot C,\ \$\\ & C\rightarrow\cdot cC,\ \$\\ & C\rightarrow\cdot d,\ \$ \end{aligned}$$

接下来，令 $X=c$。我们必须求{[$C\rightarrow c\cdot C$, c/d]}的闭包。我们将 c/d 作为第二个分量加入 C 产生式，得到：

$$\begin{aligned} I_3:\ & C\rightarrow c\cdot C,\ c/d\\ & C\rightarrow\cdot cC,\ c/d\\ & C\rightarrow\cdot d,\ c/d \end{aligned}$$

最后，令 $X=d$，我们得到项集：

$$I_4:\ C\rightarrow d\cdot,\ c/d$$

我们已经完成了 I_0 上的 GOTO 函数。我们没有从 I_1 得到新的项集，但是 I_2 有相对于 C、c 和 d 的 GOTO 后继。对于 GOTO(I_2, C)，我们有

$$I_5:\ S\rightarrow CC\cdot,\ \$$$

它不需要进行闭包运算。为了计算 GOTO(I_2, c)，我们对{[$C\rightarrow c\cdot C$, \$]}求闭包，得到

$$\begin{aligned} I_6:\ & C\rightarrow c\cdot C,\ \$\\ & C\rightarrow\cdot cC,\ \$\\ & C\rightarrow\cdot d,\ \$ \end{aligned}$$

请注意，I_6 和 I_3 只在第二个分量上有所不同。我们会经常看到一个文法的多个 LR(1)项集具有相同的第一分量，但第二分量不同。当我们为同一个文法构造规范 LR(0)项集族时，每一个 LR(0)项集将和一个或多个 LR(1)项集的第一分量集合完全一致。我们将在讨论 LALR 语法分析技术的时候更加深入地讨论这个现象。

继续计算 I_2 的 GOTO 函数，GOTO(I_2, d)就是

$$I_7:\ C\rightarrow d\cdot,\ \$$$

现在转而处理 I_3，I_3 在 c 和 d 上的 GOTO 值分别是 I_3 和 I_4。GOTO(I_3, C)是

$$I_8:\ C\rightarrow cC\cdot,\ c/d$$

I_4 和 I_5 没有 GOTO 值，因为它们的项中的点都在最右端。I_6 在 c 和 d 上的 GOTO 值分别是 I_6 和 I_7，而 GOTO(I_6, C)是

$$I_9:\ C\rightarrow cC\cdot,\ \$$$

其余的各个项集都没有 GOTO 值，因此我们完成了所有项集的计算。图 4-41 显示了这 10 个项集和它们之间的 goto 关系。 □

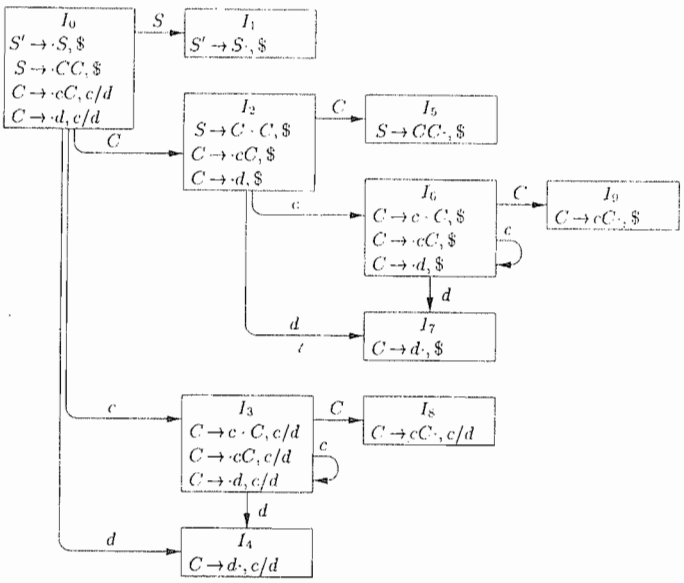

图 4-41 文法(4.55)的 GOTO 图

4.7.3 规范 LR(1)语法分析表

现在我们给出根据 LR(1)项集构造 LR(1)的 ACTION 和 GOTO 函数的规则。和前面一样，这些函数将用一个表来表示，只是表格条目中的值有所不同。

算法 4.56 规范 LR 语法分析表的构造。

输入：一个增广文法 G'。

输出：G'的规范 LR 语法分析表的函数 ACTION 和 GOTO。

方法：

1）构造 G'的 LR(1)项集族 $C' = \{I_0, I_1, \cdots, I_n\}$。

2）语法分析器的状态 i 根据 I_i 构造得到。状态 i 的语法分析动作按照下面的规则确定：

① 如果$[A \to \alpha \cdot a\beta, b]$在 I_i 中，并且 $\text{GOTO}(I_i, a) = I_j$，那么将 ACTION$[i, a]$设置为“移入 j”。这里 a 必须是一个终结符号。

② 如果$[A \to \alpha \cdot, a]$在 I_i 中且 $A \neq S'$，那么将 ACTION$[i, a]$设置为“规约 $A \to \alpha$”。

③ 如果$[S' \to S \cdot, \$]$在 I_i 中，那么将 ACTION$[i, \$]$设置为“接受”。

如果根据上述规则会产生任何冲突动作，我们就说这个文法不是 LR(1)的。在这种情况下，这个算法无法为该文法生成一个语法分析器。

3）状态 i 相对于各个非终结符号 A 的 goto 转换按照下面的规则构造得到：如果 $\text{GOTO}(I_i, A) = I_j$，那么 GOTO$[i, A] = j$。

4）所有没有按照规则(2)和(3)定义的分析表条目都设为“报错”。

5）语法分析器的初始状态是由包含$[S' \to \cdot S, \$]$的项集构造得到的状态。 □

由算法 4.56 生成的语法分析动作和 GOTO 函数组成的表称为规范 LR(1)语法分析表。使用这个表的 LR 语法分析器称为规范 LR(1)语法分析器。如果语法分析动作函数中不包含多重定义的条

目，那么给定的文法就称为 LR(1)文法。和前面一样，在大家都了解的情况下我们将省略“(1)”。

例 4.57 文法(4.55)的规范语法分析表如图 4-42 所示。产生式 1、2 和 3 分别是 $S\rightarrow CC$，$C\rightarrow cC$ 和 $C\rightarrow d$。□

状态	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

图 4-42 文法(4.55)的规范 LR 语法分析表

每个 SLR(1)文法都是 LR(1)文法。但是对于一个 SLR(1)文法而言，规范 LR(1)语法分析器的状态要比同一文法对应的 SLR 语法分析器的状态多。前一个例子中的文法是 SLR 的，它的 SLR 语法分析器有七个状态；相比之下，图 4-42 中有十个状态。

4.7.4 构造 LALR 语法分析表

现在我们介绍最后一种语法分析器构造方法，即 LALR(向前看-LR)技术。这个方法经常在实践中使用，因为用这种方法得到的分析表比规范 LR 分析表小很多，而且大部分常见的程序设计语言构造都可以方便地使用一个 LALR 文法表示。对于 SLR 文法，这一点也基本成立，只是仍然存在少量构造不能够方便地使用 SLR 技术来处理(例如，见例 4.48)。

我们对语法分析器的大小做一下比较。一个文法的 SLR 和 LALR 分析表总是具有相同数量的状态，对于像 C 这样的语言来说，通常有几百个状态。对于同样大小的语言，规范 LR 分析表通常有几千个状态。因此，构造 SLR 和 LALR 分析表要比构造规范 LR 分析表更容易，而且更经济。

为了介绍 LALR 技术，让我们再次考虑文法(4.55)。该文法的 LR(1)项集如图 4-41 所示。让我们查看两个看起来差不多的状态，比如 I_4 和 I_7。它们都只有一个项，其第一个分量都是 $C\rightarrow d\cdot$。在 I_4 中，向前看符号是 c 或 d；在 I_7 中，\$ 是唯一的向前看符号。

为了了解 I_4 和 I_7 在语法分析器中担负的不同角色，请注意这个文法生成了正则语言 $\mathbf{c}^*\mathbf{dc}^*\mathbf{d}$。当读入输入 $cc\cdots cdcc\cdots cd$ 的时候，语法分析器首先将第一组 c 以及跟在它们后面的 d 移入栈中。语法分析器在读入 d 之后进入状态 4。然后，当下一个输入符号是 c 或 d 时，语法分析器按照产生式 $C\rightarrow d$ 进行一次归约。要求 c 或 d 跟在后面是有道理的，因为它们可能是 $\mathbf{c}^*\mathbf{d}$ 中的串的开始符号。如果 \$ 跟在第一个 d 后面，我们就有形如 ccd 的输入，而它们不在这个语言中。如果 \$ 是下一个输入符号，状态 4 就会正确地报告一个错误。

语法分析器在读入第二个 d 之后进入状态 7。然后，语法分析器必须在输入中看到 \$，否则输入开头的符号串就不具有 $\mathbf{c}^*\mathbf{dc}^*\mathbf{d}$ 的形式。因此状态 7 应该在输入为 \$ 时按照 $C\rightarrow d$ 进行归约，而在输入为 c 或 d 的时候报告错误。

现在，我们将 I_4 和 I_7 替换为 I_{47}，即 I_4 和 I_7 的并集。这个项集包含了[$C\rightarrow d\cdot$，$c/d/\$$]所代表的三个项。原来在输入 d 上从 I_0、I_2、I_3 到达 I_4 或 I_7 的 goto 关系现在都到达 I_{47}。状态 47 在所有输入上的动作都是归约。这个经过修改的语法分析器行为在本质上和原分析器一样。虽然在有些情况下，原分析器会报告错误，而新分析器却将 d 归约为 C。比如，在处理 ccd 或 $cdcdc$ 这样的输入时就会出现这样的情况。新的分析器最终能够找到这个错误，实际上这个错误会在移入任何新的输入符号之前就被发现。

更一般地说，我们可以寻找具有相同核心(core)的 LR(1)项集，并将这些项集合并为一个项集。所谓项集的核心就是其第一分量的集合。比如在图 4-41 中，I_4 和 I_7 就是这样一对项集，它们的核心是{$C\rightarrow d\cdot$}。类似地，I_3 和 I_6 是另一对这样的项集，它们的核心是{$C\rightarrow c\cdot C$，$C\rightarrow\cdot cC$，$C\rightarrow\cdot d$}。另外，还有一对项集 I_8 和 I_9，它们的公共核心是{$C\rightarrow cC\cdot$}。请注意，一般而言，一个核

心就是当前正处理的文法的 LR(0)项集，一个 LR(1)文法可能产生多个具有相同核心的项集。

因为 GOTO(I, X)的核心只由 I 的核心决定，一组被合并的项集的 GOTO 目标也可以被合并。因此，当我们合并项集时可以相应地修改 GOTO 函数。动作函数也需要加以修改，以反映出被合并的所有项集的非报错动作。

假设我们有一个 LR(1)文法，也就是说，这个文法的 LR(1)项集没有产生语法分析动作冲突。如果我们将所有具有相同核心的状态替换为它们的并集，那么得到的并集有可能产生冲突。但是因为下面的原因，这种情况不大可能发生：假设在并集中有一个项$[A\to\alpha\cdot, a]$要求按照$A\to\alpha$进行归约，同时另一个项$[B\to\beta\cdot a\gamma, b]$要求进行移入，那么就会出现在向前看符号 a 上的冲突。此时必然存在某个被合并进来的项集中包含项$[A\to\alpha\cdot, a]$，同时因为所有这些状态的核心都是相同的，所以这个被合并进来的项集中必然还包含项$[B\to\beta\cdot a\gamma, c]$，其中 c 是某个终结符号。如果这样的话，这个状态中同样也有在输入 a 上的移入/归约冲突，因此这个文法不是我们假设的 LR(1)文法。因此，合并具有相同核心的状态不会产生出原有状态中没有出现的移入/归约冲突，因为移入动作仅由核心决定，不考虑向前看符号。

然而，如下面的例子所示，合并项集可能会产生归约/归约冲突。

例 4.58 考虑文法

$$S'\to S$$
$$S\to a\ A\ d \mid b\ B\ d \mid a\ B\ e \mid b\ A\ e$$
$$A\to c$$
$$B\to c$$

该文法产生四个串 acd、ace、bcd 和 bce。读者可以构造出这个文法的 LR(1)项集，以验证该文法是 LR(1)的。完成这些工作之后，我们发现项集$\{[A\to c\cdot, d], [B\to c\cdot, e]\}$是可行前缀 ac 的有效项，$\{[A\to c\cdot, e], [B\to c\cdot, d]\}$是 bc 的有效项。这两个项集都没有冲突，并且它们的核心是相同的。然而，它们的并集，即

$$A\to c\cdot, d/e$$
$$B\to c\cdot, d/e$$

产生了一个归约/归约冲突，因为当输入为 d 或 e 的时候，这个合并项集既要求按照 $A\to c$ 进行归约，又要求按照 $B\to c$ 进行归约。 □

我们将给出两个 LALR 分析表构造算法，现在来介绍其中的第一个。这个算法的基本思想是构造出 LR(1)项集，如果没有出现冲突，就将具有相同核心的项集合并。然后我们根据合并后得到的项集族构造语法分析表。我们将要描述的方法的主要用途是定义 LRLA(1)文法。构造整个 LR(1)项集族需要的空间和时间太多，因此很少在实践中使用。

算法 4.59 一个简单，但空间需求大的 LALR 分析表的构造方法。

输入：一个增广文法 G'。

输出：文法 G'的 LALR 语法分析表函数 ACTION 和 GOTO。

方法：

1）构造 LR(1)项集族 $C=\{I_0, I_1, \cdots, I_n\}$。

2）对于 LR(1)项集中的每个核心，找出所有具有这个核心的项集，并将这些项集替换为它们的并集。

3）令 $C'=\{J_0, J_1, \cdots, J_m\}$是得到的 LR(1)项集族。状态 i 的语法分析动作是按照和算法 4.56 中的方法根据 J_i 构造得到的。如果存在一个分析动作冲突，这个算法就不能生成语法分析

器，这个文法就不是 LALR(1)的。

4) GOTO 表的构造方法如下。如果 J 是一个或多个 LR(1)项集的并集，也就是说 $J = I_1 \cup I_2 \cup \cdots \cup I_k$，那么 GOTO($I_1$, X)，GOTO(I_2, X)，…，GOTO(I_k, X)的核心是相同的，因为 I_1、I_2、…、I_k具有相同的核心。令 K 是所有和 GOTO(I_1, X)具有相同核心的项集的并集，那么 GOTO(J, X) = K。 □

算法 4.59 生成的分析表称为 G 的 *LALR 语法分析表*。如果没有语法分析动作冲突，那么给定的文法就称为 *LALR*(1)*文法*。在第(3)步中构造得到的项集族被称为 *LALR*(1)*项集族*。

例 4.60 再次考虑文法(4.55)。该文法的 GOTO 图已经显示在图 4-41 中。我们前面提到过，有三对可以合并的项集。I_3 和 I_6 被替换为它们的并集：

$$I_{36}: C \rightarrow c \cdot C,\ c/d/\$$$
$$C \rightarrow \cdot cC,\ c/d/\$$$
$$C \rightarrow \cdot d,\ c/d/\$$$

I_4 和 I_7 被替换为它们的并集：

$$I_{47}: C \rightarrow d \cdot,\ c/d/\$$$

I_8 和 I_9 被替换为它们的并集：

$$I_{89}: C \rightarrow cC \cdot,\ c/d/\$$$

这些压缩过的项集的 LALR 动作和 GOTO 函数显示在图 4-43 中。

要了解如何计算 GOTO 关系，考虑 GOTO(I_{36}, C)。在原来的 LR(1)项集中，GOTO(I_3, C) = I_8，而现在 I_8 是 I_{89}的一部分，因此我们令 GOTO(I_{36}, C)为 I_{89}。如果我们考虑 I_6，即 I_{36}的另一部分，我们仍然可以得到相同的结论。也就是说，GOTO(I_6, C) = I_9，I_9 现在是 I_{89}的一部分。再举一个例子。考虑 GOTO(I_2, c)，即在状态 I_2 上输入为 c 时执行移入之后的状态。在原来的 LR(1)项集中，GOTO(I_2, C) = I_6。因为 I_6 现在是 I_{36}的一部分，所以 GOTO(I_2, c)变成了 I_{36}。因此，图 4-43 中对应于状态 2 和输入 c 的条目被设置为 s36，表示移入并将状态 36 压入栈中。 □

状态	ACTION			GOTO	
	c	*d*	$	*S*	*C*
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

图 4-43 例子 4.54 的文法的 LALR 分析表

当处理语言 **c * dc * d** 中的一个串时，图 4-42 的 LR 语法分析器和图 4-43 的 LALR 语法分析器执行完全相同的移入和归约动作序列，尽管栈中状态的名字有所不同。比如，在 LR 语法分析器将 I_3 或 I_6 压入栈中时，LALR 语法分析器将 I_{36}压入栈中。这个关系对于所有的 LALR 文法都成立。在处理正确的输入时，LR 语法分析器和 LALR 语法分析器将相互模拟。

在处理错误的输入时，LALR 语法分析器可能在 LR 语法分析器报错之后继续执行一些归约动作。然而，LALR 语法分析器决不会在 LR 语法分析器报错之后移入任何符号。比如，在输入为 *ccd* 且后面跟有 $ 时，图 4-42 的 LR 语法分析器将

0 3 3 4

压入栈中，并且在状态 4 上发现一个错误，因为下一个输入符号是 $ 而状态 4 在 $ 上的动作为报错。相应地，图 4-43 中的 LALR 语法分析器将执行对应的操作，将

0 36 36 47

压入栈中。但是状态 47 在输入为 $ 时的动作为归约 $C \rightarrow d$。因此，LALR 语法分析器将把栈中内容改为

0 36 36 89

现在，状态 89 在输入 $ 上的动作为归约 $C\rightarrow cC$。栈中内容变为

0 36 89

此时仍要求进行一个类似的归约，得到栈

0 2

最后，状态 2 在输入 $ 上的动作为报错，因此现在发现了这个错误。

4.7.5 高效构造 LALR 语法分析表的方法

我们可以对算法 4.59 进行多处修改，使得在创建 LALR(1)语法分析表的过程中不需要构造出完整的规范 LR(1)项集族。

- 首先，我们可以只使用内核项来表示任意的 LR(0)或 LR(1)项集。也就是说，只使用初始项$[S'\rightarrow \cdot S]$或$[S'\rightarrow \cdot S, \$]$以及那些点不在产生式体左端的项来表示项集。
- 我们可以使用一个“传播和自发生成”的过程(我们稍后将描述这个方法)来生成向前看符号，根据 LR(0)项的内核生成 LALR(1)项的内核。
- 如果我们有了 LALR(1)内核，我们可以使用图 4-40 中的 CLOSURE 函数对各个内核求闭包，然后再把这些 LALR(1)项集当作规范 LR(1)项集族，使用算法 4.56 来计算分析表条目，从而得到 LALR(1)语法分析表。

例 4.61 我们将使用例子 4.48 中的非 SLR 文法作为一个例子，说明高效的 LALR(1)语法分析表构造方法。下面我们重新给出这个文法的增广形式：

$$S'\rightarrow S$$
$$S\rightarrow L=R \mid R$$
$$L\rightarrow *R \mid \mathbf{id}$$
$$R\rightarrow L$$

这个文法的完整 LR(0)项集显示在图 4-39 中。这些项集的内核显示在图 4-44 中。□

现在我们必须给这些用内核表示的 LR(0)项加上正确的向前看符号，创建出 LALR(1)项集的内核。在两种情况下，向前看符号 b 可以添加到某个 LALR(1)项集 J 中的 LR(0)项 $B\rightarrow\gamma\cdot\delta$ 之上：

I_0:	$S'\rightarrow \cdot S$	I_5:	$L\rightarrow \mathbf{id}\cdot$
I_1:	$S'\rightarrow S\cdot$	I_6:	$S\rightarrow L=\cdot R$
I_2:	$S\rightarrow L\cdot=R$ $R\rightarrow L\cdot$	I_7:	$L\rightarrow *R\cdot$
I_3:	$S\rightarrow R\cdot$	I_8:	$R\rightarrow L\cdot$
I_4:	$L\rightarrow *\cdot R$	I_9:	$S\rightarrow L=R\cdot$

图 4-44 文法(4.49)的 LR(0)项集的内核

1) 存在一个包含内核项$[A\rightarrow\alpha\cdot\beta, a]$的项集 I，并且 $J=\text{GOTO}(I, X)$。不管 a 为何值，在按照图 4-40 的算法构造 $\text{GOTO}(\text{CLOSURE}(\{[A\rightarrow\alpha\cdot\beta, a]\}), X)$ 时得到的结果中总是包含$[B\rightarrow\gamma\cdot\delta, b]$。对于 $B\rightarrow\gamma\cdot\delta$ 而言，这个向前看符号 b 被称为自发生成的。作为一个特殊情况，向前看符号 $ 对于初始项集中的项$[S'\rightarrow \cdot S]$而言是自发生成的。

2) 其余条件和(1)相同，但是 $a=b$，且按照图 4-40 所示计算 $\text{GOTO}(\text{CLOSURE}(\{[A\rightarrow\alpha\cdot\beta, b]\}), X)$ 得到的结果中包含$[B\rightarrow\gamma\cdot\delta, b]$的原因是项 $A\rightarrow\alpha\cdot\beta$ 有一个向前看符号 b。在这种情况下，我们说向前看符号从 I 的内核中的 $A\rightarrow\alpha\cdot\beta$ 传播到了 J 的内核中的 $B\rightarrow\gamma\cdot\delta$ 上。请注意，传播关系并不取决于某个特定的向前看符号，要么所有的向前看符号都从一个项传播到另一个项，要么都不传播。

我们需要确定每个 LR(0)项集中自发生成的向前看符号，同时也要确定向前看符号从哪些

项传播到了哪些项。这个检测实际上相当简单。令#为一个不在当前文法中的符号。令 $A\to\alpha\cdot\beta$ 为项集 I 中的一个内核 LR(0)项。对每个 X 计算 $J=\text{GOTO}(\text{CLOSURE}(\{[A\to\alpha\cdot\beta,\#]\}),X)$。对于 J 中的每个内核项，我们检查它的向前看符号集合。如果#是它的向前看符号，那么向前看符号就从 $A\to\alpha\cdot\beta$ 传播到了这个项。所有其他的向前看符号都是自发生成的。这个思想在下面的算法中被精确地表达了出来。这个算法还用到了一个性质：J 中的所有内核项中点的左边都是 X，也就是说，它们必然是形如 $B\to\gamma X\cdot\delta$ 的项。

算法 4.62 确定向前看符号。

输入： 一个 LR(0)项集 I 的内核 K 以及一个文法符号 X。

输出： 由 I 中的项为 GOTO(I, X)中内核项自发生成的向前看符号，以及 I 中将其向前看符号传播到 GOTO(I, X)中内核项的项。

方法： 算法在图 4-45 中给出。 □

```
for ( K 中的每个项 A → α·β ) {
    J := CLOSURE({[A → α·β,#]} );
    if ( [B → γ·Xδ, a] 在 J 中，并且 a 不等于# )
        断定 GOTO(I, X)中的项 B → γX·δ 的向前看符号 a
        是自发生成的 ;
    if ( [B → γ·Xδ, #] 在 J 中 )
        断定向前看符号从 I 中的项 A → α·β 传播到了 GOTO(I, X) 中的项
        B → γX·δ 之上 ;
}
```

图 4-45 发现传播的和自发生成的向前看符号

现在我们可以把向前看符号附加到 LR(0)项集的内核上，从而得到 LALR(1)项集。首先，我们知道 \$ 是初始 LR(0)项集中的 $S'\to\cdot S$ 的向前看符号。算法 4.62 给出了所有自发生成的向前看符号。将所有这些向前看符号列出之后，我们必须让它们不断传播，直到不能继续传播为止。有很多方法可以实现这个传播过程。从某种意义上说，所有这些方法都跟踪已经传播到某个项但是尚未传播出去的“新”向前看符号。下面的算法描述了一个将向前看符号传播到所有项中的技术。

算法 4.63 LALR(1)项集族的内核的高效计算方法。

输入： 一个增广文法 G'。

输出： 文法 G'的 LALR(1)项集族的内核。

方法：

1) 构造 G 的 LR(0)项集族的内核。如果空间资源不紧张，最简单的方法是像 4.6.2 节那样构造 LR(0)项集，然后再删除其中的非内核项。如果内存空间非常紧张，我们可以只保存各个项集的内核项，并在计算一个项集 I 的 GOTO 之前先计算 I 的闭包。

2) 将算法 4.62 应用于每个 LR(0)项集的内核和每个文法符号 X，确定 GOTO(I, X)中各内核项的哪些向前看符号是自发生成的，并确定向前看符号从 I 中的哪个项被传播到 GOTO(I, X)中的内核项上。

3) 初始化一个表格，表中给出了每个项集中的每个内核项相关的向前看符号。最初，每个项的向前看符号只包括那些被我们在步骤(2)中确定为自发生成的符号。

4) 不断扫描所有项集的内核项。当我们访问一个项 i 时，使用步骤(2)中得到的、用表格表示的信息，确定 i 将它的向前看符号传播到了哪些内核项中。项 i 的当前向前看符号集合被加到

和这些被传播的内核项相关联的向前看符号集合中。我们继续在内核项上进行扫描，直到没有新的向前看符号被传播为止。 □

例 4.64 我们为例子 4.61 的文法构造 LALR(1)项集的内核。这个文法的 LR(0)项集的内核如图 4-44 所示。当我们将算法 4.62 应用于项集 I_0 的内核时，我们首先计算 CLOSURE({[$S' \rightarrow \cdot S$, #]})，即

$S' \rightarrow \cdot S$, #　　$L \rightarrow \cdot * R$, #/ =

$S \rightarrow \cdot L = R$, #　　$L \rightarrow \cdot$ **id**, #/ =

$S \rightarrow \cdot R$, #　　$R \rightarrow \cdot L$, #

在这个闭包的项中，我们看到两个项中的向前看符号 = 是自发生成的。第一个项是 $L \rightarrow \cdot * R$。这个项中点的右边是 *，它生成了[$L \rightarrow * \cdot R$, =]。也就是说，= 是 I_4 中 $L \rightarrow * \cdot R$ 的自发生成的向前看符号。类似地，[$L \rightarrow \cdot$ **id**, =]告诉我们 = 是 I_5 中 $L \rightarrow$ **id** · 的自发生成的向前看符号。

因为#是这个闭包中六个项的向前看符号，所以我们确定 I_0 中的项 $S' \rightarrow \cdot S$ 将它的向前看符号传播到下面的六个项中：

I_1 中的 $S' \rightarrow S \cdot$　　I_4 中的 $L \rightarrow * \cdot R$

I_2 中的 $S \rightarrow L \cdot = R$　　I_5 中的 $L \rightarrow$ **id** ·

I_3 中的 $S \rightarrow R \cdot$　　I_2 中的 $R \rightarrow L \cdot$

在图 4-47 中，我们说明了算法 4.63 的步骤(3)和(4)。标号为 INIT 的列给出了各个内核项的自发生成的向前看符号。这些符号中只包括前面讨论过的 = 的两次出现，以及初始项 $S' \rightarrow \cdot S$ 的自发生成的向前看符号 $。

在第一趟扫描中，向前看符号 $ 从 I_0 中的 $S' \rightarrow \cdot S$ 传播到图 4-46 中列出的六个项上。向前看符号 = 从 I_4 中的 $L \rightarrow * \cdot R$ 传播到 I_7 中的 $L \rightarrow * R \cdot$ 和 I_8 中的 $R \rightarrow L \cdot$ 上。它还传递到它自身以及 I_5 中的 $L \rightarrow$ **id** · 上，但是这些向前看符号本来就已经存在了。在第二和第三趟扫描时，唯一被传播的新向前看符号是 $，它在第二趟扫描时被传播到 I_2 和 I_4 的后继中，并在第三趟扫描时到达 I_6 的后继中。在第四趟扫描时没有新的向前看符号被传播，因此最终的向前看符号集合如图 4-47 最右边的列所示。

自	到
I_0: $S' \rightarrow \cdot S$	I_1: $S' \rightarrow S \cdot$ I_2: $S \rightarrow L \cdot = R$ I_3: $R \rightarrow L \cdot$ I_4: $L \rightarrow * \cdot R$ I_5: $L \rightarrow$ **id** ·
I_2: $S \rightarrow L \cdot = R$	I_6: $S \rightarrow L = \cdot R$
I_4: $L \rightarrow * \cdot R$	I_4: $L \rightarrow * \cdot R$ I_5: $L \rightarrow$ **id** · I_7: $L \rightarrow * R \cdot$ I_8: $R \rightarrow L \cdot$
I_6: $S \rightarrow L = \cdot R$	I_4: $L \rightarrow * \cdot R$ I_5: $L \rightarrow$ **id** · I_8: $R \rightarrow L \cdot$ I_9: $S \rightarrow L = R \cdot$

图 4-46　向前看符号的传播

项集　项	向前看符号			
	初始值	第一趟	第二趟	第三趟
I_0: $S' \rightarrow \cdot S$	$	$	$	$
I_1: $S' \rightarrow S \cdot$		$	$	$
I_2: $S \rightarrow L \cdot = R$		$	$	$
$R \rightarrow L \cdot$		$	$	$
I_3: $S \rightarrow R \cdot$		$	$	$
I_4: $L \rightarrow * \cdot R$	=	=/$	=/$	=/$
I_5: $L \rightarrow$ **id** ·	=	=/$	=/$	=/$
I_6: $S \rightarrow L = \cdot R$			$	$
I_7: $L \rightarrow * R \cdot$		=	=/$	=/$
I_8: $R \rightarrow L \cdot$		=	=/$	=/$
I_9: $S \rightarrow L = R \cdot$				$

图 4-47　向前看符号的计算

请注意,在例4-48 中,使用 SLR 方法时发现的移入/归约冲突在使用 LALR 技术时消失了。虽然 I_2 中的 $S \rightarrow L \cdot = R$ 生成了在输入 = 上的移入动作,但是 I_2 中 $R \rightarrow L \cdot$ 的向前看符号只包括 $,因此两者之间不再有冲突。 □

4.7.6 LR 语法分析表的压缩

一个典型的具有 50 ~ 100 个终结符号和 100 个产生式的程序设计语言文法的 LALR 语法分析表中可能包含几百个状态。分析表的动作函数常常包含 20000 多个条目,每个条目至少需要 8 个二进制位进行编码。对于小型设备,有一个比二维数组更加高效的编码方法是很重要的。我们将简短地描述一些可以用于压缩 LR 语法分析表中的 ACTION 字段和 GOTO 字段的技术。

一个可用于压缩动作字段的技术所基于的原理是动作表中通常有很多相同的行。比如,图4-42中的状态 0 和3 就有相同的动作条目,状态2 和6 也是这样。因此,如果我们为每个状态创建一个指向一维数组的指针,我们就可以节省可观的空间,而付出的时间代价却很小。具有相同动作的状态的指针指向相同的位置。为了从这个数组获取信息,我们给各个终结符号赋予一个编号,编号范围为从零开始到终结符号总数减一。对于每个状态,这个整数编号将作为从指针值开始的偏移量。在一个给定的状态中,第 i 个终结符号对应的语法分析动作可以在该状态的指针值之后的第 i 个位置上找到。

如果为每个状态创建一个动作列表,我们可以获得更高的空间效率,但语法分析器会变慢。这个列表由(终结符号,动作)对组成。一个状态的最频繁的动作可以放在列表的结尾处,并且我们可以在这个对中原本放终结符号的地方放上符号“**any**”,表示如果没有在列表中找到当前输入,那么不管这个输入是什么,我们都选择这个动作。不仅如此,为了使得一行中的内容更加一致,我们可以把报错条目安全地替换为规约动作。对错误的检测会稍有延后,但仍可以在执行下一个移入动作之前发现错误。

例 4.65 考虑图 4-37 的语法分析表。首先,请注意状态 0、4、6 和 7 的动作是相同的。我们可以用下面的列表来表示它们:

符号	动作
id	s5
(	s4
any	error

状态 1 有一个类似的列表:

+	s6
$	acc
any	error

在状态 2 中,我们可以把报错条目替换为 r2,因此对于除 * 之外的输入都按照产生式 2 进行归约。因此状态 2 的列表是

*	s7
any	r2

状态 3 只有报错和 r4 条目。我们可以把前者替换为后者,因此状态 3 的列表只有一个对(**any**, r4)。状态 5、10 和 11 也可以做类似处理。状态 8 的列表是

+	s6
)	s11
any	error

而状态 9 的列表是

*	s7
any	r1

□

我们也可以把 GOTO 表编码为一个列表，但这里更加高效的方法是为每个非终结符号 A 构造一个数对的列表。A 的列表中的每个对形如(当前状态，下一状态)，表示

GOTO[当前状态，A] = 下一状态

这个技术很有用，因为 GOTO 表的一列中常常只有很少几个状态。原因是对于某个非终结符号 A 上的 GOTO 目标状态的项集中必然存在某些项，这些项中 A 紧靠在点的左边。对于任意两个不同的文法符号 X、Y，没有哪个 GOTO 目标项集既有点左边为 X 的项，又有点左边为 Y 的项。因此，每个状态最多只出现在 GOTO 表的一列中。

为了进一步减少使用的空间，我们注意到 GOTO 表中的报错条目从来都不会被查询到。因此，我们可以把每个报错条目替换为该列中最常用的非报错条目。这个条目变成了默认选择。在每一列的列表中，它被表示为一个“当前状态”字段为 **any** 的对。

例 4.66 再次考虑图 4-37。F 对应的列中与状态 7 对应的条目是 10，所有其他的条目所对应的要么是 3 要么报错。我们可以用 3 来替换报错条目，为 F 列创建列表

当前状态	下一状态
7	10
any	3

类似地，T 列的列表可以是

6	9
any	2

对于 E 列，我们可以选择 1 或 8 作为默认选择。这两种选择都需要两个列表条目。比如，我们可以为 E 列创建如下列表

4	8
any	1

□

这些小例子中体现出来的空间节省效果可能具有误导性。因为在这个例子和前一个例子中创建的列表中的条目数量，再加上从状态到动作列表的指针以及从非终结符号到后继状态表的指针，它们需要的空间和图 4-37 中的矩阵实现方法相比，并没有令人印象深刻的空间节省效果。但是对于现实中的文法，列表表示法所需要的空间通常比矩阵表示法所需空间少 10%。在 3.9.8 节中讨论的用于有穷自动机的表压缩方法也可以用来表示 LR 语法分析表。

4.7.7 4.7 节的练习

练习 4.7.1：为练习 4.2.1 的文法 $S \rightarrow S\ S\ +\ |\ S\ S\ *\ |\ a$ 构造

1) 规范 LR 项集族。

2) LALR 项集族。

练习 4.7.2：对练习 4.2.2(1) ~ (7) 的各个(增广)文法重复练习 4.7.1。

! **练习 4.7.3**：对练习 4.7.1 的文法，使用算法 4.63，根据该文法的 LR(0)项集的内核构造出它的 LALR 项集族。

! **练习 4.7.4**：说明下面的文法

$$S \rightarrow A\ a\ |\ b\ A\ c\ |\ d\ c\ |\ b\ d\ a$$

$$A \rightarrow d$$

是 LALR(1)的，但不是 SLR(1)的。

！ **练习 4.7.5**：说明下面的文法

$$S \rightarrow A\ a \mid b\ A\ c \mid B\ c \mid b\ B\ a$$
$$A \rightarrow d$$
$$B \rightarrow d$$

是 LR(1)的，但不是 LALR(1)的。

4.8 使用二义性文法

实际上，每个二义性文法都不是 LR 的，因此它们不在前面两节讨论的任何文法类之内。然而，某些类型的二义性文法在语言的规约和实现中很有用。对于像表达式这样的语言构造，二义性文法能提供比任何等价的无二义性文法更短、更自然的规约。二义性文法的另一个用途是隔离经常出现的语法构造，以对其进行特殊的优化。使用二义性文法，我们可以向文法中精心加入新的产生式来描述特殊情况的构造。

虽然使用的文法是二义性的，但我们在所有的情况下都会给出消除二义性的规则，使得每个句子只有一棵语法分析树。通过这个方法，语言的规约在整体上是无二义性的，有时还可以构造出遵循这个二义性解决方法的 LR 语法分析器。我们强调应该保守地使用二义性构造，并且必须在严格控制之下使用，否则无法保证一个语法分析器识别的到底是什么样的语言。

4.8.1 用优先级和结合性解决冲突

考虑带有运算符 + 和 * 的有二义性的表达式文法(4.3)。为方便起见，这里再次给出此文法：

$$E \rightarrow E\ +\ E \mid E\ *\ E \mid (\ E\ ) \mid \mathbf{id}$$

这个文法是二义性的，因为它没有指明运算符 + 和 * 的优先级和结合性。无二义性的文法(4.1)(包含产生式 $E \rightarrow E\ +\ T$ 和 $T \rightarrow T\ *\ F$)生成同样的语言，但是指定 + 的优先级低于 *，并且两个运算符都是左结合的。出于两个原因，我们愿意使用这个二义性文法。第一，我们将会看到的，可以很容易地改变运算符 + 和 * 的优先级和结合性，既不需要修改文法(4.3)的产生式，也不需要改变相应语法分析器的状态数目。第二，相应无二义性文法的语法分析器将把部分时间用于归约产生式 $E \rightarrow T$ 和 $T \rightarrow F$。这两个产生式的功能就是保证结合性和优先级。二义性文法(4.3)的语法分析器不会把时间浪费在对这些单产生式(即产生式体中只包含一个非终结符号的产生式)的归约上。

使用 $E' \rightarrow E$ 增广之后的二义性表达式文法(4.3)的 LR(0)项集显示在图 4-48 中。因为文法(4.3)是二义性的，在我们试图用这些项集生成一个 LR 语法分析表时会出现分析动作冲突。对应于项集 I_7 和 I_8 的两个状态就产生了这样的冲突。假设我们使用 SLR 方法来

I_0:
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

I_1:
$E' \rightarrow E \cdot$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

I_2:
$E \rightarrow (\cdot E)$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

I_3:
$E \rightarrow \mathbf{id} \cdot$

I_4:
$E \rightarrow E + \cdot E$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

I_5:
$E \rightarrow E * \cdot E$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

I_6:
$E \rightarrow (E \cdot)$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

I_7:
$E \rightarrow E + E \cdot$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

I_8:
$E \rightarrow E * E \cdot$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

I_9:
$E \rightarrow (E) \cdot$

图 4-48 一个增广表达式文法的 LR(0)项集

构造语法分析动作表。I_7 在输入 + 或 * 上产生了冲突，不能确定应该按照 $E \rightarrow E + E$ 归约还是应该移入。这个冲突无法解决，因为 + 和 * 都在 FOLLOW(E) 中。因此在输入为 * 或 + 时，这两种动作都被要求执行。I_8 也产生了类似的冲突，即在输入为 + 或 * 时，不能确定应该按照 $E \rightarrow E * E$ 归约还是应该移入。实际上，任意一种 LR 语法分析表构造方法都会产生这样的冲突。

然而，这些问题可以使用 + 和 * 的优先级和结合性信息来解决。考虑输入 **id** + **id** * **id**。它使得基于图 4-48 的语法分析器在处理完 **id** + **id** 之后进入状态 7。更明确地说，语法分析器进入如下的格局：

前缀	栈	输入
$E + E$	0 1 4 7	* **id** $

为方便起见，我们同时将对应于状态 1、4 和 7 的符号显示在“前缀”列中。

如果 * 的优先级高于 +，我们知道语法分析器应该将 * 移入栈中，准备将这个 * 和它两边的 **id** 符号归约为一个表达式。图 4-37 显示了根据等价的无二义性文法得到的 SLR 语法分析器。这个分析器也做出同样的选择。另一方面，如果 + 的优先级高于 *，我们知道语法分析器应该将 $E + E$ 归约为 E。因此，+ 和 * 之间的相对优先关系可以被用于解决状态 7 上的冲突，确定在输入 * 上应该按照 $E \rightarrow E + E$ 归约还是应该移入。

假如输入是 **id** + **id** + **id**，语法分析器在处理了输入 **id** + **id** 之后，仍然能获得栈内容为 0 1 4 7 的格局。在输入为 + 时，状态 7 中仍然有一个移入/归约冲突。然而，现在运算符 + 的结合性可以决定如何解决这个冲突。如果 + 是左结合的，正确的动作是按照 $E \rightarrow E + E$ 进行归约。也就是说，第一个 + 号两边的 **id** 必须被分在一组。这个选择仍然和相应无二义性文法的 SLR 语法分析器的做法一致。

概括地讲，假设 + 是左结合的，状态 7 在输入 + 时的动作应该是按照 $E \rightarrow E + E$ 进行归约。假设 * 的优先级高于 +，状态 7 在输入 * 上的动作应该是移入。类似地，假设 * 是左结合的，并且它的优先级高于 +。因为只有当栈中最上端的三个符号是 $E * E$ 时，状态 8 才能出现在栈顶。我们可以认为状态 8 在输入 * 和 + 上的动作都是按照 $E \rightarrow E * E$ 归约。对于输入为 + 的情况，理由是 * 的优先级高于 +；而对于输入为 * 的情况，理由是 * 是左结合的。

状态	ACTION						GOTO
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

图 4-49 文法(4.3)的语法分析表

按照这个方式进行处理，我们可以得到图 4-49 所示的 LR 语法分析表。产生式 1 ~ 4 分别是 $E \rightarrow E + E$、$E \rightarrow E * E$、$E \rightarrow (E)$ 和 $E \rightarrow$ **id**。很有意思的是，如果从图 4-37 所示的无二义性表达式文法(4.1)的 SLR 分析表中删除单产生式 $E \rightarrow T$ 和 $T \rightarrow F$ 的归约动作，我们可以得到一个相似的语法动作表。在使用 LALR 和规范 LR 语法分析技术时，我们也可以使用类似的方法来处理这种二义性文法。

4.8.2 “悬空-else”的二义性

再次考虑下面的条件语句文法：

stmt → **if** *expr* **then** *stmt* **else** *stmt*

| **if** *expr* **then** *stmt*

| **other**

如我们在 4.3.2 节中指出的，这个文法是二义性的，因为它没有解决悬空-else 的二义性问题。为了简化这个讨论，我们考虑这个文法的一个抽象表示，其中 i 表示 **if** *expr* **then**，e 表示 **else**，a 表示“所有其他的产生式”。那么我可以用增广产生式 $S'\rightarrow S$ 重写这个文法：

$$S'\rightarrow S$$

$$S\rightarrow i\ S\ e\ S \mid i\ S \mid a \tag{4.67}$$

文法(4.67)的 LR(0)项集显示在图 4-50 中。因为文法(4.67)的二义性，在 I_4 中有一个移入/归约冲突。在该项集中，$S\rightarrow iS\cdot eS$ 要求将 e 移入，又因为 FOLLOW(S) = {e, \$}，项 $S\rightarrow iS\cdot$ 要求在输入为 e 的时候用 $S\rightarrow iS$ 进行归约。

I_0: $S'\rightarrow\cdot S$
$S\rightarrow\cdot iSeS$
$S\rightarrow\cdot iS$
$S\rightarrow\cdot a$

I_1: $S'\rightarrow S\cdot$

I_2: $S\rightarrow i\cdot SeS$
$S\rightarrow i\cdot S$
$S\rightarrow\cdot iSeS$
$S\rightarrow\cdot iS$
$S\rightarrow\cdot a$

I_3: $S\rightarrow a\cdot$

I_4: $S\rightarrow iS\cdot eS$
$S\rightarrow iS$

I_5: $S\rightarrow iSe\cdot S$
$S\rightarrow\cdot iSeS$
$S\rightarrow\cdot iS$
$S\rightarrow\cdot a$

I_6: $S\rightarrow iSeS\cdot$

图 4-50 增广文法(4.67)的 LR(0)状态

把这些讨论翻译回 **if-then-else** 的术语，假设栈中内容为

if *expr* **then** *stmt*

且 **else** 是第一个输入符号，我们应该将 **else** 移入栈中(即移入 e)呢？还是应该将 **if** *expr* **then** *stmt* 归约(即按照 $S\rightarrow iS$ 归约)呢？答案是我们应该移入 **else**，因为它是和前一个 **then** “相关”的。按照文法(4.67)的术语，输入中代表 **else** 的 e 只能作为以 iS 开头的产生式体的一部分，而现在栈顶内容就是 iS。如果输入中跟在 e 后面的符号不能被归约为 S，使得分析器无法归约得到完整的产生式体 $iSeS$，那么可以证明别的语法分析过程也不可能得到这个产生式体。

我们可以确定在解决 I_4 中的移入/归约冲突时应该在输入为 e 时执行移入动作。使用这个方式解决了 I_4 在输入 e 上的语法分析动作冲突之后，根据图 4-50 的项集构造得到的 SLR 语法分析表显示在图 4-51 中。产生式 1 ~ 3 分别是 $S\rightarrow iSeS$、$S\rightarrow iS$ 和 $S\rightarrow a$。

比如，在处理输入 *iiaea* 时，根据正确的“悬空-else”冲突的解决方法，语法分析器执行了图 4-52 中所示的步骤。在第 5 行，状态 4 在输入 e 上选择了移入动作；而在第 9 行，状态 4 在输入 \$ 上要求按照 $S\rightarrow iS$ 进行归约。

状态	ACTION				GOTO
	i	e	a	\$	S
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		s5		r2	
5	s2		s3		6
6		r1		r1	

图 4-51 悬空 else 文法的 LR 分析表

	栈	符号	输入	动作
(1)	0		$iiaea\$$	移入
(2)	0 2	i	$iaea\$$	移入
(3)	0 2 2	ii	$aea\$$	移入
(4)	0 2 2 3	iia	$ea\$$	根据 $S\rightarrow a$ 归约
(5)	0 2 2 4	iiS	$ea\$$	移入
(6)	0 2 2 4 5	$iiSe$	$a\$$	移入
(7)	0 2 2 4 5 3	$iiSea$	\$	根据 $S\rightarrow a$ 归约
(8)	0 2 2 4 5 6	$iiSeS$	\$	根据 $S\rightarrow iSeS$ 归约
(9)	0 2 4	iS	\$	根据 $S\rightarrow iS$ 归约
(10)	0 1	S	\$	接受

图 4-52 处理输入 *iiaea* 时的语法分析动作

我们做一个比较，如果我们不能使用二义性文法来描述条件语句，那么我们将不得不使用例 4.16 中给出的笨拙的文法来描述。

4.8.3 LR 语法分析中的错误恢复

当 LR 语法分析器在查询语法分析动作表并发现一个报错条目时，它就检测到了一个语法错误。在查询 GOTO 表时不会发现语法错误。如果当前已扫描的输入部分不可能存在正确的后续符号串，LR 语法分析器就会立刻报错。规范 LR 语法分析器不会做任何多余的归约动作，会立刻报告错误。SLR 和 LALR 语法分析器可能会在报错之前执行几次归约动作，但是它们决不会把一个错误的输入符号移入到栈中。

在 LR 语法分析过程中，我们可以按照如下方式实现恐慌模式的错误恢复策略。我们从栈顶向下扫描，直到发现某个状态 s，它有一个对应于某个非终结符号 A 的 GOTO 目标。然后我们丢弃零个或多个输入符号，直到发现一个可能合法地跟在 A 之后的符号 a 为止。之后语法分析器将 GOTO(s, A)压入栈中，继续进行正常的语法分析。在实践中可能会选择多个这样的非终结符号 A。通常这些非终结符号代表了主要的程序段，比如表达式、语句或块。比如，如果 A 是非终结符号 *stmt*，a 就可能是分号或者}。其中，}标记了一个语句序列的结束。

这个错误恢复方法试图消除包含语法错误的短语。语法分析器确定一个从 A 推导出的串中包含错误。这个串的一部分已经被处理，并形成了栈顶部的一个状态序列。这个串的其余部分还在输入中，语法分析器则在输入中查找可以合法地跟在 A 后面的符号，从而试图跳过这个串的其余部分。通过从栈中删除状态，跳过一部分输入，并将 GOTO(s, A)压入栈中，语法分析器假装它已经找到了 A 的一个实例，并继续进行正常的语法分析。

实现短语层次错误恢复的方法如下：检查 LR 语法分析表中的每个报错条目，并根据语言的使用方法来决定程序员所犯的何种错误最有可能引起这个语法错误。然后构造出适当的恢复过程，通常会根据各个报错条目来确定适当的修改方法，修改栈顶状态和/或第一个输入符号。

在为一个 LR 语法分析器设计专门的错误处理例程时，我们可以在表的动作字段的每个空条目中填写一个指向错误处理例程的指针。该例程将执行编译器设计者所选定的恢复动作。这些动作包括在栈和/或输入中删除或插入符号，也包含替换输入符号或将输入符号换位。我们必须谨慎地做出选择，避免 LR 语法分析器陷入无限循环。一个安全的策略是保证最终至少有一个输入符号被删除或移入，并且如果到达输入结束位置时要保证栈会缩小。应该避免从栈中弹出一个和某非终结符号对应的状态，因为这样的修改相当于从栈中消除了一个已经被成功分析的语言构造。

例 4.68 再次考虑表达式文法

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

图 4-49 中显示了这个文法的 LR 分析表。图 4-53中显示的是对这个分析表进行修改后得到的语法分析表。修改后的表添加了错误检测和恢复的动作。对于那些在某些输入上执行特定归约动作的状态，我们将这个状态中的报错条目替换为这个归约动作。这种修改可能会使得报错延后至一次或多次归约动作之后，但是错误仍然会在任何移入动作发生之前被发现。图 4-49 中剩余的空白项已经被替换为对错误处理与过程的调用。

状态	ACTION						GOTO
	id	+	*	(	)	$	*E*
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	acc	
2	s3	e1	e1	s2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	s3	e1	e1	s2	e2	e1	7
5	s3	e1	e1	s2	e2	e1	8
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

图 4-53 带有错误处理子程序的 LR 语法分析表

错误处理例程如下：

e1：这个例程在状态 0、2、4 和 5 上被调用。所有这些状态都期望读入一个运算分量的第一个符号，这个符号可能是 **id** 或左括号，但是实际读入的却是 + 、 * 或输入结束标记。

将状态 3(状态 0、2、4 和 5 在输入 **id** 上的 GOTO 目标)压入栈中；

发出诊断信息“缺少运算分量。”

e2：在状态 0、1、2、4 和 5 上发现输入为右括号时调用这个过程。

从输入中删除右括号；

发出诊断信息“不匹配的右括号。”

e3：当在状态 1 和 6 上，期待读入一个运算符却发现了一个 **id** 或左括号时调用。

将状态 4(对应于符号 + 的状态)压入栈中。

发出诊断信息“缺少运算符。”

e4：当在状态 6 上发现输入结束标记时调用。

将状态 9(对应于右括号)压入栈中；

发出诊断信息“缺少右括号。”

在处理错误的输入 **id** +) 时，语法分析器进入的格局序列显示在图 4-54 中。 □

栈	符号	输入	动作
0		id +) $	
0 3	id	+) $	
0 1	E	+) $	
0 1 4	E +	) $	“不匹配的右括号” e2 删除了右括号
0 1 4	E +	$	“缺少运算分量” e1 将状态 3 压入栈中
0 1 4 3	E + id	$	
0 1 4 7	E + E	$	
0 1	E	$	

图 4-54 一个 LR 语法分析器所做的语法分析和错误恢复步骤

4.8.4 4.8 节的练习

！练习 4.8.1：下面是一个二义性文法，它描述了包含 n 个二目中缀运算符且具有 n 个不同优先级的表达式：

$$E \rightarrow E\ \theta_1\ E \mid E\ \theta_2\ E \mid \cdots \mid E\ \theta_n\ E \mid (\ E\ ) \mid \mathbf{id}$$

1) 将 SLR 项集表示为 n 的函数。

2) 要使得所有的运算符都是左结合的，并且 θ_1 的优先级高于 θ_2，θ_2 的优先级高于 θ_3，依次类推，我们应该如何解决 SLR 项之间的冲突？

3) 根据你在(2)中的决定，给出相应的 SLR 语法分析表。

4) 图 4-55 中的无二义性文法定义了相同的表达式集合。对这个文法重复(1)和(3)部分。

5) 比较这两个(二义性和无二义性)文法的项集总数以及它们的语法分析表的大小，你能得出什么结论？关于二义性表达式文法的使用，这个比较结果告诉我们什么信息？

$$\begin{array}{lcl} E_1 & \rightarrow & E_1\,\theta_n\,E_2 \mid E_2 \\ E_2 & \rightarrow & E_2\,\theta_{n-1}\,E_3 \mid E_3 \\ & \cdots & \\ E_n & \rightarrow & E_n\ \theta_1 E_{n+1} \mid E_{n+1} \\ E_{n+1} & \rightarrow & (\ E_1\ ) \mid \mathbf{id} \end{array}$$

图 4-55 含有 n 个运算符的表达式的无二义性文法

！练习 4.8.2：图 4-56 给出了某种语句的文法。这些语句和练习 4.4.12 中讨论的语句类似。在这里，e 和 s 仍然是分别代表条件表达式和“其他语句”的终结符号。

1) 为这个文法构造一个 LR 语法分析表，并用解决悬空-else问题的常用方法来解决其中的冲突。

2) 在这个语法分析表中填入额外的归约动作或适当的错误恢复例程，实现语法分析中的错误恢复。

3) 给出你的语法分析器在处理下列输入时的行为：

ⓐ **if** e **then** s ; **if** e **then** s **end**

ⓑ **while** e **do begin** s ; **if** e **then** s ; **end**

$$\begin{array}{lcl} stmt & \rightarrow & \textbf{if}\ e\ \textbf{then}\ stmt \\ & \mid & \textbf{if}\ e\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\ & \mid & \textbf{while}\ e\ \textbf{do}\ stmt \\ & \mid & \textbf{begin}\ list\ \textbf{end} \\ & \mid & s \\ list & \rightarrow & list\ ;\ stmt \\ & \mid & stmt \end{array}$$

图 4-56 某类语句的文法

4.9 语法分析器生成工具

本节将介绍如何使用语法分析器生成工具来帮助构造一个编译器的前端。我们将使用 LALR 语法分析器生成工具 Yacc 作为我们讨论的基础，因为它实现了我们在前两节中讨论的很多概念，并且这个工具很容易获得。Yacc 表示“yet another compiler-compiler”，即“又一个编译器的编译器”。这个名字反映出当 S. C. Johnson 在 20 世纪 70 年代早期创建出 Yacc 的第一个版本时，语法分析器生成工具非常流行。Yacc 在 UNIX 系统中是以命令的方式出现的，它已经用于实现多个编译器产品。

4.9.1 语法分析器生成工具 Yacc

按照图 4-57 中演示的方法就可以使用 Yacc 来构造一个翻译器。首先要准备好一个文件，比如 translate. y，文件中包含了对将要构造的翻译器的规约。UNIX 系统命令

```
yacc translate.y
```

使用算法 4.63 中给出的 LALR 方法将文件 translate. y 转换成为一个名为 y. tab. c 的 C 程序。程序 y. tab. c 是一个用 C 语言编写的 LALR 语法分析器，另外还包括由用户准备的 C 语言例程。其中的 LALR 分析表是按照 4.7 节中描述的方法压缩的。使用命令

```
cc y.tab.c -ly
```

[⊖]

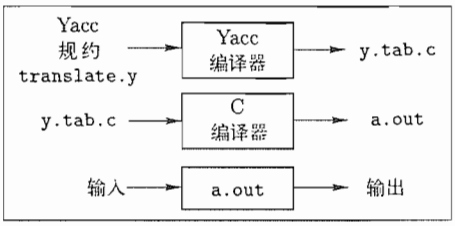

图 4-57 用 Yacc 创建一个输入/输出翻译器

对 y. tab. c 进行编译，并和包含 LR 语法分析程序的库 ly 连接，我们就得到了想要的目标程序 *a. out*。这个程序执行了由最初的 Yacc 程序 translate. y 所描述的翻译工作。如果需要其他过程，它们可以和其他的 C 程序一样，和 y. tab. c 一起编译并加载。

一个 Yacc 源程序由三个部分组成：

```
声明
%%
翻译规则
%%
辅助性 C 语言例程
```

例 4.69 为了说明如何编写一个 Yacc 源程序，我们构造一个简单的桌上计算器。该计算器读入一个算术表达式，对表达式求值，然后打印出表达式的结果。我们将从下面的算术表达式文法开始构造这个桌上计算器：

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \textbf{digit}$$

其中的词法单元 **digit** 是一个 0 ~ 9 之间的数字。根据这个文法得到的 Yacc 桌上计算器程序显示在图 4-58 中。 □

声明部分

一个 Yacc 程序的声明部分分为两节，它们都是可选的。在第一节中放置通常的 C 声明，这个声明用%{和}%括起来。那些由第二和第三部分中的翻译规则及过程使用的临时变量都在这里

⊖ 函数库的名字 ly 和具体系统相关。

声明。在图 4-58 中，这一节只包含 include 语句

```
#include <ctype.h>
```

这个语句使得 C 语言的预处理器将标准头文件 < ctype. h > 包含进来，这个头文件中包含了断言 isdigit。

在声明部分中还包括对词法单元的声明。在图 4-58 中，语句

```
%token DIGIT
```

声明 DIGIT 是一个词法单元。在这一节中声明的词法单元可以在 Yacc 规约的第二和第三部分中使用。如果向 Yacc 语法分析器传送词法单元的词法分析器是使用 Lex 创建的，那么如 3. 5. 2 节中讨论的，Lex 生成的词法分析器也可以使用这里声明的词法单元。

```
%{
#include <ctype.h>
%}

%token DIGIT

%%
line   : expr '\n'        { printf("%d\n", $1); }
       ;
expr   : expr '+' term    { $$ = $1 + $3; }
       | term
       ;
term   : term '*' factor  { $$ = $1 * $3; }
       | factor
       ;
factor : '(' expr ')'     { $$ = $2; }
       | DIGIT
       ;
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

图 4-58 一个简单的桌上计算器的 Yacc 规约

翻译规则部分

我们将翻译规则放置在 Yacc 规约中第一个%%对之后的部分。每个规则由一个文法产生式和一个相关联的语义动作组成。我们前面写作

$$<\text{产生式头}> \rightarrow <\text{产生式体}>_1 \mid <\text{产生式体}>_2 \mid \cdots \mid <\text{产生式体}>_n$$

的一组产生式在 Yacc 中被写成：

$$<\text{产生式头}> : <\text{产生式体}>_1 \{ <\text{语义动作}>_1 \}$$
$$\mid <\text{产生式体}>_2 \{ <\text{语义动作}>_2 \}$$
$$\cdots$$
$$\mid <\text{产生式体}>_n \{ <\text{语义动作}>_n \}$$
$$;$$

在一个 Yacc 产生式中，如果一个由字母和数位组成的字符串没有加引号且未被声明为词法单元，它就会被当作非终结符号处理。带引号的单个字符，比如‘c’，会被当作终结符号 c 以及

它所代表的词法单元所对应的整数编码(即 Lex 将把'c'的字符编码当作整数返回给语法分析器)。不同的产生式体用竖线分开,每个产生式头以及它的可选产生式体及语义动作之后跟一个分号。第一个产生式的头符号被看作开始符号。

一个 Yacc 语义动作是一个 C 语句的序列。在一个语义动作中,符号 $$ 表示和相应产生式头的非终结符号关联的属性值,而 $i 表示和相应产生式体中第 i 个文法符号(终结符号或非终结符号)关联的属性值。当我们按照一个产生式进行归约时就会执行和该产生式相关联的语义动作,因此语义动作通常根据 $i 的值来计算 $$ 的值。在上面的 Yacc 规范中,我们将两个 E 产生式

$$E \rightarrow E + T \mid T$$

和它们的相关语义动作写作:

```
expr : expr '+' term    { $$ = $1 + $3; }
     | term
     ;
```

请注意,第一个产生式中的非终结符号 term 是该产生式体中的第三个文法符号,而 + 是第二个文法符号。与第一个产生式关联的语义动作将产生式体中的 expr 和 term 的值相加,并把结果赋给产生式头上的非终结符号 expr。我们省略了第二个产生式的语义动作,因为对于体中只包含一个文法符号的产生式,默认的语义动作就是拷贝属性值。总的来说,默认动作是{ $ $ = $ 1; }。

请注意,我们向这个 Yacc 规范中加入了一个新的开始符号产生式

```
line : expr '\n'    { printf("%d\n", $1); }
```

这个产生式说明桌面计算器的输入是一个跟着换行符的表达式。和这个产生式相关的语义动作打印出了输入表达式的十进制取值和一个换行符。

辅助性 C 语言例程部分

一个 Yacc 规约的第三部分由辅助性 C 语言例程组成。这里必须提供一个名为 yylex( )的词法分析器。用 Lex 来生成 yylex( )是一个常用的选择,见 4.9.3 节。在需要时可以添加错误恢复例程这样的过程。

词法分析器 yylex( )返回一个由词法单元名和相关属性值组成的词法单元。如果要返回一个词法单元名字,比如 DIGIT,那么这个名字必须先在 Yacc 规约的第一部分进行声明。一个词法单元的相关属性值通过一个 Yacc 定义的变量 yylval 传送给语法分析器。

图 4-58 中的词法分析器是非常原始的。它使用 C 函数 getchar( )逐个读入字符。如果字符是一个数位,这个数位的值就存放在变量 yylval 中,返回词法单元的名字 DIGIT。否则,字符本身被当作词法单元名返回。

4.9.2 使用带有二义性文法的 Yacc 规约

现在让我们修改这个 Yacc 规约,使得这个桌面计算器更加有用。首先,我们将允许桌面计算器对一个表达式序列进行求值,其中每个表达式占一行。我们还将允许表达式之间出现空行。我们将第一个规则修改为:

```
lines : lines expr '\n'    { printf("%g\n", $2); }
      | lines '\n'
      | /* empty */
      ;
```

在 Yacc 中,像第三行那样的空白产生式表示 ϵ。

其次,我们将扩展表达式的种类,使得它的语言可以包含数字,而不是单个数位,并且包含算术运算符 + 、-(包括双目和单目)、* 和/。描述这类表达式的最容易的方式是使用下面的二义性文法:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid - E \mid (E) \mid \textbf{number}$$

得到的 Yacc 规约如图 4-59 所示。

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double  /* double type for Yacc stack */
%}
%token NUMBER

%left '+' '-'
%left '*' '/'
%right UMINUS
%%

lines : lines expr '\n'   { printf("%g\n", $2); }
      | lines '\n'
      | /* empty */
      ;
expr  : expr '+' expr     { $$ = $1 + $3; }
      | expr '-' expr     { $$ = $1 - $3; }
      | expr '*' expr     { $$ = $1 * $3; }
      | expr '/' expr     { $$ = $1 / $3; }
      | '(' expr ')'      { $$ = $2; }
      | '-' expr  %prec UMINUS { $$ = - $2; }
      | NUMBER
      ;
%%
yylex() {
    int c;
    while ( ( c = getchar() ) == ' ' );
    if ( (c == '.') || (isdigit(c)) ) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    return c;
}
```

图 4-59 一个更加先进的桌上计算器的 Yacc 规约

因为图 4-59 中 Yacc 规约的文法是二义性的，LALR 算法将会出现语法分析动作冲突。Yacc 会报告产生的语法分析动作冲突的数量。使用 -v 选项调用 Yacc 可以得到关于项集和语法分析动作冲突的描述。这个选项会产生一个附加的文件 y.output，它包含文法的项集的内核，对 LALR 算法产生的语法分析动作冲突的描述，以及 LR 语法分析表的一个可读表示形式。这个可读表示形式显示了 Yacc 是如何解决这些语法分析动作冲突的。只要 Yacc 报告发现了语法分析动作冲突，那么最好创建并查阅 y.output 文件，了解为什么会产生这些语法分析动作冲突，并检查 Yacc 是否已经正确解决了它们。

除非另行指定，否则 Yacc 会使用下面的两个规则来解决所有的语法分析动作冲突：

1）解决一个归约/归约冲突时，选择在 Yacc 规约中列在前面的那个冲突产生式。

2）解决移入/归约冲突时总是选择移入。这个规则正确地解决了因为悬空 else 二义性而产生的移入/归约冲突。

因为这些默认规则不可能总是编译器作者需要的，所以 Yacc 提供了一个通用的机制来解决移入/归约冲突。在声明部分，我们可以给终结符号赋予优先级和结合性。声明

```
%left '+' '-'
```

使得 + 和 - 具有相同的优先级，并且都是左结合的。我们可以把一个运算符声明为右结合的，比如：

```
%right '^'
```

我们可以声明一个运算符是非结合性的二目运算符(即这个运算符的两次出现不能合并到一起),方法如下:

```
%nonassoc '<'
```

词法单元的优先级是根据它们在声明部分的出现顺序而定的。优先级最低的词法单元最先出现。同一个声明中的词法单元具有相同的优先级。因此,图 4-59 中的声明

```
%right UMINUS
```

赋予词法单元 UMINUS 的优先级要高于前面五个终结符号的优先级。

除了给各个终结符号赋予优先级,Yacc 也可以给和某个冲突相关的各个产生式赋予优先级和结合性,来解决移入/归约冲突。如果它必须在移入一个输入符号 a 和按照 $A\rightarrow\alpha$ 进行归约之间进行选择,那么当这个产生式的优先级高于 a 的优先级时,或者当两者的优先级相同但产生式是左结合的时,Yacc 就选择归约;否则就选择移入动作。

通常,一个产生式的优先级被设定为它的最右终结符号的优先级。在大多数情况下,这是一个明智的选择。比如,给定产生式

$$E\rightarrow E + E \mid E * E$$

我们将在向前看符号为 + 时按照 $E\rightarrow E + E$ 进行归约,因为产生式体中的 + 和这个向前看符号具有相同的优先级,且它是左结合的。在向前看符号为 * 时,我们将选择移入,因为这个向前看符号的优先级高于产生式体中 + 的优先级。

在那些最右终结符号不能为产生式提供正确优先级的情况下,我们可以在产生式后增加一个标记

```
%prec 〈终结符号〉
```

来指明该产生式的优先级。此时这个产生式的优先级和结合性将和这个终结符号相同,而这个终结符号的优先级和结合性应该在声明部分定义。Yacc 不会报告那些已经使用这个优先级/结合性机制解决了的移入/归约冲突。

这里的“终结符号”可以仅仅作为一个占位符,就像图 4-59 中的 UMINUS 那样。这个终结符号不会被词法分析器返回,声明它的目的仅仅是为了定义一个产生式的优先级。在图 4-59 中,声明

```
%right UMINUS
```

赋予词法单元 UMINUS 一个高于 * 和/的优先级。在翻译规则部分,产生式

```
expr  : '-' expr
```

后面的标记

```
%prec UMINUS
```

使得这个产生式中的单目减运算符具有比其他运算符更高的优先级。

4.9.3 用 Lex 创建 Yacc 的词法分析器

Lex 的作用是生成可以和 Yacc 一起使用的词法分析器。Lex 库 ll 将提供一个名为 `yylex( )` 的驱动程序。Yacc 要求它的词法分析器的名字为 `yylex( )`。如果用 Lex 来生成词法分析器,那么我们可以将 Yacc 规约的第三部分的例程 `yylex( )` 替换为语句

```
#include "lex.yy.c"
```

并令每个 Lex 动作都返回 Yacc 已知的终结符号。通过使用语句 `#include "lex.yy.c"`,程序 `yylex` 能够访问 Yacc 定义的词法单元名字,因为 Lex 的输出文件是作为 Yacc 的输出文件 `y.tab.c` 的一部分被编译的。

在 UNIX 系统中,如果 Lex 规约存放在文件 `first.l` 中,且 Yacc 规约在 `second.y` 中,我

们可以使用命令

```
lex first.l
yacc second.y
cc y.tab.c -ly -ll
```

来得到想要的翻译器。

```
number    [0-9]+\.?|[0-9]*\.[0-9]+
%%
[ ]       { /* skip blanks */ }
{number}  { sscanf(yytext, "%lf", &yylval);
            return NUMBER; }
\n|.      { return yytext[0]; }
```

图 4-60 图 4-59 中的 yylex 的 Lex 规约

图 4-60 中的 Lex 规约可以用在图 4-59 中需要词法分析器的地方。最后的表示“任意字符”的模式必须被写作 \n|.，因为在 Lex 中，点(.)表示除了换行符之外的任意字符。

4.9.4 Yacc 中的错误恢复

Yacc 的错误恢复使用了错误产生式的形式。首先，用户定义了哪些“主要”非终结符号将具有相关的错误恢复动作。通常的选择是非终结符号的某个子集，包括那些用于生成表达式、语句、块和函数的非终结符号。然后，用户在文法中加入形如 $A \rightarrow$ **error** α 的错误产生式，其中 A 是一个主要非终结符号，α 是一个可能为空的文法符号串；**error** 是 Yacc 的一个保留字。Yacc 把这样的错误产生式当作普通产生式，根据这个规约生成一个语法分析器。

然而，当 Yacc 生成的语法分析器碰到一个错误时，它就以一种特殊的方法来处理那些对应项集包含错误产生式的状态。当碰到一个错误时，Yacc 就会从它的栈中不断弹出符号，直到它碰到一个满足如下条件的状态：该状态对应的项集包含一个形如 $A \rightarrow \cdot$ **error** α 的项。然后语法分析器就好像在输入中看到了 **error**，将虚构的词法单元 **error** 移入栈中。

当 α 为 ϵ 时，语法分析器立刻就执行一次归约到 A 的动作，并调用和产生式 $A \rightarrow$ **error** 相关的语义动作(这可能是一个用户定义的错误恢复例程)。然后语法分析器抛弃一些输入符号，直到它找到某个使它可以继续进行正常的语法分析的符号为止。

如果 α 不为空，Yacc 将向前跳过一些输入符号，寻找可以被归约为 α 的子串。如果 α 全部由终结符号组成，那么它就在输入中寻找这个终结符号串，并将它们移入到栈中进行“归约”。此时，语法分析器栈的顶部是 **error** α。然后语法分析器将把 **error** α 归约为 A，并继续进行正常的语法分析。

比如，一个形如

$$stmt \rightarrow \textbf{error} \ ;$$

的错误产生式规定语法分析器在碰到一个错误的时候要跳到下一个分号之后，并假装已经找到了一个语句。这个错误产生式的语义例程不需要处理输入，而是可以直接生成诊断消息并做出一些处理，比如设置一个标志来禁止生成目标代码。

例 4.70 图 4-61 在图 4-59 所示的 Yacc 桌上计算器中增加了错误产生式

```
lines : error '\n'
```

这个错误产生式使得这个桌上计算器在输入中发现一个语法错误时停止正常的语法分析工作。当碰到错误时，桌上计算器的语法分析器开始从它的栈中弹出符号，直到它在栈中发现一个在输入为 **error** 时执行移入动作的状态。状态 0 就是这样的一个状态(在这个例子里面，它是唯一一个这样的状态)，因为它的项包括了

```
lines→ · error '\n'
```

同时，状态 0 总是在栈的底部。语法分析器将词法单元 **error** 移入栈中，然后向前跳过输入符号，直到它发现一个换行符为止。此时，语法分析器将换行符移入到栈中，将 **error** '\n'归约为 *lines*，并发出诊断消息“请重新输入前一行”。专门的 Yacc 例程 yyerrok 将语法分析器的状态重新设置为正常操作模式。 □

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double  /* double type for Yacc stack */
%}
%token NUMBER

%left '+' '-'
%left '*' '/'
%right UMINUS
%%

lines : lines expr '\n'   { printf("%g\n", $2); }
      | lines '\n'
      | /* empty */
      | error '\n' { yyerror("reenter previous line:");
                     yyerrok; }
      ;
expr  : expr '+' expr     { $$ = $1 + $3; }
      | expr '-' expr     { $$ = $1 - $3; }
      | expr '*' expr     { $$ = $1 * $3; }
      | expr '/' expr     { $$ = $1 / $3; }
      | '(' expr ')'      { $$ = $2; }
      | '-' expr  %prec UMINUS { $$ = - $2; }
      | NUMBER
      ;
%%
#include "lex.yy.c"
```

图 4-61 带有错误恢复的桌面计算器

4.9.5 4.9 节的练习

! **练习 4.9.1**：编写一个 Yacc 程序。它以布尔表达式(如练习 4.2.2(7)中的文法所描述的)作为输入，并计算出这个表达式的值。

! **练习 4.9.2**：编写一个 Yacc 程序。它以列表(如练习 4.2.2(5)中的文法所定义的，但是其元素可以是任意的单个字符，而不仅仅是 a)作为输入，并输出这个列表的线性表示，即这些元素的单一列表，并且元素顺序和它们在输入中的顺序相同。

! **练习 4.9.3**：编写一个 Yacc 程序。它的功能是说明输入是否一个回文(即向前和向后读都一样的字符序列)。

!! **练习 4.9.4**：编写一个 Yacc 程序。它以正则表达式(如练习 4.2.2(4)中文法的定义的，但是参数可以是任意字符，而不仅仅是 a)作为输入，并输出一个能够识别相同语言的不确定有穷自动机的转换表。

4.10 第 4 章总结

- *语法分析器*。语法分析器的输入是来自词法分析器的词法单元序列。它将词法单元的名字作为一个上下文无关文法的终结符号。然后，语法分析器为它的词法单元输入序列构造出一棵语法分析树。可以象征性地构造这棵语法分析树(即仅仅遍历相应的推导步骤)，也可以显式生成分析树。
- *上下文无关文法*。一个文法描述了一个终结符号集合(输入)，另一个非终结符号集合(表示语法构造的符号)和一组产生式。每个产生式说明了如何从一些部件构造出某个非终结符号所代表的符号串。这些部件可以是终结符号，也可以是另外一些非终结符号所代表的串。一个产生式由头部(将被替换的非终结符号)和产生式体(用来替换的文法符号串)组成。

- 推导。从文法的开始非终结符号出发，不断将某个非终结符号替换为它的某个产生式体的过程称为推导。如果总是替换最左(最右)的非终结符号，那么这个推导就称为最左推导(最右推导)。
- 语法分析树。一棵语法分析树是一个推导的图形表示。在推导中出现的每一个非终结符号都在树中有一个对应结点。一个结点的子结点就是在推导中用来替换该结点对应的非终结符号的文法符号串。在同一终结符号串的语法分析树、最左推导、最右推导之间存在一一对应关系。
- 二义性。如果一个文法的某些终结符号串有两棵或多棵语法分析树，或者等价地说有两个或多个最左推导/最右推导，那么这个文法就称为二义性文法。在实践中的大多数情况下，我们可以对一个二义性文法进行重新设计，使它变成一个描述相同语言的无二义性文法。然而，有时使用二义性文法并应用一些技巧可以得到更加高效的语法分析器。
- 自顶向下和自底向上语法分析。语法分析器通常可以按照它们的工作方式分为自顶向下的(从文法的开始符号出发，从顶部开始构造语法分析树)和自底向上的(从构成语法分析树叶子结点的终结符号串开始，从底部开始构造语法分析树)。自顶向下的语法分析器包括递归下降语法分析器和 LL 语法分析器，而最常见的自底向上语法分析器是 LR 语法分析器。
- 文法的设计。和自底向上语法分析器使用的文法相比，适合进行自顶向下语法分析的文法通常较难设计。我们必须要消除文法的左递归，即一个非终结符号推导出以这个非终结符号开头的符号串的情况。我们还必须提取左公因子——也就是对同一个非终结符号的具有相同的产生式体前缀的多个产生式进行分组。
- 递归下降语法分析器。这些分析器对每个非终结符号使用一个过程。这个过程查看它的输入并确定应该对它的非终结符号应用哪个产生式。相应产生式体中的终结符号在适当的时候和输入中的符号进行匹配，而产生式体中的非终结符号则引发对它们的过程的调用。当选择了错误的产生式时，有可能需要进行回溯。
- LL(1)语法分析器。对于一个文法，如果只需要查看下一个输入符号就可以选择正确的产生式来扩展一个给定的非终结符号，那么这个文法就称为是 LL(1)的。这类文法允许我们构造出一个预测语法分析表。对于每个非终结符号和每个向前看符号，这个表指明了应该选择哪个产生式。在某些或所有没有合法产生式的空条目中放置错误处理例程有助于实现错误恢复。
- 移入 - 归约语法分析技术。自底向上语法分析器一般按照如下方式运行：根据下一个输入符号(向前看符号)和栈中的内容，选择是将下一个输入移入栈中，还是将栈顶部的某些符号进行归约。归约步骤将栈顶部的一个产生式体替换为这个产生式的头。
- 可行前缀。在移入 - 归约语法分析过程中，栈中的内容总是一个可行前缀——也就是某个最右句型的前缀，且这个前缀的结尾不会比这个句型的句柄的结尾更靠右。句柄是在这个句型的最右推导过程中在最后一步加入此句型中的子串。
- 有效项。在一个产生式的体中某处加上一个点就得到一个项。一个项对某个可行前缀有效的条件是该项的产生式被用来生成该可行前缀对应的句型的句柄，且这个可行前缀中包括项中位于点左边的所有符号，但是不包含点右边的任何符号。
- LR 语法分析器。每一种 LR 语法分析器都首先构造出各个可行前缀的有效项的项集(称为 LR 状态)，并且在栈中跟踪每个可行前缀的状态。有效项集合引导语法分析器做出移入 - 归约决定。如果项集中某个有效项的点在产生式体的最右端，那么我们就进行归约；如果下一个输入符号出现在某个有效项的点的右边，我们就会把向前看符号移入栈中。
- 简单 LR 语法分析器。在一个 SLR 语法分析器中，我们按照某个点在最右端的有效项进行归约的条件是：向前看符号能够在某个句型中跟在该有效项对应的产生式的头符号的后面。如果没有语法分析动作冲突，那么这个文法就是 SLR 的，就可以应用这个方法。所谓

没有语法分析动作冲突，就是说对于任意项集和任意向前看符号，都不存在两个要归约的产生式，也不会同时存在归约或移入的可选动作。

- 规范 LR 语法分析器。这是一种更复杂的 LR 语法分析器。它使用的项中增加了一个向前看符号集合。当应用这个产生式进行归约时，下一个输入符号必须在这个集合中。只有当存在一个点在最右端的有效项，并且当前的向前看符号是这个项允许的向前看符号之一时，我们才可以决定按照这个项的产生式进行归约。一个规范 LR 语法分析器可以避免某些在 SLR 语法分析器中出现的分析动作冲突，但是它的状态常常会比同一个文法的 SLR 语法分析器的状态更多。
- 向前看 LR 语法分析器。LALR 语法分析器同时具有 SLR 语法分析器和规范 LR 语法分析器的很多优点。它将具有相同核心(忽略了相关向前看符号集合之后的项的集合)的状态合并到一起。因此，它的状态数量和 SLR 语法分析器的状态数量相同，但是在 SLR 语法分析器中出现的某些语法分析动作冲突不会出现在 LALR 语法分析器中。LALR 语法分析器是实践中经常选择的方法。
- 二义性文法的自底向上语法分析。在很多重要的场合下，比如对算术表达式进行语法分析时，我们可以使用二义性文法，并利用一些附加的信息，比如运算符的优先级，来解决移入和归约之间的冲突，或者两个不同产生式之间的归约冲突。这样，LR 语法分析技术就被扩展应用于很多二义性文法中。
- Yacc。语法分析器生成工具 Yacc 以一个(可能的)二义性文法以及冲突解决信息作为输入，构造出 LALR 状态集合。然后，它生成一个使用这些状态来进行自底向上语法分析的函数。该函数在执行每一个归约动作时都会调用和相应产生式关联的函数。

4.11 第 4 章参考文献

上下文无关文法的形式化表示是作为自然语言研究的一部分由 Chomsky[5]提出的。在两个早期语言的语法描述中也使用了这种思想。这两个语言是 Backus 的 Fortran[2]和 Naur 的 Algol 60[26]。学者 Panini 也在公元前 400 到 200 年之间发明了一种等价的语法表示方法，用来描述梵语文法的规则。

文法二义性现象最早是由 Cantor[4]和 Floyd[13]观察到的。Chomsky 范式(练习 4.4.8)的思想来自[6]。[17]中总结了上下文无关文法的理论。

递归下降语法分析技术是早期编译器(比如[16])和编译器编写系统(比如 META[28]和 TMG[25])所选择的方法。LL 文法由 Lewis 和 Stearns[24]引入。练习 4.4.5 中的递归下降方法的线性时间模拟方法来自[3]。

由 Floyd[14]提出的最早的一种语法分析技术考虑了运算符的优先级问题。Wirth 和 Weber[29]将这个思想推广到了语言中不包含运算符的部分。现在已经很少使用这些技术了，但是它们可以被看作是使 LR 分析技术取得进展的先驱技术。

LR 语法分析器是由 Knuth[22]引入的，该著作首先给出了规范-LR 语法分析表。因为这个语法分析表要比当时常用计算机的主存大，所以这个方法被认为不可行的，直到 Korenjak[23]给出了一个方法来为典型的程序设计语言生成适当大小的语法分析表。DeRemer 发明了现在使用的 LALR[8]和 SLR[9]方法。为二义性文法构造 LR 语法分析表的方法来自[1]和[12]。

Jonhson 的 Yacc 很快证明了使用 LALR 语法分析器生成工具来为编译器产品生成语法分析器的可行性。Yacc 语法分析器生成工具的使用手册可以在[20]中找到。在[10]中描述了 Yacc 的开源版本 Bison。一个类似的基于 LALR 技术的语法分析器生成工具被称为 CUP[18]，它支持用 Java 编写的语义动作。自顶向下语法分析器生成工具包括 Antlr[27]和 LLGen。Antlr 是一个递归下降语法分析器生成工

具，它接受以 C ++ 、Java 或 C#编写的语义动作。LLGen 是一个基于 LL[1]的生成工具。

Dain[7]给出了一个关于语法错误处理的文献列表。

练习 4.4.9 中描述的通用动态规划语法分析算法是由 J. Cocke(未发表)和 Younger[30]以及 Kasami[21]各自独立发明的，因此被命名为“CYK 算法”。Earley[11]还发明了一种更加复杂的通用算法，它以表格的方式给出一个给定输入的各个子串的 LR-项。虽然这个算法在一般情况下的复杂度是 $O(n^3)$，但是对于无二义性文法，它的复杂度只有 $O(n^2)$。

1. Aho, A. V., S. C. Johnson, and J. D. Ullman, “Deterministic parsing of ambiguous grammars,” *Comm. ACM* **18**:8 (Aug., 1975), pp. 441–452.

2. Backus, J.W, “The syntax and semantics of the proposed international algebraic language of the Zurich-ACM-GAMM Conference,” *Proc. Intl. Conf. Information Processing*, UNESCO, Paris, (1959) pp. 125–132.

3. Birman, A. and J. D. Ullman, “Parsing algorithms with backtrack,” *Information and Control* **23**:1 (1973), pp. 1–34.

4. Cantor, D. C., “On the ambiguity problem of Backus systems,” *J. ACM* **9**:4 (1962), pp. 477–479.

5. Chomsky, N., “Three models for the description of language,” *IRE Trans. on Information Theory* **IT-2**:3 (1956), pp. 113–124.

6. Chomsky, N., “On certain formal properties of grammars,” *Information and Control* **2**:2 (1959), pp. 137–167.

7. Dain, J., “Bibliography on Syntax Error Handling in Language Translation Systems,” 1991. Available from the `comp.compilers` newsgroup; see `http://compilers.iecc.com/comparch/article/91-04-050` .

8. DeRemer, F., “Practical Translators for LR(k) Languages,” Ph.D. thesis, MIT, Cambridge, MA, 1969.

9. DeRemer, F., “Simple LR(k) grammars,” *Comm. ACM* **14**:7 (July, 1971), pp. 453–460.

10. Donnelly, C. and R. Stallman, “Bison: The YACC-compatible Parser Generator,” `http://www.gnu.org/software/bison/manual/` .

11. Earley, J., “An efficient context-free parsing algorithm,” *Comm. ACM* **13**:2 (Feb., 1970), pp. 94–102.

12. Earley, J., “Ambiguity and precedence in syntax description,” *Acta Informatica* **4**:2 (1975), pp. 183–192.

13. Floyd, R. W., “On ambiguity in phrase-structure languages,” *Comm. ACM* **5**:10 (Oct., 1962), pp. 526–534.

14. Floyd, R. W., “Syntactic analysis and operator precedence,” *J. ACM* **10**:3 (1963), pp. 316–333.

15. Grune, D and C. J. H. Jacobs, “A programmer-friendly LL(1) parser generator,” *Software Practice and Experience* **18**:1 (Jan., 1988), pp. 29–38. See also `http://www.cs.vu.nl/~ceriel/LLgen.html` .

16. Hoare, C. A. R., "Report on the Elliott Algol translator," *Computer J.* **5**:2 (1962), pp. 127–129.

17. Hopcroft, J. E., R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Boston MA, 2001.

18. Hudson, S. E. et al., "CUP LALR Parser Generator in Java," Available at `http://www2.cs.tum.edu/projects/cup/` .

19. Ingerman, P. Z., "Panini-Backus form suggested," *Comm. ACM* **10**:3 (March 1967), p. 137.

20. Johnson, S. C., "Yacc — Yet Another Compiler Compiler," Computing Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, 1975. Available at `http://dinosaur.compilertools.net/yacc/` .

21. Kasami, T., "An efficient recognition and syntax analysis algorithm for context-free languages," AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA, 1965.

22. Knuth, D. E., "On the translation of languages from left to right," *Information and Control* **8**:6 (1965), pp. 607–639.

23. Korenjak, A. J., "A practical method for constructing LR(k) processors," *Comm. ACM* **12**:11 (Nov., 1969), pp. 613–623.

24. Lewis, P. M. II and R. E. Stearns, "Syntax-directed transduction," *J. ACM* **15**:3 (1968), pp. 465–488.

25. McClure, R. M., "TMG — a syntax-directed compiler," *PROO. 20th ACM Natl. Conf.* (1965), pp. 262–274.

26. Naur, P. et al., "Report on the algorithmic language ALGOL 60," *Comm. ACM* **3**:5 (May, 1960), pp. 299–314. See also *Comm. ACM* **6**:1 (Jan., 1963), pp. 1–17.

27. Parr, T., "ANTLR," `http://www.antlr.org/` .

28. Schorre, D. V., "Meta-II: a syntax-oriented compiler writing language," *Proc. 19th ACM Natl. Conf.* (1964) pp. D1.3-1–D1.3-11.

29. Wirth, N. and H. Weber, "Euler: a generalization of Algol and its formal definition: Part I," *Comm. ACM* 9:1 (Jan., 1966), pp. 13–23.

30. Younger, D.H., "Recognition and parsing of context-free languages in time n^3," *Information and Control* **10**:2 (1967), pp. 189–208.