编译原理

2020年4月

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

编译原理
第四章

上海交通大学

张冬茉

Email:zhang-dm@cs.sjtu.edu.cn

2020年3月

第4章 语法分析

本章将介绍编译器中使用的典型语法分析方法。主要包括:

- 介绍基本概念
- 讨论适合于手工实现的技术,
- 介绍自动生成工具中所使用的算法
- 扩展语法分析方法，使之能从常见的错误中得以恢复。

4.1 引论

上下文无关文法可以给出一个程序设计语言的精确易懂的语法描述，本节将通过一个算术表达式的典型文法，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结构。

4.1.1 语法分析器的作用

在本书的编译器模型中，语法分析器接受词法分析器提供的记号串，检查它们是否能由源程序语言的文法产生，如图4-1所示。我们希望词法分析器能用易于理解的方式提示语法错误信息，并能从常见的错误中恢复过来，以便后面的输入能继续处理下去。

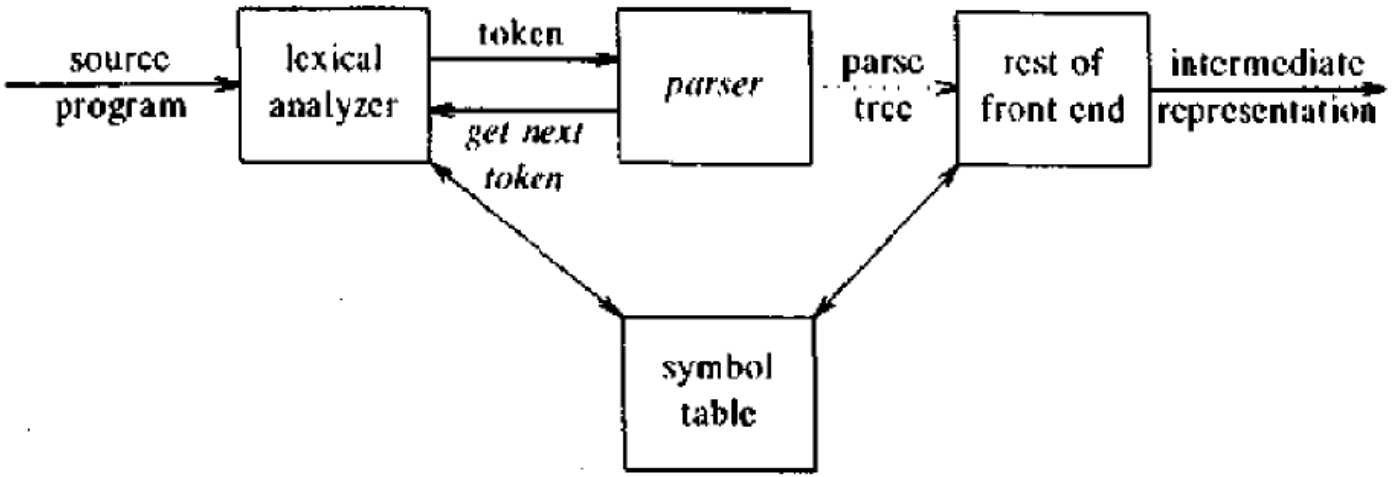


Fig. 4.1. Position of parser in compiler model.

- 典型的文法的语法分析器有三类
  - 通用的语法分析方法
  - 自顶向下的语法分析方法
  - 自底向上的语法分析方法
- 本节的剩余部分将考虑语法错误的性质和错误恢复的一般策略。我们在讨论各种语法分析方法时将详细介绍两种错误恢复策略:
  - 应急模式恢复策略
  - 短语级恢复策略

4.1.2 代表性文法

- 考虑下面的算术表达式文法：

E -> E + T | T

T -> T * F | F (4.1)

F -> (E) | id

消除E和T的直接左递归，可以得到：

E -> TE’

E’ -> +TE’ | ϵ

T -> FT’ (4.2)

T’ -> *FT’ | ϵ

F -> (E) | id

- 考虑下面的算术表达式文法：

E -> E + E | E * E | (E) | id (4.3)

4.1.3 语法错误的处理

- 我们知道程序可能包含不同级别的错误。下边是一些程序错误的示例:
  - 词法错误，如标识符、关键字或操作符的拼写错误。
  - 语法错误，如算术表达式的括号不配对。
  - 语义错误，如操作符作用于不相容的操作数。
  - 逻辑错误，如无限的递归调用。

- 语法分析器中出错处理程序的基本目标是:
  - 清楚而准确地报告错误的出现。
  - 迅速地从每个错误中恢复过来，以便能继续检查后面的错误。
  - 不能过分降低正确程序的处理速度。

- 有些分析方法，如LL方法和LR方法，可以尽快地检查出语法错误。更准确地说，它们具有可行前缀性（viable-prefix property），即当它们一旦发现一个输入字符串的前缀不是该语言任何字符串的前缀时就能检测出错误。
- 60%的被编译程序语法和语义都是正确的。即使有错误出现，错误的种类也相当少。80%的出错语句只含一个错误，只有13%的出错语句含有两个错误。多数错误是微不足道的，90%的错误是单个记号错。
- 多数错误可简单地分为以下几类：60%是标点符号错，20%是操作符或运算对象错，15%是关键字错，剩下的5%是其他类型的错误。多数标点号错都属于分号的不正确使用。

- 出错处理程序应该怎样报告错误呢？很多编译器普遍采用的方法是：显示错处的程序行，用指针指出检测到错误的位置。如果能够知道实际错误可能是什么，编译器还会显示附带的诊断信息，如“此处遗漏了分号”等。

- 一但检测出错误，语法分析器将如何恢复这个错误呢？有很多一般性的策略，但没有哪种策略占绝对优势。

4.1.4 错误恢复策略

我们将主要介绍下面几种策略:

- 应急模式恢复策略。最容易实现。当发现错误时，语法分析器开始抛弃输入记号，每次抛弃一个记号，直到发现某个指定的同步记号为止。比较简单，不会陷入死循环。对一个语句中出现错误较少时比较合适。
- 短语级恢复策略。发现错误时，语法分析器对剩余的输入字符串作局部纠正，即用一个能使语法分析器继续工作的字符串来代替剩余输入的前缀。可能引起死循环。被用于自顶向下语法分析中。主要缺点是难以应付实际错误出现在诊断点之前的情况。

- 出错产生式策略。如果对经常遇到的错误有很清楚地了解，我们可以扩充语言的文法，增加产生错误结构的产生式。然后用由这些错误产生式扩充的文法构造语法分析器。

- 全局纠正策略。我们总是希望一个理想的编译器在处理不正确的输入字符串时尽可能少的改动。如果给定错误输入串x和文法G，这些算法会发现y的一棵分析树，以便使用最少的符号插入、删除和修改操作把x变换成正确的输入字符串y。不幸的是，实现这些算法的时间和空间开销太大，目前只作理论上的探讨。

4.2 上下文无关文法

- 在第二章我们已经介绍了上下文无关文法（简称文法）的例子，

例如：if-else语句的构造规则可以表达为：

stmt -> if (*expr*) *stmt* else *stmt*

这里，箭头“->”可以读作“可以具有形式”。这样的规则称为产生式(production)。在一个产生式中，像关键字if，else和括号这样的词法元素称为记号(token)或终结符(terminal)，像*expr*和*stmt*这样的变量表示一个记号序列，并称之为非终结符(nonterminal)。

4.2 .1 上下文无关文法的正式定义

- 上下文无关文法由终结符、非终结符、开始符号和产生式组成。其中：
  - **终结符是组成字符串的基本符号**。在讨论程序设计语言的文法时，“记号”和“终结符”是同义词。
  - **非终结符是表示字符串集合的语法变量**。非终结符所定义的字符串集合有助于定义该文法所产生的语言。非终结符强加给语言一种层次结构，这种层次结构对语法分析和翻译都非常有用。
  - **有一个非终结符被定义为开始符号**。
  - **文法的产生式说明了终结符和非终结符组合成串的方式**。

例4.1

- 具有下述产生式的文法定义了简单的算术表达式。

$$
\begin{aligned}
expr &\rightarrow expr\ op\ expr \\
expr &\rightarrow (\ expr\ ) \\
expr &\rightarrow -\ expr \\
expr &\rightarrow \mathbf{id} \\
op &\rightarrow + \\
op &\rightarrow - \\
op &\rightarrow * \\
op &\rightarrow / \\
op &\rightarrow \uparrow
\end{aligned}
$$

- 在该文法中，终结符包括id、+、-、*、/、↑、(和)，非终结符包括expr和op，expr是开始符号。

4.2.2 符号的使用约定

- 1、下列符号是终结符：
  - 字母表中比较靠前的小写字母，如a、b、c等
  - 操作符号，如+、-等
  - 标点符号，如括号、逗号等
  - 数字0，1，……，9
  - 黑体串，如id、if等
- 2、下列符号是非终结符：
  - 字母表中比较靠前的大写字母，如A、B、C等
  - 字母S，它常常代表开始符号
  - 小写斜体名字，如*expr*、*stmt*等

- 3、字母表中比较靠后的大写字母，如X、Y、Z等，表示文法符号，也就是说，可以是非终结符也可以是终结符。
- 4、字母表中比较靠后的小写字母，如u、v、……、z等，表示终结符号的串。
- 5、小写希腊字母，如α、β、γ等，表示文法符号的串。
- 6、如果$A\rightarrow\alpha_1$、$A\rightarrow\alpha_2$、……、$A\rightarrow\alpha_k$是所有以A为左部的产生式（称为A产生式），则可以把它们写成：

  $$A\rightarrow\alpha_1|\alpha_2|\ldots\ldots|\alpha_k,$$

  我们将α_1、α_2、……、α_k称为A的候选式。
- 7、除非特别说明，否则第一个产生式左部的符号是开始符号。

例4.2

使用上述简写约定，例4.1的文法可以写成：

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

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid \text{id}$

根据上述约定，E,T和F是非终结符，E是开始符，其他符号都是终结符。

4.2.3 推导

- 考虑下面的算术表达式文法：

  E -> E + E | E * E | (E) | -E | id　　(4-3)

  则称：　E => -E => -(E) => -(id) 是 E 到-(id)的推导。

- 更抽象地，我们说αAβ=>αγβ，如果A->γ是产生式，而且α和β是任意的文法符号的串。如果：

  $\alpha_1 => \alpha_2 => \cdots => \alpha_n$，则说$\alpha_1$推导出$\alpha_n$。

  符号=>表示“一步推导”。通常我们用 $\overset{*}{\Rightarrow}$ 表示“零步或多步推导”，因此，

  - 对任何串α，α $\overset{*}{\Rightarrow}$ α。
  - 如果α $\overset{*}{\Rightarrow}$ β，而且β=>γ，则α $\overset{*}{\Rightarrow}$ γ。

- 类似地，我们用 $\overset{+}{\Rightarrow}$ 表示“一步到多步的推导”。当且仅当S $\overset{+}{\Rightarrow}$ w时，我们说终结符w在L(G)中。终结符串w成为G的句子。由上下文无关文法产生的语言称为上下文无关语言。如果两个产生相同的语言，则称这两个文法等价。
- 对于开始符号S的文法G，如果S $\overset{*}{\Rightarrow}$ α，则称α为G的句型，其中α可能含有非终结符。句子是不含非终结符的句型。

例4.3

- 字符串-(id+id)是文法(4-3)的句子，因为存在如下推导：

  E => -E => -(E) => -(E+E) => -(id+E) => -(id+id)　　(4-4)

- 出现在这个推导中的字符串E、-E、-(E)、…、-(id+id)都是该文法的句型。我们用E $\overset{*}{\Rightarrow}$ -(id+id)表示-(id+id)可以由E推导出来。

- 按推导长度进行归纳，我们可以证明文法(4-3)产生的语言中的每个句子都是由二元操作符+和*、一元操作符-、括号以及运算对象id组成的算术表达式。同样按算术表达式长度进行归纳，我们也可以证明这样的算术表达式都可以由文法(4-3)产生。因此，文法(4-3)正好产生所有包括二元操作符+和*、一元操作符-、括号以及操作数id的算术表达式的集合。

术语：

- 每一步都替代最左非终结符的推导，叫做最左推导。
- 为了强调α通过最左推导推导出β这一事实，我们写 $\alpha \overset{*}{\underset{lm}{\Rightarrow}} \beta$。
- 如果 $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$，则称 α 是该文法的左句型。
- 我们可以类似地定义最右推导，即每步推导都替代最右非终结符的推导。
- 最右推导有时也称为规范推导。

4.2.4 分析树和推导

- 分析树可以看成是推导的图形表示，但它不能显示出替代顺序的选择。回顾2.2节，分析树的每个内节点都标以某个非终结符A。A的子节点从左到右分别被用来替换A所使用的产生式右部的各符号标记。分析树的叶节点用非终结符或终结符来标记，它们从左到右构成一个句型，称为树的边界或果实。例如，-(id+id)的推导过程如(4-4)所示，其分析树如图4-2所示。

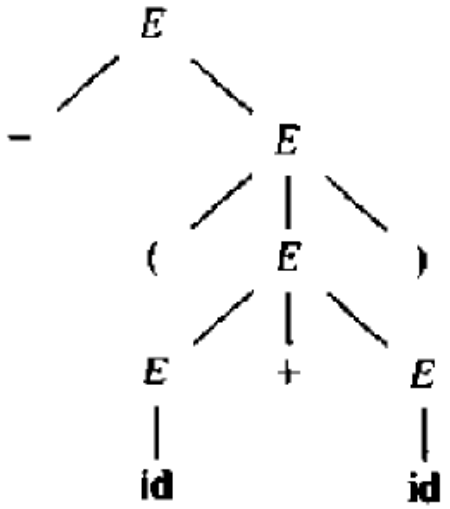


Fig. 4.2. Parse tree for −(id + id).

从该推导所构造出的分析树序列如图4-3所示。推导的第一步是E=>-E。为了模拟这一步，我们为最初的分析树的根节点E增加两个子节点，分别标记为-和E。推导第二步是-E=>-(E)，所以为第二个分析树的标记为E的叶结点增加三个子节点，分别标记为(、E和)，从而获得带有结果-(E)的第三棵树。如此继续，我们将得到第六棵树所示的这个分析树。

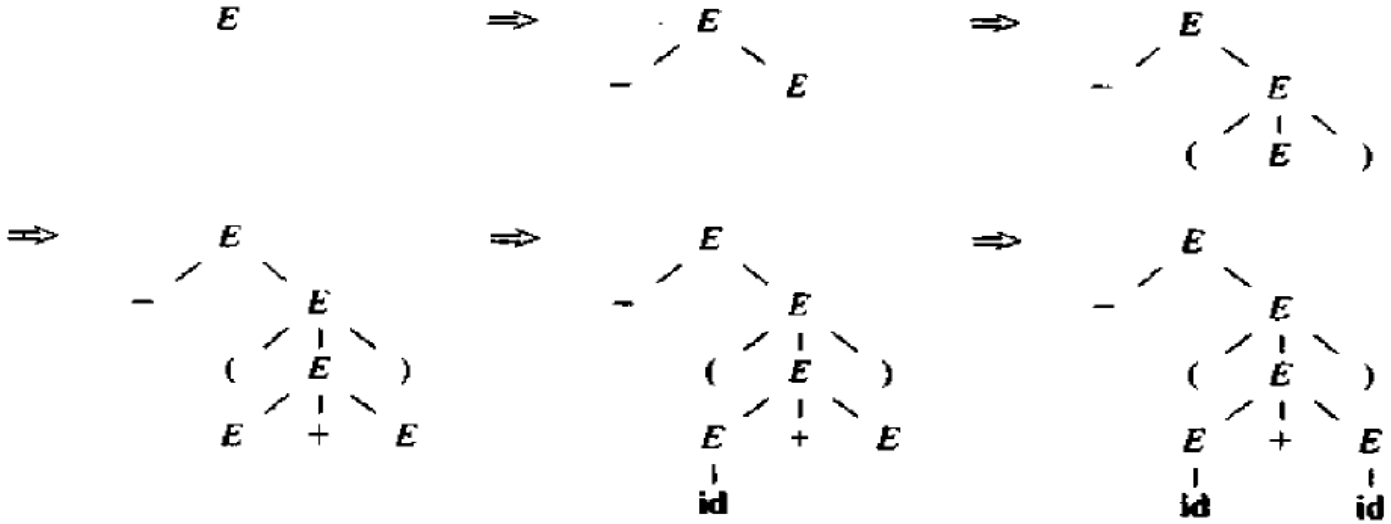


Fig. 4.3. Building the parse tree from derivation (4.4).

例4.4

- 每棵分析树都有一个与之对应的唯一的最左推导和唯一的最右推导。然而，每一个句子不一定只有一个分析树，或者说不一定只有一个最左推导或最右推导。

- 再次考虑算术表达式文法(4-3)。句子id+id*id有两个不同的推导：

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

4.2.5 二义性

- 给定一个文法G，如果L(G)中存在一个具有两棵或两棵以上分析树的句子，则称G是二义性的。我们也可以如下定义二义性：如果L(G)中存在一个具有两个或两个以上最左（或最右）推导的句子，则G是二义性文法。
- 例4.4表明算术表达式文法：

  E -> E + E | E * E | (E) | -E | id

  就是一个二义性文法。

//4.2.6 验证文法所产生的语言

对“文法G产生语言L”的证明包括两部分：

(1) 我们必须证明由G产生的每个字符串都在L中；

(2) 反之，L中的每个字符串都能由G产生。

例4.5

文法G：S -> (S)S | ϵ 能而且仅能产生所有的配对的括号串

证明L(G) = {s | s是一个配对的括号串}的方法：

（1）首先证明从S推导出的所有句子都是配对的括号串，

（2）然后证明每个配对的括号串都可以从S推导出来。

- 对推导的步数使用数学归纳法，证明从S推导出的所有句子都是配对的括号串。从S经过一步推导得出的终结符串只有空串。空串可以视为配对的括号串。显然，当推导步数为1时命题正确。
- 现在假设所有少于n步的推导所产生的句子都是配对的括号串考察一个n步最左推导。这个推导一定具有如下形式：

  S => (S)S $\overset{*}{\Rightarrow}$ (x)S $\overset{*}{\Rightarrow}$ (x)y

  由于从S推导出x和y的步数少于n步，根据归纳假设，x和y都是配对括号串。因此，(x)y也一定是配对括号串。

- 因此我们已经证明了任何从S中推导出来的串都是配对的括号串。下边，我们对括号串的长度使用数学归纳法，证明每个配对括号串都可以从S推导出来。空串是配对的括号串，并且使用产生式S -> ϵ由S推导出来。于是，长度为0的括号串可以从S推导出来。

- 假设每个长度小于2n的配对括号串都可以从S推导出来，我们来考虑长度为2n(n >= 1)的配对括号串w。可以肯定，w是由左括号开始的，令(x)是w的具有相同个数的左右括号的最短前缀。那么w可以写作(x)y，其中x和y都是配对的括号串。既然x和y的长度小于2n，由归纳假设，它们可以从S推导出来。于是，我们可以找出如下形式的推导：

  S => (S) S $\overset{*}{\Rightarrow}$ (x) S $\overset{*}{\Rightarrow}$ (x) y

  从而证明w=(x) y也能从S推导出来。

4.2.7 正规表达式和上下文无关文法的比较

我们可以用下列规则机械地把一个NFA转换成一个等价的上下文无关文法：

- 对NFA的每一个状态i，创建一个非终结符A_i
- 如果状态i遇见输入符号a转换到状态j，则引入产生式A_i -> aA_j
- 如果状态i遇见输入符号ϵ转换到状态j，则引入产生式A_i -> A_j
- 如果状态i是接受状态，则引入产生式A_i -> ϵ
- 如果状态i是开始状态，则A_i是文法的开始符号

既然正规集都是上下文无关语言，我们可能要问：为什么要用正规表达式而不用上下文无关文法来定义语言的词法？理由如下：

- 语言的词法规则通常都非常简单，不必动用强大的文法来解决
- 对于记号，正规表达式比上下文无关文法提供了更简洁且易于理解的定义
- 从正规表达式可以自动地构造出有效的词法分析器，从任何文法都很难构造词法分析器
- 把语言的语法结构分成词法和非词法两部分为编译器前端的模块划分提供了方便的途径

4.3 设计文法

文法能够描述程序设计语言的大部分语法成分，但不能描述程序设计语言的全部语法成分。语法分析以后的各编译阶段必须分析语法分析器的输出，以保证输入字符串符合语法分析器无法检查的那些规则。

4.3.1词法分析和语法分析

- 词法分析器和语法分析器的分工明确。
- 词法规则的识别可以满足快速进行词法分析的要求。
- 每种语法分析方法只能处理一种形式的文法。为了适应所选择的分析方法，我们常常不得不改写初始文法。

4.3.2 消除二义性

- 有些二义性文法可以通过改写来消除二义性。作为一个例子，我们来消除下面的“不匹配else”文法的二义性：

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

这里，other代表任何其他语句。

- 文法(4-9)是具有二义性的，因为串

  if E_1 then if E_2 then S_1 else S_2

  有两棵分析树，

如图4-6。

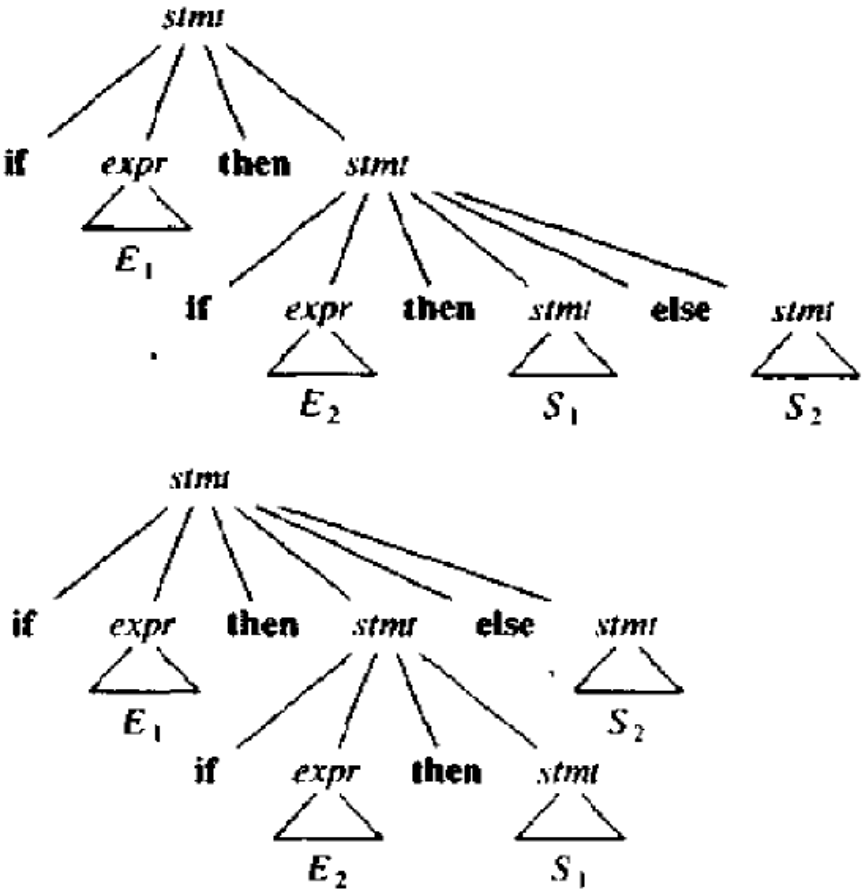


Fig. 4.6. Two parse trees for an ambiguous sentence.

例4.6

所有包含这种条件语句的程序设计语言都使用第一种分析树。

一般规则是，“每个else和前面最近的没有配对的then配对”。

这条避免二义性的规则可以直接并入文法中。

例如：可以把文法(4-9)改写成下面的无二义性文法，

其基本思想是：出现在then和else之间的语句必须是“配对”的，即它不能以一个未配对的then后面跟随任意的非else语句结束，于是else被迫与这个未配对的then配对。配对的语句是一个不包含不配对语句的if-then-else语句或者任何非条件语句。

改写如下：

stmt → *matched_stmt*
| *unmatched_stmt*

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

unmatched_stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*

4.3.3 消除左递归

- 左递归的定义：如果文法G具有一个非终结符A使得对某个字符串α存在推导A $\overset{+}{\Rightarrow}$ Aα，则称G是左递归的文法。
- 消除左递归的原因：在最左推导中会造成无限推导的情况。
- 消除左递归的方法：对于文法中形如：$A \to A\alpha \mid \beta$ 直接左递归形式的产生式

  其中β不以A为前缀。它产生的符号串为βα*结构。引进非终结符A'，令：

$$A \to \beta A'$$
$$A' \to \alpha A' \mid \varepsilon$$

它同样产生βα*形符号串，所以与等价，但是它不是左递归的。

例4.7

- 考虑下面的算术表达式文法：

E -> E + T | T

T -> T * F | F (4-10)

F -> (E) | id

消除E和T的直接左递归，可以得到：

E -> TE’

E’ -> +TE’ | ϵ

T -> FT’ (4-11)

T’ -> *FT’ | ϵ

F -> (E) | id

消除左递归的一般方法

无论有多少A产生式，我们都可以用下面的技术来消除直接左递归。

首先，把A产生式放在一起：

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

其中：每个βi都不以A开头。

然后用下面的产生式代替A产生式：

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

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$

但不能消除包括两步或多步推导的左递归，如：

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \epsilon$ (4-12)

算法4.1 消除左递归

输入：无循环推导和ϵ产生式的文法G

输出：与G等价的无左递归文法

方法：对文法G应用图4-7的算法。

注意，得到的非左递归文法可能包含有ϵ产生式。

1. Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2. **for** $i := 1$ **to** n **do begin**
   for $j := 1$ **to** $i - 1$ **do begin**
   replace each production of the form $A_i \rightarrow A_j\gamma$
   by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$,
   where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all the current A_j-productions;
   end
   eliminate the immediate left recursion among the A_i-productions
   end

Fig. 4.7. Algorithm to eliminate left recursion from a grammar.

文法(4-12)应用消除左递归算法后的过程为：

- 令非终结符的次序是S、A。在S产生式中没有直接左递归，所以在算法第2步，对于i=1，什么也没做。i=2时，用S产生式替换A->Sd中的S，得到下面的A产生式：

  A -> Ac | Aad | bd | ϵ

消除A产生式中的直接左递归，产生下面的文法：

S -> Aa | b

A -> bdA' | A'

A' -> cA' | adA' | ϵ

4.3.5 提取左因子

算法4.2 提取左因子

输入：文法G

输出：一个等价的提取了左因子的文法

方法：对每个非终结符A，找出它的两个或更多候选式的最长公共前缀α。

如果$\alpha \neq \epsilon$，既有一个非平凡的公共前缀，则用下面的产生式代替所有A产生式$A->\alpha\beta_1 | \alpha\beta_2 | \cdots | \alpha\beta_n | \gamma$，其中γ表示所有不以α开头的候选式：

$A -> \alpha A' | \gamma$

$A' -> \beta_1 | \beta_2 | \cdots | \beta_n$

其中：A'是一个新的非终结符。反复应用这种变换，直到任一非终结符都没有两个候选式具有公共前缀为止。

例4.8

- 下面是条件语句抽象出来的文法:

S -> iEtS | iEtSeS | a

E -> b (4-13)

这里的i、t和e分别表示if、then和else，E和S表示表达式和语句。提取左因子后，该文法变为

S -> iEtSS' | a

S' -> eS | **ϵ** (4-14)

E -> b

于是，如果输入是i，我们可以将S扩展到iEtSS'，等到iEtS出现后，再决定是将S'扩展到eS还是ϵ。当然由于文法(4-13)和(4-14)都是具有二义性，所以对于输入e，我们不清楚应该选择S'的哪个候选式。

4.3.6 非上下文无关语言的结构

- 考虑抽象语言L_1={wcw | w属于(a|b)*}。
- L1是所有由c隔开的两个相同的a、b串组成的字母串集合，
- 例如：aabcaab。这个语言是检查程序中标识符的声明应先于其引用的抽象，即wcw中的第一个w表示标识符w的声明，第二个w表示它的引用。可以证明该语言不是上下文无关语言。（上下文无关文法不能描述两个相同的a、b串组成的字母串集合）

  这个例子意味着C++和Java等程序设计语言都不是上下文无关语言，因为它们要求标识符的声明先于引用，并且允许标识符任意长。
- 由于上述原因，描述C++和Java语法的文法并不定义标识符中的字符，而只用文法中id这样的记号代表所有的标识符。在这类语言的编译器中，语义分析阶段检查标识符的声明是否先于引用。

- 同理语言$L_2=\{a^nb^mc^nd^m \mid n\geqslant 1且m\geqslant 1\}$不是上下文无关语言。
- L_2是由正规表达式a*b*c*d * 所表示的语言的子集合，在它的每个句子中，a和c的个数相同，b和d的个数相同。它是“过程声明中的形参个数和过程引用的实参个数应该一致”的抽象。
- 注意，过程定义和引用的语法并不涉及到参数的个数，通常在语义分析阶段检查实参个数是否正确。

作业

对下列每一个文法和对应的串完成：

- 给出串的一个最左推导；
- 给出串的一个最右推导；
- 给出串的一棵语法分析树；

（1）S->SS+|SS*|a　　和串aa+a*

（2）S->0S1|01　　和串000111

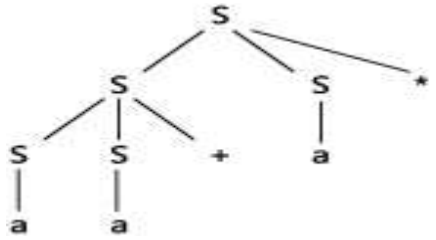
（3）S->aSbS|bSaS|ϵ　　和串aabbab

答案：

1. 最左推导：S=>SS* =>SS+ S* =>aS+ S* =>aa+ S* =>aa+ a*

   最右推导：S=>SS* =>Sa* =>SS+ a* =>Sa+ a* =>aa+ a*

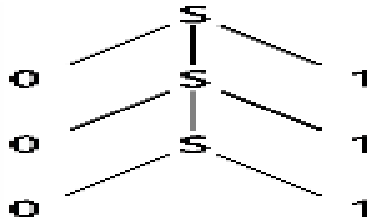
   语法分析树：



2. 最左推导：S=>0S1=>00S11=>000111

   最右推导：S=>0S1=>00S11=>000111

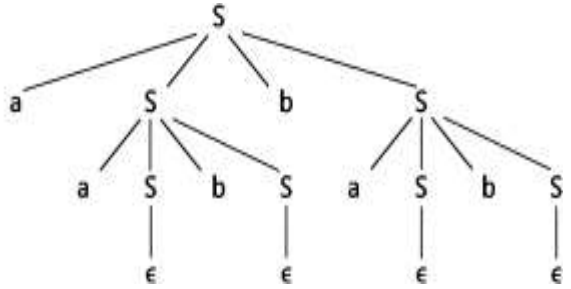
   语法分析树：



3. 最左推导：S=>aSbS=>aaSbSbS=>aabSbS=>aabbS=>aabbaSbS=>aabbabS=>aabbab

   最右推导：S=>aSbS=>aSbaSbS=>aSbaSb=>aSbab=>aaSbSbab=>aaSbbab=>aabbab

   语法分析树：



4.4 自顶向下语法分析

- 语法分析的方法通常有两类，即自上而下分析方法和自下而上分析方法，或称为自顶向下(top-down)和自底向上(bottom-up)分析方法。它们的目标都是判断一输入串是否为一合法句子。
- 自上而下分析就是能否找到从文法开始符开始的推导序列，使得推导出的句子恰为输入串。或者说，能否从根结点出发，向下生长出一棵语法分析树，其叶结点组成的句子恰为输入串。显然语法分析树的每一步生长(每一步推导)都以能否与输入串匹配为准。

4.4.1 递归下降语法分析法

递归下降分析法是一种自顶向下的分析方法，文法的每个非终结符，对应于一个递归过程。分析就是从方法开始符出发执行一组递归过程，向下推导，直到推导出句子。或者说从根结点出发，自上而下为输入串寻找一个最左匹配序列，建立一棵语法分析树。

例4.9

考虑下述文法：S -> cAd

A -> ab | a

若输入字符串为w=cad

为了自顶向下地为w建立分析树，我们首先建立只有标记为S的单个节点的树。输入指针指向w的第一个符号c。然后，我们用S的第一个产生式来扩展该树，图4-9a所示的树。

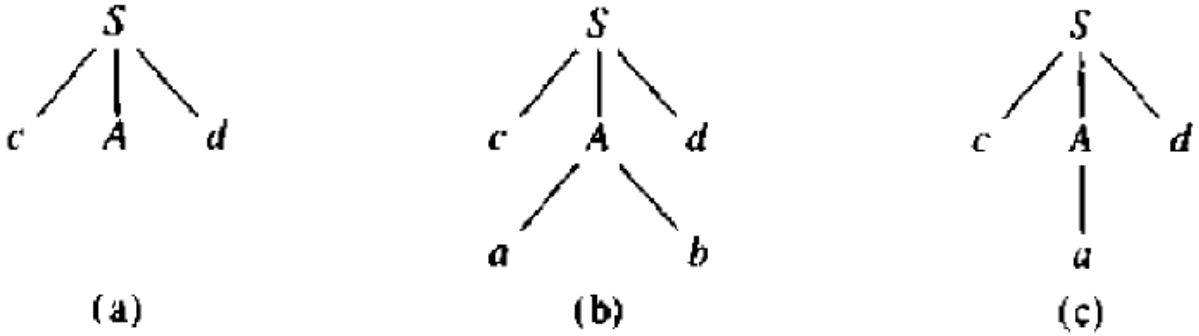


Fig. 4.9. Steps in top-down parse.

- 最左边的叶子标记为c，匹配w的第一个符号。现在，我们将输入指针移到w的第二个符号a。考虑下一个标记为A的叶子。用A的第一个候选式扩展A，得到图4-9b所示的树。现在我们已经匹配了第二个输入符号a，再将输入指针移到第三个输入符号d，把它和下一个标记为b的叶结点进行比较。因为b和d不匹配，报告失败，回到A，看是否还有别的候选式可试。
- 回到A时，我们必须将输入指针重置到第二个符号，即第一次进入A时的位置。这意味着A的程序必须将输入指针保存在一个局部变量中。现在尝试A的第二种候选式，得到图4-9c所示的分析树。叶子a匹配w的第二个符号，叶子d匹配w的第三个符号。因为已经产生了w的分析树，我们停止分析。
- 注意：(1) 左递归文法可能会使其进入无限循环。
- (2) 递归下降分析法可能需要回溯，即需要重复地扫描输入。

4.4.2 预测语法分析器

- 在许多情况下，通过仔细地编写文法，消除左递归，提取左因子，我们可以获得一个有效的文法，这个文法可以用于不带回溯的递归下降语法分析器。
- 为了构造预测语法分析器，对给定的当前输入符号a和将要扩展的非终结符A，我们必须知道，在A的所有可选产生式$A \rightarrow \alpha_1 | \alpha_2 | \cdots | \alpha_n$中，哪个候选式是唯一能推导出以a开头的串。
- 预测语法分析器能够通过观察候选式所推导出的第一个符号，确定正确的候选式。这种方法可以检测出多数程序设计语言中具有不同关键字的控制流结构。

4.4.3 预测语法分析器的状态转换图

- 词法分析器的状态转换图和预测语法分析器的状态转换图具有明显的区别。
- 对于预测语法分析器，每个非终结符都对应一个状态转换图，边上的标记是记号和非终结符。
- 记号（终结符）上的转换意味着如果该记号是下一个输入符号，就应该进行转换。
- 非终结符A上的转换是对与A对应的过程的调用。
- 为了由文法构造预测语法分析器的状态转换图，首先消除文法中的左递归，然后提取左因子，并对每个非终结符A执行如下操作：
  - **创建一个开始状态和一个终态**
  - **对每一个产生式$A->X_1X_2\cdots X_n$，创建一条从开始状态到终止状态的路径，边上的标记分别为X_1，X_2，…，X_n。**

4.4.2 预测语法分析器

- 基于状态转换图的预测分析程序试图进行终结符和输入的匹配，并且，当它经过标记为非终结符的边时，进行潜在的递归过程调用。一种非递归的实现方法是，当在状态S上有一个标记为非终结符的指向其他状态的转换时，则将状态S压入栈中，当到达该非终结符的终止状态时，将状态S弹出栈。
- 如果给定的状态转换图是确定的，即一个状态对于一个输入仅有一个转换，则上述方法是有效的。如果出现二义性，可以用下边例4.10中的方法解决。如果不能消除不确定性，我们就不能构造预测语法分析器。但我们可以构造递归下降语法分析器，用回溯的方法尝试所有可能的情况。

例4.10

图4-10给出了文法(4-11)所对用的状态转换图。

E -> TE'
E' -> +TE' | ϵ
T -> FT' (4-11)
T' -> *FT' | ϵ
F -> (E) | id

唯一的二义性在于确定是否经过ϵ边。如果我们把E'的开始状态的出边解释为：对E'的开始状态，如果下一个输入是+，则选择+上的转换，否则选择ϵ上的转换。对T'也作同样的假设，我们就消除了二义性。

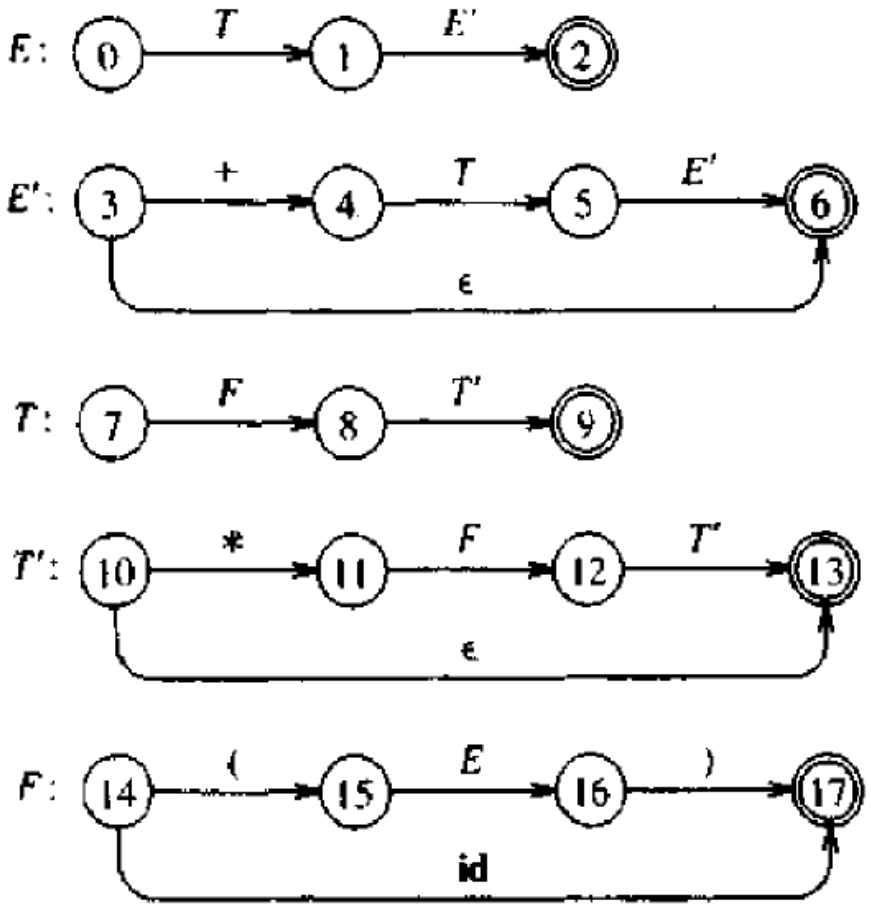


Fig. 4.10. Transition diagrams for grammar (4.11).

状态转换图的变换化简

通过图4-11的变换化简状态转换图。用C实现的与图4-12对应的预测语法分析器要比图4-10所对应的快20%~25%。

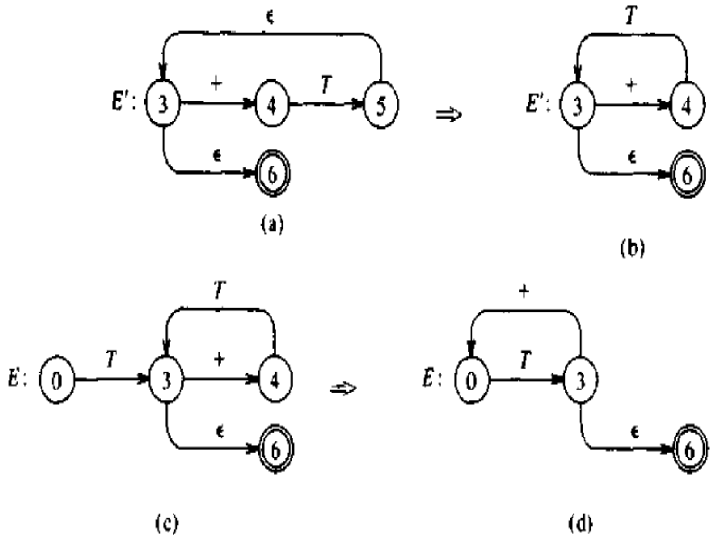


Fig. 4.11. Simplified transition diagrams.

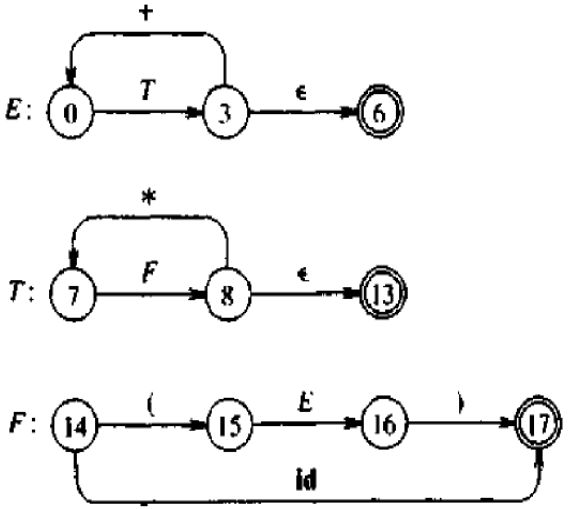


Fig. 4.12. Simplified transition diagrams for arithmetic expressions.

状态转换图的实现

```
void E()
{ T(); while(lookahead==’+’){match(‘+’);T();}

void T()
{ F();  while(lookahead==’*’ ){ match(‘*’);F();T’();}
}

void F()
{
   if(lookahead==’i’){match(‘i’);}
     else if (lookahead==’(’)
            {match(‘(’);E();if(lookahead== ‘)’)
                                {match(‘ )’);}
                                else {error();}}
          else{error();}
}
```

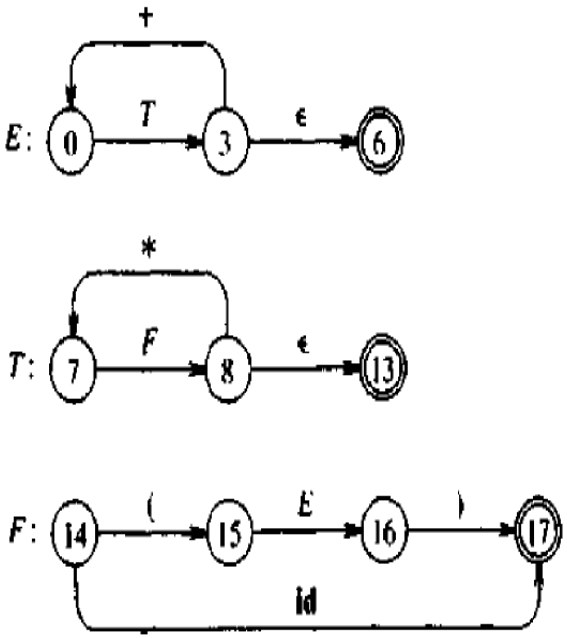


Fig. 4.12. Simplified transition diagrams for arithmetic expressions.

4.4.4 非递归的预测分析

通过显示地维护一个状态栈，而不是通过隐式的递归调用，我们可以构造非递归的预测语法分析器。预测分析的关键问题是确定用于扩展非终结符的产生式，图4-13中的非递归语法分析器通过查分析表来选取产生式。

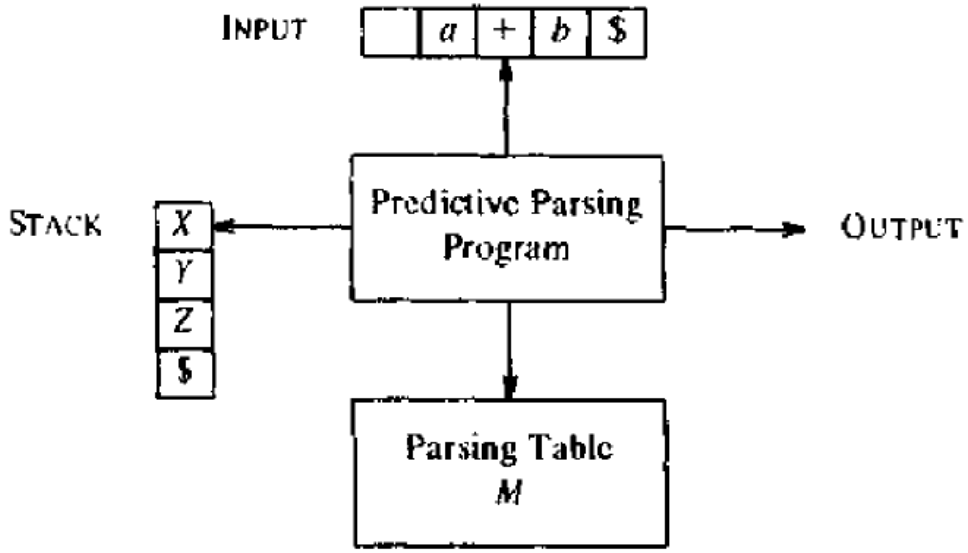


Fig. 4.13. Model of a nonrecursive predictive parser.

语法分析器工作方式

- 语法分析器由一个按如下方式工作的程序控制：程序根据栈顶当前的符号X和当前的输入符号a决定语法分析器的动作：
  - 如果X=a=$，则语法分析器宣告分析成功并停止。
  - 如果X=a≠$，则语法分析器弹出栈顶符号X，并将输入指针移到下一个输入符号上。
  - 如果X是非终结符，则程序访问分析表M的M[X,a]项。M[X,a]项是文法的一个X产生式或者是出错信息。

算法4.3 非递归的预测分析

输入：串w和文法G的分析表M。

输出：如果w属于L(G)，则输出w的最左推导，否则报告错误。

方法：开始时，语法分析器的格局是$S在栈里（其中S是G的开始符号且在栈顶），w$在输入缓存区。

图4-14是程序。

```
set ip to point to the first symbol of w$;
repeat
    let X be the top stack symbol and a the symbol pointed to by ip;
    if X is a terminal or $ then
        if X = a then
            pop X from the stack and advance ip
        else error()
    else    /* X is a nonterminal */
        if M[X, a] = X → Y1Y2 ··· Yk then begin
            pop X from the stack;
            push Yk, Yk-1, ..., Y1 onto the stack, with Y1 on top;
            output the production X → Y1Y2 ··· Yk
        end
        else error()
until X = $    /* stack is empty */
```

Fig. 4.14. Predictive parsing program.

例4.11

考虑例4.6中的文法(4-11)：

E -> TE'

E' -> +TE' | ϵ

T -> FT' (4-11)

T' -> *FT' | ϵ

F -> (E) | id

该文法预测分析表如图4-15所示。表中空白表项表示出错，非空白表项表示一个产生式。

NONTER-MINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ϵ	E'→ϵ
T	T→FT'			T→FT'		
T'		T'→ϵ	T'→*FT'		T'→ϵ	T'→ϵ
F	F→id			F→(E)		

Fig. 4.15. Parsing table M for grammar (4.11).

- 如果输入时id+id*id，预测语法分析器所做的移动如图4-16所示。
- 输入指针指向输入栏中符号串最左边的符号。语法分析器跟踪的是输入的最左推导，即产生式输出的正好是最左推导中使用的那些产生式。
- 已经扫描过的输入符号加上栈中的文法符号（从顶到底）构成该推导的左句型。

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	$E \rightarrow TE'$
$E'T'F	id + id * id$	$T \rightarrow FT'$
$E'T'id	id + id * id$	$F \rightarrow id$
$E'T'	+ id * id$	
$E'	+ id * id$	$T' \rightarrow \epsilon$
$E'T+	+ id * id$	$E' \rightarrow +TE'$
$E'T	id * id$	
$E'T'F	id * id$	$T \rightarrow FT'$
$E'T'id	id * id$	$F \rightarrow id$
$E'T'	* id$	
$E'T'F*	* id$	$T' \rightarrow *FT'$
$E'T'F	id$	
$E'T'id	id$	$F \rightarrow id$
$E'T'	$	
$E'	$	$T' \rightarrow \epsilon$
$	$	$E' \rightarrow \epsilon$

Fig. 4.16. Moves made by predictive parser on input id+id*id

4.4.5 FIRST和FOLLOW

- 构造文法G的分析表需要两个与G有关的函数FIRST和FOLLOW。
- 我们可以用这两个函数来填写G的分析表的表项。
- 由FOLLOW函数产生的记号集合还可用作紧急方式错误恢复期间的同步记号。

FIRST集合的构造方法

为了计算文法符号X的FIRST(X)，我们可以应用下列规则，直到没有终结符或ϵ可加到某个FIRST集合为止：

- **如果X是终结符，则FIRST(X)是{X}。**
- **如果X->ϵ是一个产生式，则将ϵ加到FIRST(X)中。**
- **如果X是非终结符，且$X->Y_1Y_2...Y_k$是一个产生式，**

  则：若对于某个i，a属于$FIRST(Y_i)$且ϵ属于$FIRST(Y_1)$，…,$FIRST(Y_{i-1})$，

  即$Y_1\cdots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$，

  则将a加入FIRST(X)中；

  若对于所有的j=1，2，…，k，ϵ在$FIRST(Y_j)$中，

  则将加到FIRST(X)中。例如：$FIRST(Y_1)$中的每个元素确实都在FIRST(X)中。

  如果Y_1不能推导出ϵ，则不再往FIRST(X)中增加新的符号，如果$Y_1 \overset{*}{\Rightarrow} \epsilon$，

  则将$FIRST(Y_2)$加到FIRST(X)中，以此类推。

FOLLOW集合的构造方法

- 为计算所有非终结符A的后继符号集合FOLLOW(A)，我们可以应用如下规则，直到每个FOLLOW集合都不能再加入任何符号或$为止：

  - 将$放入FOLLOW(S)中，其中S是开始符号，$是输入串的结束符。

  - 如果存在产生式A->αBβ，则将FIRST(β)中除ϵ以外的符号都放入FOLLOW(B)中。
  - 如果产生式A->αB，或A->αBβ，其中：FIRST(β)中包含ϵ（即β $\stackrel{*}{\Rightarrow}$ ϵ），则将FOLLOW(A)中的所有符号都放入FOLLOW(B)中。

例4.12

我们再来看文法(4-11)：

E -> TE'

E' -> +TE' | ϵ

T -> FT' (4-11)

T' -> *FT' | ϵ

F -> (E) | id

那么：

FIRST(E) = FIRST(T) = FIRST(F) = {(, id}

FIRST(E') = {+, ϵ}

FIRST(T') = {*, ϵ}

FOLLOW(E) = FOLLOW(E') = {), $}

FOLLOW(T) = FOLLOW(T') = {+,), $}

FOLLOW(F) = {+, *,), $}

4.4.6 预测分析表的构造

算法4.4 构造预测分析表。

- *输入*：文法G。
- *输出*：分析表M。
- *方法*：

1、对于文法中的每个产生式A->α，执行第2和第3步。

2、对FIRST(α)中的每个终结符a，将A->α加入到M[A,a]中。

3、若ϵ在FIRST(α)中，则对FOLLOW(A)的每个终结符b，将A->α加入到M[A,b]中；若ϵ在FIRST(α)中，且$在FOLLOW(A)中，则将A->α加入到M[A,$]中。

4、将M中每个没定义的表项均设置为error。

例4.13

算法4.4应用于文法(4-11)后产生的分析表如图4-15所示。

NONTER-MINAL	INPUT SYMBOL					
	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	$F \to id$			$F \to (E)$		

Fig. 4.15. Parsing table M for grammar (4.11).

- ∵ FIRST(TE')=FIRST(T)={(, id},
- ∴ 产生式E->TE'被填入M[E, (]和M[E, id] 表项中.
- ∵ FIRST(+TE')={+},
- ∴ 产生式E'->+TE'被填入M[E', +]表项中
- ∵ FIRST(ϵ)={ ϵ }, 而需要求FOLLOW(E')={), $}
- ∴ 产生式E'-> ϵ 被填入M[E',)]和M[E', $]表项含有E'->ϵ。

例4.14

消除下列文法中左递归， 并构造一个预测分析表

S -> (L) | a

L -> L , S | S

4.4.7 LL(1)文法

- 算法4.4应用于下面文法后产生的分析表如图4-17所示：

S -> iEtSS' | a

S' -> eS | ϵ

E -> b

NONTER-MINAL	INPUT SYMBOL					
	a	*b*	*e*	*i*	*t*	$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to \epsilon$ $S' \to eS$			$S' \to \epsilon$
E		$E \to b$				

Fig. 4.17. Parsing table *M* for grammar (4.13).

- ∵ FOLLOW(S')={e, $}，
- ∴ M[S', e]同时包含S'->eS和S'->ϵ。该文法具有二义性的。
- LL(1)文法的定义：分析表中没有多重定义表项的文法叫做LL(1)文法。算法4.4可以为任何LL(1)文法G产生分析表。这个分析表能分析G的所有句子，而且只能分析G的句子。

LL(1)文法的性质

- LL(1)文法有一种特殊的性质。它不是二义性的，也不含左递归。

  可以证明，G是LL(1)文法，当且仅当G的任何两个不同的产生式

  A->α|β满足下面的条件：

  - 不存在这样的终结符a，使得α和β导出的串都以a开始。
  - α和β中至多有一个能导出空串。
  - 如果β $\overset{*}{\Rightarrow}$ ϵ，那么α不能导出以FOLLOW(A)中的终结符开始的任何字符串。

- **注意：**如果一个文法经过消除左递归和提取所有可能的左因子后也不能产生LL(1)文法。一般来说，没有一个普遍适用的规则可以用来删除多重定义的表项，使其成为单值而不影响语法分析器所识别的语言。

作业

对下列每一个文法完成：

- 提取左公因子；
- 消除左递归；
- 构造预测分析表；
- 判断文法是否是LL(1)文法；

(1) S->SS+|SS*|a

(2) S->0S1|01

(3) S->aSbS|bSaS| ϵ

作业答案

(1) S->SS+|SS*|a

- 提取左公因子：S ->SSS'| a

  S' -> + | *

- 消除左递归： S->aS"

  S"->SS'S"| ϵ

  S' -> + | *

- 构造预测分析表：FIRST(aS")={a} , FIRST(SS'S")={a},

  FIRST(+)={+} , FIRST(*)={*}, FIRST(ϵ)={ϵ}

  FOLLOW(S")=FOLLOW(S)={+,*,$}

	+	*	a	$
S			S->aS"	
S"	S"-> ϵ	S"-> ϵ	S"->SS'S"	S'-> ϵ
S'	S'->+	S'->*		

- 表中没有多重定义的表项，所以是 LL（1）文法。

作业答案

（2）S->0S1|01

- 提取左公因子： S->0S’

  S’->S1|1

- 构造预测分析表： FIRST(0S)={0}

  FIRST(S1)={0}

  FIRST(1)={1}

	0	1	$
S	S->0S’		
S’	S’->S1	S->1	

- 表中没有多重定义的表项，所以是 LL（1）文法。

作业答案

（3）S->aSbS|bSaS| ϵ

- 构造预测分析表： FIRST(aSbS)={a} , FIRST(bSaS)={b}

  FIRST(ϵ)={ϵ}， FOLLOW(S)={a,b,$}

	a	b	$
S	S->aSbS S-> ϵ	S->bSaS S-> ϵ	 S-> ϵ

- 表中有多重定义的表项，文法有二义性，不是 LL（1）文法

4.5 自底向上语法分析

- 本节介绍一种比较常用的自底向上分析方法，称为移动归约分析法。
- 移动归约分析法为输入串构造分析树时从叶结点开始，像根节点前进。
- 我们可以把该过程看成是把输入串w"归约"成文法开始符号的过程。在每一步归约中，如果一个子串和某个产生式的右部匹配，则用该产生式的左部符号代替该子串。如果每一步都能恰当地选择子串了，我们就可以得到最右推导的逆过程。

例4.15

- 考虑文法

  S -> aABe

  A -> Abc | b

  B -> d

  句子abbcde可按下述步骤归约到S:

  abbcde

  aAbcde

  aAde

  aABe

  S

- 事实上，这些归约与如下最右推导的逆过程相对应

  $S \underset{rm}{\Rightarrow} aABe \underset{rm}{\Rightarrow} aAde \underset{rm}{\Rightarrow} aAbcde \underset{rm}{\Rightarrow} abbcde$

4.5.1 归约

- 非形式定义：一个符号串的“句柄”是和一个产生式右部匹配的子串，而且把它‘归约’到该产生式左部的非终结符代表了最右推导逆过程的一步。
- 形式定义：右句型γ的句柄是一个产生式A->β以及γ的一个位置，在该位置可以找到串β，而且用A代替β可以得到γ的最右推导的前一个右句型，

  即如果$S \overset{*}{\underset{rm}{\Rightarrow}} \alpha Aw \underset{rm}{\Rightarrow} \alpha\beta w$，那么紧跟在α后面的A->β是αβw的句柄。

  注意：如果文法是具有二义性的，则句柄不一定唯一，因为可能有不止一个αβw的最右推导。只有文法没有二义性时，它的每个右句型才有一个句柄。

在上面的例子中，abbcde是右句型，句柄是在位置2的A->b。

例4.16

- 考虑文法:

$$
\begin{aligned}
&(1)\quad E \to E + E\\
&(2)\quad E \to E * E\\
&(3)\quad E \to (E)\\
&(4)\quad E \to \mathbf{id}
\end{aligned}
\qquad (4\text{-}16)
$$

- 最右推导:

$$
\begin{aligned}
E &\underset{rm}{\Rightarrow} \underline{E + E}\\
&\underset{rm}{\Rightarrow} E + \underline{E * E}\\
&\underset{rm}{\Rightarrow} E + E * \underline{\mathbf{id}_3}\\
&\underset{rm}{\Rightarrow} E + \underline{\mathbf{id}_2} * \mathbf{id}_3\\
&\underset{rm}{\Rightarrow} \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3
\end{aligned}
$$

方便起见，我们给id加以下标，并给每个右句型的句柄加上下划线。例如，id_1是右句型$id_1+id_2*id_3$的句柄。注意句柄右边的串中仅含终结符。

- 因为文法(4-16)是具有二义性，存在$id_1+id_2*id_3$的另一个右推导：

$$
\begin{aligned}
E &\underset{rm}{\Rightarrow} \underline{E * E} \\
&\underset{rm}{\Rightarrow} E * \underline{\mathbf{id}_3} \\
&\underset{rm}{\Rightarrow} \underline{E + E} * \mathbf{id}_3 \\
&\underset{rm}{\Rightarrow} E + \underline{\mathbf{id}_2} * \mathbf{id}_3 \\
&\underset{rm}{\Rightarrow} \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3
\end{aligned}
$$

- 考虑右句型$E+E*id_3$，在该推导中$E+E$是$E+E*id_3$的句柄，而在上一个推导中id_3是该右句型的句柄

4.5.2 句柄裁剪

- 通过“裁剪句柄”可以得到最右推导的逆过程。

- 考虑例4.16的文法(4-16)的文法和输入串$id_1+id_2*id_3$。它的规约序列如图4-21所示。容易看出，该右句型序列正好是例4.16中第一个最右推导序列的逆序。

RIGHT-SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

Fig. 4.21. Reductions made by shift-reduce parser.

4.5.3 移动-归约语法分析技术

- 实现移动归约分析的一种简单方法是用栈来保存文法符号，用输入缓存区来保存要分析的串w，用$来标记栈底，也用它标记输入串的右端。初始，栈是空的，串w在输入缓冲区中，如下所示：

STACK	INPUT
$	w $

- 语法分析器将零个或多个输入符号压入栈，直到句柄β在栈顶出现为止，语法分析器再把β归约成某个恰当的产生式右部。语法分析器重复此过程，直到它发现错误或者栈中只含有开始符号并且输入串为空：

STACK	INPUT
$S	$

- 进入这个格局后，语法分析器停止并宣告分析成功。

例4.17

让我们逐步看一下移动归约语法分析器在分析输入串 $id_1+id_2*id_3$ 时的动作，文法是(4-16)，使用例4.16的第一种推导。动作序列如图4-22所示。注意，由于文法(4-16)对该输入有两种最右推导，所以语法分析器还可能采取另一个动作序列。

	STACK	INPUT	ACTION
(1)	$	$id_1 + id_2 * id_3$	shift
(2)	$id_1	$+ id_2 * id_3$	reduce by $E \rightarrow id$
(3)	$E	$+ id_2 * id_3$	shift
(4)	$E +	$id_2 * id_3$	shift
(5)	$E + id_2	$* id_3$	reduce by $E \rightarrow id$
(6)	$E + E	$* id_3$	shift
(7)	$E + E *	id_3	shift
(8)	$E + E * id_3	$	reduce by $E \rightarrow id$
(9)	$E + E * E	$	reduce by $E \rightarrow E * E$
(10)	$E + E	$	reduce by $E \rightarrow E + E$
(11)	$E	$	accept

Fig. 4.22. Configurations of shift-reduce parser on input $id_1 + id_2 * id_3$.

移动归约语法分析器的基本动作

- 移动归约语法分析器的基本动作是移动和归约的，但实际上有四种可能的动作：移动、归约、接受、出错。
  - 移动：把下一个输入符号移动到栈顶。
  - 归约：语法分析器知道句柄的右端已在栈顶。它必须在栈中确定句柄的左端，并选择正确的非终结符替代句柄。
  - 接受：语法分析器宣告分析成功。
  - 出错：语法分析器发现了一个语法错误，并调用错误恢复处理程序进行错误处理。

4.5.4 可行前缀（活前缀）

- 出现在移动归约语法分析器栈中的右句型的前缀集合称为可行前缀。
- 等价的定义为：可行前缀是右句型的前缀，而且其右端不会超过该句型的最右边句柄的末端。

即:若 $A \to \alpha\beta$ 是文法的一个产生式，S是文法开始符，

并有： $S \overset{*}{\underset{R}{\Rightarrow}} \delta A\omega \underset{R}{\Rightarrow} \delta\alpha\beta\omega$

则 $\delta\alpha\beta$ 的任何前缀都是规范句型 $\delta\alpha\beta\omega$ 的可行前缀。

在上述定义中，$\alpha\beta$ 是句型 $\delta\alpha\beta\omega$ 关于A的直接短语，并且是一最右推导(规范推导)，所以 $\alpha\beta$ 是最左直接短语，是一句柄，

因此 $\delta\alpha\beta$ 的任何前缀不含句柄后的任何符号，特别，句柄是 $\alpha\beta$ 的后缀,$\delta\alpha\beta$ 是分析栈栈顶的符号串。

4.5.5 移动归约分析过程的冲突

有些上下文无关文法不能使用移动归约分析方法进行分析。这种文法的每一个移动归约语法分析器会形成这样的格局：根据栈中的内容和下个输入符号不能决定是移动还是归约（移动-归约冲突），或不能决定按哪一个产生式进行归约（归约-归约冲突）。

二义性文法一定不能使用移动归约分析方法进行分析。

4.6 LR语法分析技术介绍：简单LR技术

- 本节介绍一种有效的自底向上的语法分析技术。

我们也将给出三种构造LR分析表的方法：

- 第一种方法称为简单LR方法（简称SLR），它最容易实现，但功能最弱，对某些文法可能失败。
- 第二种方法称为规范的LR方法，它功能最强，代价也最高。
- 第三种方法叫做向前看的LR方法（简称LALR），其功能和代价介于前两者之间。

4.6.1 为什么使用 LR语法分析技术

- LR分析富有吸引力的原因如下：
  - LR语法分析器能识别几乎所有能用上下文无关文法描述的程序设计语言的结构。
  - LR分析法是已知的最一般的无回溯移动归约语法分析法，而且可以和其他移动归约分析法一样被有效地实现。
  - LR分析法分析的文法类是预测分析法能分析的文法类的真超集。
  - 在自左向右扫描输入符号串时，LR语法分析器能及时发现语法错误。
- 主要缺点是：对典型的程序设计语言文法，手工构造LR语法

分析器的工作量太大，需要专门的工具。如Yacc。

4.6.2 LR语法分析算法

- LR语法分析器的模型如图4-29所示。
- 驱动程序对所有的LR语法分析器都是一样的，不同的语法分析器只是语法分析表不同。

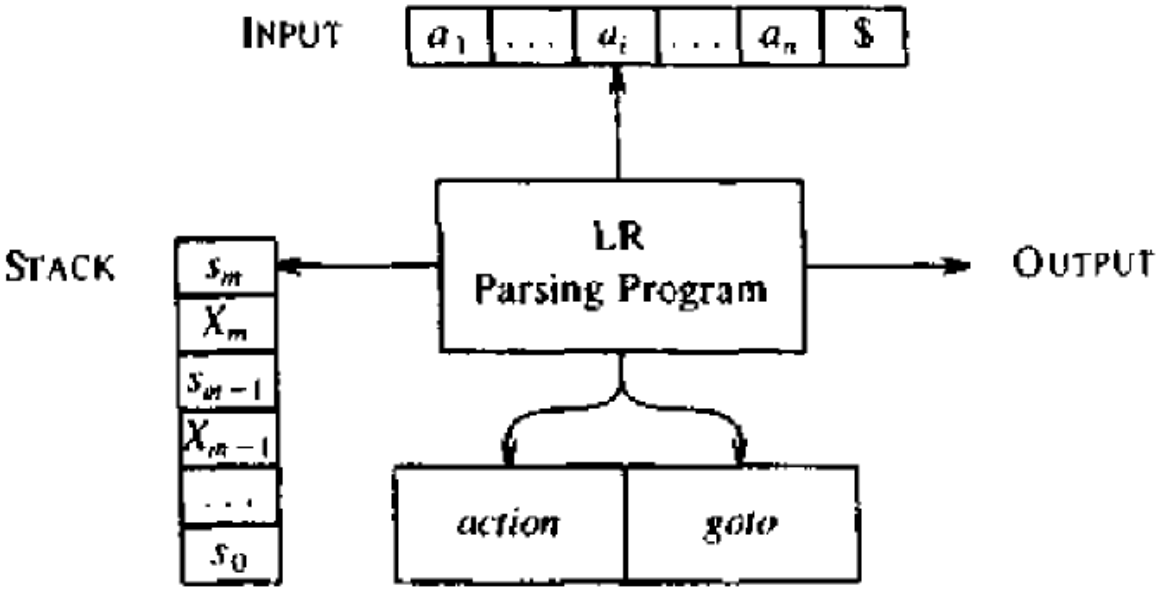


Fig. 4.29. Model of an LR parser.

- 语法分析表由动作函数action和转移函数goto两部分组成。驱动LR语法分析器的程序工作如下：它根据栈顶状态s_m和当前输入符号a_i访问action[s_m, a_i]，即s_m和a_i所对应的语法分析表项的动作部分，可能的动作如下：
  - 移动状态s进栈。
  - 按文法产生式A->β归约。
  - 接受
  - 出错
- 函数goto以状态和文法符号为参数，产生一个状态。我们将看到，用SLR、规范的LR和LALR方法从文法G构造的语法分析表的goto函数是识别G的活前缀的确定的有穷自动机的转换函数。

- 算法4.7 LR分析算法

输入：文法G的LR语法分析表和输入串w。

输出：如果w属于L(G)，则输出w的自底向上分析，否则报错。

方法：首先，把初始状态s_0放在语法分析器的栈顶，把w$放在输入缓存区；然后，语法分析器执行图4-30的程序，直到遇见接受或出错动作为止。

```
set ip to point to the first symbol of w$;
repeat forever begin
    let s be the state on top of the stack and
        a the symbol pointed to by ip;
    if action[s, a] = shift s' then begin
        push a then s' on top of the stack;
        advance ip to the next input symbol
    end
    else if action[s, a] = reduce A → β then begin
        pop 2*|β| symbols off the stack;
        let s' be the state now on top of the stack;
        push A then goto[s', A] on top of the stack;
        output the production A → β
    end
    else if action[s, a] = accept then
        return
    else error()
end
```

Fig. 4.30. LR parsing program.

例4.18

- 下面是含有二元操作符

  +和*的算术表达式文法:

  (1) $E \rightarrow E + T$
  (2) $E \rightarrow T$
  (3) $T \rightarrow T * F$
  (4) $T \rightarrow F$
  (5) $F \rightarrow (E)$
  (6) $F \rightarrow \mathbf{id}$

- 图4-31给出了文法G的LR语法分析表，包括动作函数和转移函数

STATE	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			

Fig. 4.31. Parsing table for expression grammar.

- 各个动作的编码的含义是：
  - 1、s_j表示把当前输入符号和状态i压进栈。
  - 2、r_j表示按第j个产生式归约。
  - 3、acc表示接受。
  - 4、空白表示出错。
- 注意，对于终结符a，状态转移动作goto[s, a]可以在表的动作表项action[s, a]中找到，所以转移域中仅给出非终结符A的goto[s, A]。另外，我们还没有解释图4-31的表项都是怎么确定的，这个问题稍后讨论。

- 输入字符串： id*id+id,
- 栈和输入内容的变化序列如图4-32所示。
- 例如第一行，状态0，id是第一个输入符号，对应的动作域是s5，则id进栈，状态5压入栈顶。
- 然后，*为当前输入符号，状态5，按F->id归约：弹出栈顶两个符号，栈顶状态0。因为goto[0, F]是3，则把F和3压入栈。
- 其余类推。

	STACK	INPUT	ACTION
(1)	0	id * id + id $	shift
(2)	0 id 5	* id + id $	reduce by $F \rightarrow$ id
(3)	0 F 3	* id + id $	reduce by $T \rightarrow F$
(4)	0 T 2	* id + id $	shift
(5)	0 T 2 * 7	id + id $	shift
(6)	0 T 2 * 7 id 5	+ id $	reduce by $F \rightarrow$ id
(7)	0 T 2 * 7 F 10	+ id $	reduce by $T \rightarrow T*F$
(8)	0 T 2	+ id $	reduce by $E \rightarrow T$
(9)	0 E 1	+ id $	shift
(10)	0 E 1 + 6	id $	shift
(11)	0 E 1 + 6 id 5	$	reduce by $F \rightarrow$ id
(12)	0 E 1 + 6 F 3	$	reduce by $T \rightarrow F$
(13)	0 E 1 + 6 T 9	$	$E \rightarrow E+T$
(14)	0 E 1	$	accept

Fig. 4.32. Moves of LR parser on id * id + id.

4.6.3 LR文法

- 给定文法G，如果我们能为G构造出LR语法分析表，则称G是LR文法。很多上下文无关文法不是LR文法。但是典型的程序设计语言的结构一般都可以避免非LR文法。
- 直观地，为了使一个文法是LR文法，只要保证在句柄出现在栈顶时，自左向右扫描的移动归约分析器能够及时地识别它。
- 能够用来帮助LR语法分析器做出移动归约决定的另一个信息源是下k个输入符号。例如图4-31是LR(1)。每步需要向前看k个符号的LR语法分析器所分析的文法叫做LR(k)文法。

LL文法和LR文法之间区别

- 对于LR(k)文法，我们必须通过向前看k个输入符号就能够知道一个产生式的右部所能推导出的所有字符串，进而识别出这个产生式右部的出现。这个要求远远不如LL(k)那么严格。

- LL(k)文法，在决定是否使用某个产生式时，要求只能向前看该产生式右部推出的前k个符号。所以LR文法比LL文法描述的语言更多。

4.6.4 构造SLR语法分析表

- 文法G的LR(0)项目是在G的产生式右部的某处加点的产生式。例如，产生式A->XYZ可以生成如下四个项目：

  A -> ·XYZ

  A -> X·YZ

  A -> XY·Z

  A -> XYZ·

- SLR方法的主要思想是首先从文法构造出识别活前缀的确定有穷自动机。
- 我们把项目划分成一组集合，这些集合对应SLR语法分析器的状态。项目可以看成识别活前缀的NFA的状态，而“项目分组”就是3.6节中讨论的子集构造法。

- 被称为规范LR(0)项目集族是构造SLR语法分析表的基础。为了构造文法的规范LR(0)项目集族，我们需要定义拓广文法的概念，并引入闭包(closure)运算和转移函数(goto)。
- 如果文法G的开始符号是S，那么G的拓广文法G'是在G的基础上增加一个新的开始符号S'和产生式S'->S。新产生式的目的是用来指示语法分析器什么时候应该停止分析并宣布接受输入，即当且仅当语法分析器执行归约S'->S时，分析成功。

4.6.4.1 闭包运算closure

- 如果I是文法G的项目集，那么closure(I)是从I出发由下面两条规则构造的项目集：
  - 1、初始时，把I的每个项目都加入到closure(I)中。
  - 2、如果A->α·Bβ在closure(I)中，且存在产生式B->γ，若B->·γ不在closure(I)中，则将其加入到closure(I)中。反复运用这条规则，直到没有更多的项目可以加到closure(I)为止。

例4.19

- 考虑拓广的表达式文法：

$$
\begin{aligned}
E' &\rightarrow E \\
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
\tag{4-19}
$$

- 如果I是一个项目集合{[E' -> ·E]}，那么closure(I)包含下列项目：

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

- 例计算函数closure的算法如图4-33所示。

```
function closure ( I );
begin
        J := I;
        repeat
              for each item A → α·Bβ in J and each production
                    B → γ of G such that B → ·γ is not in J do
                          add B → ·γ to J
        until no more items can be added to J;
        return J
end
```

Fig. 4.33. Compotation of *closure*.

- 核心项目： 初始项S'->S和所有点不在左端的项目
- 非核心项目： 点在左端的非初始项目。

4.6.4.2 goto函数

- 第二个非常有用的函数是goto(I, X)，其中I是项目集，X是文法符号。goto(I, X)定义为所有项目集[A->αX·β]（[A->α·Xβ]在I中）的闭包。

- 例4.35 若I是两个项目的集合{[E'->E], [E->E·+T]}，则goto(I, +)包含下列项目：

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

4.6.4.3 项目集的构造

项目集构造算法如图4-34所示：

```
procedure items(G');
begin
    C := {closure({[S' → ·S]})};
    repeat
        for each set of items I in C and each grammar symbol X
            such that goto(I, X) is not empty and not in C do
                add goto(I, X) to C
    until no more sets of items can be added to C
end
```

Fig. 4.34. The sets-of-items construction.

例4.20

- 考虑拓广的表达式文法：

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

- 规范LR(0)项目集族如图4-35所示。

I_0: $E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

I_1: $E' \to E \cdot$
$E \to E \cdot + T$

I_2: $E \to T \cdot$
$T \to T \cdot * F$

I_3: $T \to F \cdot$

I_4: $F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

I_5: $F \to \mathbf{id} \cdot$

I_6: $E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

I_7: $T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

I_8: $F \to (E \cdot)$
$E \to E \cdot + T$

I_9: $E \to E + T \cdot$
$T \to T \cdot * F$

I_{10}: $T \to T * F \cdot$

I_{11}: $F \to (E) \cdot$

Fig. 4.35. Canonical LR(0) collection for grammar (4.19).

- 考虑拓广的表达式文法：

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

- 这个项目集的goto函数被示为图4-36中有穷自动机D的转换状态图。

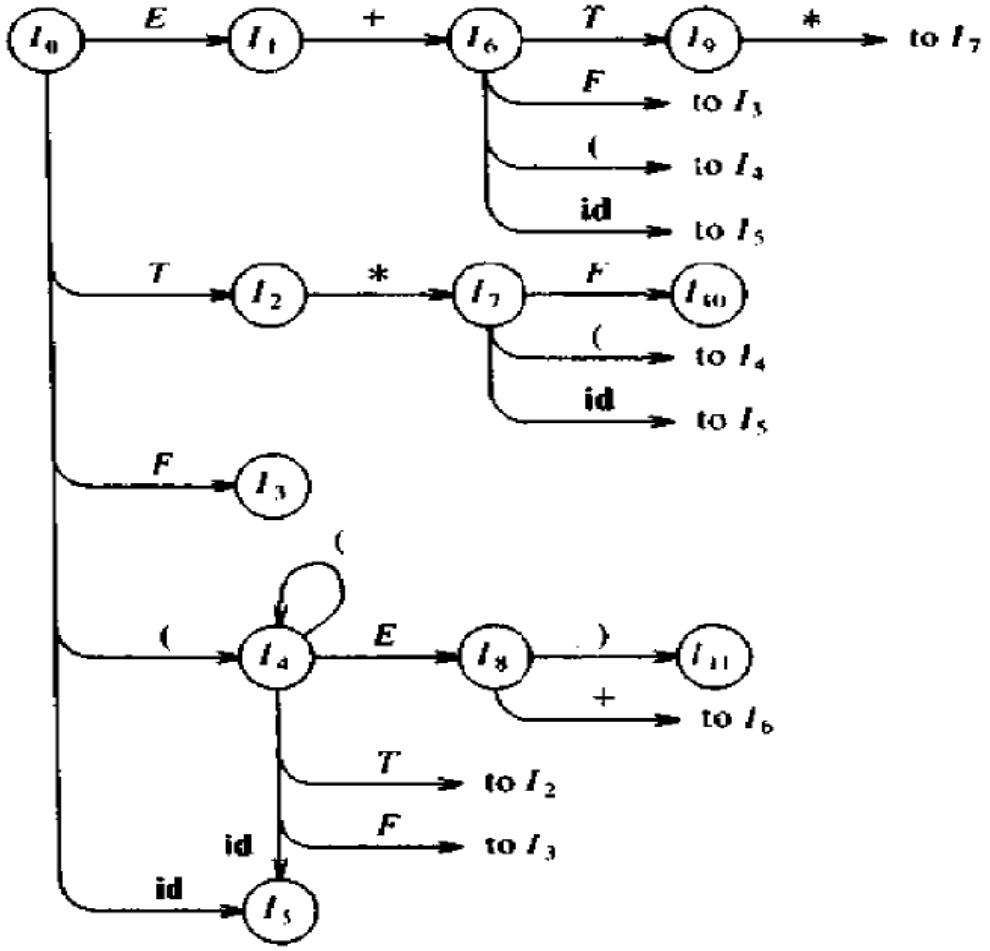


Fig. 4.36. Transition diagram of DFA *D* for viable prefixes.

- 有效项目：如果存在一个推导$S' \overset{*}{\underset{rm}{\Rightarrow}} \alpha Aw \overset{*}{\underset{rm}{\Rightarrow}} \alpha\beta_1\beta_2 w$，

  则：项目$A->\beta_1\cdot\beta_2$ 对可行前缀$\alpha\beta_1$是有效的。
- 一般而言，同一个项目可能对多个可行前缀有效。
- $A->\beta_1\cdot\beta_2$对前缀$\alpha\beta_1$有效这个事实告诉我们，在发现$\alpha\beta_1$在分析栈时是 移进还是归约，特别是：

  如果：$\beta_2 \neq \epsilon$，它暗示句柄还没有完全进栈，

  动作应该是移进。

  如果：$\beta_2=\epsilon$， 那么$A->\beta_1$是句柄，

  应该用这个产生式归约。

- **注意（1）：** 当然，同一个可行前缀的两个有效项目可能告诉我们做不同的事情，有些这样的冲突可以通过向前看下一个输入符号来解决，其他一些冲突可以用下一节的方法解决。当LR方法用于构造任意文法的语法分析表时，不能保证所有冲突都能解决。

- **注意（2）：** 如果A是一个由规范项目集族构造、以goto函数为转换函数的DFA，则一个活前缀γ的有效项目集正好从A的初态出发，沿着标记为γ的路径所能到达的那些项目的集合。这是LR分析理论的一条基本定理。

4.6.4.4 SLR语法分析表的构造算法

- *输入*：拓广文法G'
- *输出*：G'的SLR语法分析表函数action和goto
- *方法*：1、构造C={I_0，I_1,…，I_n}，即G'的规范LR(0)项目集族。

2、从I_i构造状态i，它的分析动作确定如下：

(a) 如果[A->α·aβ]在I_i中，并且goto(I_i,a)=I_j，

则置action[i,a]为“移动j进栈”，这里a必须是终结符。

(b) 如果[A->α·]在I_i中，

则对FOLLOW(A)中的所有a，

置action[i,a]为“归约A->α”，这里A不能是S'。

(c) 如果[S'->S·]在I_i中，则置action[I,$]为“接受”。

如果:由上面的规则产生的动作有冲突,则G不是SLR(1)文法，

就构造不出语法分析器。

3、对所有的非终结符A，使用下面的规则构造状态i的goto函数：

如果goto(I_i,A)=I_j，则goto[i,A]=j。

4、不能由规则(2)和(3)定义的表项都置为“出错”。

5、语法分析器的初始状态是从包含[S'->·S]的项目集构造出的状态。

例4.21

- 让我们为文法(4-19)构造SLR表。它的规范LR(0)项目集族已在图4-35中给出。考虑I_0:

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

- 项目F->· (E)使得action[0,(]="移动4进栈"，项目F->·id使得action[0,id]="移动5进栈"。I_0其他项目不产生动作。其他项目集一次类推，可得到图4-31的分析表。

STATE	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			

Fig. 4.31. Parsing table for expression grammar.

注意：

- 每个SLR(1)文法都不是二义的，
- 但有很多非二义的文法不是SLR(1)文法。
- 考虑下面的文法：

S -> L=R
S -> R
L -> *R (4-20)
L -> id
R -> L

文法(4-20)的规范LR(0)项目集族如图4-37所示。

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

I_1: $S' \to S \cdot$

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

I_3: $S \to R \cdot$

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

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

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

I_7: $L \to *R \cdot$

I_8: $R \to L \cdot$

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

Fig. 4.37. Canonical LR(0) collection for grammar (4.20).

- 考虑项目集I_2。该集合的第一项使得action[2,=]为“移动6进栈”，因为FOLLOW(R)包含=。而第二项使得action[2,=]为“归约R-> L”。于是，action[2,=]有多重定义，存在移动归约冲突。但文法(4-20)不是二义性的。
- 部分这类有冲突的无二义性文法可以用后面要讨论的规范LR文法和LALR文法来消除。

4.7 更强大的LR语法分析器

构造LR语法分析表的最一般技术：在SLR表中

- 如果项目集I_i包含项目[A->α·]并且a在FOLLOW(A)中，则状态i调用A->α归约。

- 然而，在某些情况下，当状态i出现在栈顶时，栈内的可行前缀βα使得在任何右句型中，βA不能跟随a。因此在这种情况下，用A->α进行归约无效。

例4.22

- 让我们重新考虑下面的文法：

  S -> L=R
  S -> R
  L -> *R　　(4-20)
  L -> id
  R -> L

文法(4-20)的规范LR(0)项目集族如图4-37所示。

- 状态2中项目R->L·对应于上面的A->α，a对应于符号=，=在FOLLOW(R)中。于是，当SLR语法分析器处于状态2而且下一个输入符号为=时，SLR语法分析器调用R->L归约。然而，例4.39的文法没有以R=…开始的右句型。因此，仅于活前缀L相对应的状态2不该进行归约。

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

I_1: $S' \to S \cdot$

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

I_3: $S \to R \cdot$

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

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

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

I_7: $L \to *R \cdot$

I_8: $R \to L \cdot$

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

Fig. 4.37. Canonical LR(0) collection for grammar (4.20).

4.7.1 规范LR(1)项目

- 通过重新定义项目，使之包含一个终结符作为第二个分量，可以把更多的信息并入状态中。项目的一般形式也就变成了[A->α·β,a]，其中A->αβ是产生式，a是终结符或$。这样的对象叫做LR(1)项目。1是第二个分量的长度，这个分量叫做项目的搜索符。
- 形式地，我们说LR(1)项目[A->α·β,a]对活前缀γ有效，如果存在推导$S \overset{*}{\underset{rm}{\Rightarrow}} \delta Aw \overset{}{\underset{rm}{\Rightarrow}} \delta\alpha\beta w$，其中：
  - 1、γ=δα
  - 2、a是w的第一个符号，或者w是ϵ且a是$。

例4.23

- 让我们考虑如下的文法：

S -> BB

B -> aB | b

它有一个最右推导$S \overset{*}{\underset{rm}{\Rightarrow}} aaBab \underset{rm}{\Rightarrow} aaaBab$。在上面的定义中，令δ=aa，A=B，w=ab，α=a，β=B，我们可以看到，项目[B->a·B,a]对活前缀γ=aaa是有效的。这个文法的另一个最右推导是$S \overset{*}{\underset{rm}{\Rightarrow}} BaB \underset{rm}{\Rightarrow} BaaB$。从这个推导可以看出，项目[B->a·B,$]对于活前缀Baa是有效的。

4.7.2 LR(1)项目集的构造算法

- *输入*：拓广文法G'。
- *输出*：LR(1)项目集，它们是对G'的一个或多个活前缀有效的项目集。
- *方法*：构造项目集的过程closure和goto及主例程items如图4-38所示。

```
function closure(I);
begin
    repeat
        for each item [A → α·Bβ, a] in I,
            each production B → γ in G',
            and each terminal b in FIRST(βa)
            such that [B → ·γ, b] is not in I do
                add [B → ·γ, b] to I;
    until no more items can be added to I;
    return I
end;

function goto(I, X);
begin
    let J be the set of items [A → αX·β, a] such that
        [A → α·Xβ, a] is in I;
    return closure(J)
end;

procedure items(G');
begin
    C := {closure({[S' → ·S, $]})};
    repeat
        for each set of items I in C and each grammar symbol X
            such that goto(I, X) is not empty and not in C do
                add goto(I, X) to C
    until no more sets of items can be added to C
end
```

Fig. 4.38. Sets of LR(1) items construction for grammar G'.

例4.24

- 考虑下面的拓广文法：

  S' -> S

  S -> CC (4-21)

  C -> cC | d

- 首先计算{[S'->S·,$]}的闭包，我们用项目[S'->S·,$]来匹配过程closure中的项目[A->αB·β,a]，即令A=S'，α=ϵ，B=S，β=ϵ，a=$。在函数closure中，要求对每个产生式B->γ和FIRST(βa)的每个终结符b，把[B->·γ,b]加到闭包中。根据当前文法，B->·γ只能是S->CC，而且因为β=ϵ，a=$，b也只能是$，因此将[S->·CC,$]加入到闭包中。

- 接下来，对于FIRST(C$)中的b，将项目[C->·γ,b]加入闭包。

也就是说，用项目[S->·CC,$]匹配项目[A->αB·β,a]，

其中A=S，α=ϵ，B=C，β=C，a=$。

因为C不会推导出空串，所以FIRST(C$)=FIRST(C)。

又因为FIRST(C)包含终结符c和d，所以将以下项目加入闭包：

[C->·cC,c]， [C->·cC,d]， [C->·d,c]， [C->·d,d]。

因为再没有紧跟在点右面的非终结符，所以我们完成了对一个LR(1)项目集的计算。这个初始项目集为：

I_0:
$S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot CC, \$$
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d, c/d$

- 为方便起见，我们把LR(1)项目的方括号省略了，而且用[C->·cC,c|d]表示[C->·cC,c]和[C->·cC,d]的缩写。
- 现在对不同的X值计算goto(I_0,X)。对X=S，因为点在右端，项目集中只有项目[S'->S,$]，因此第二个项目集为：$I_1$: S' -> S·，$

以此类推，得到图4-39所示的。

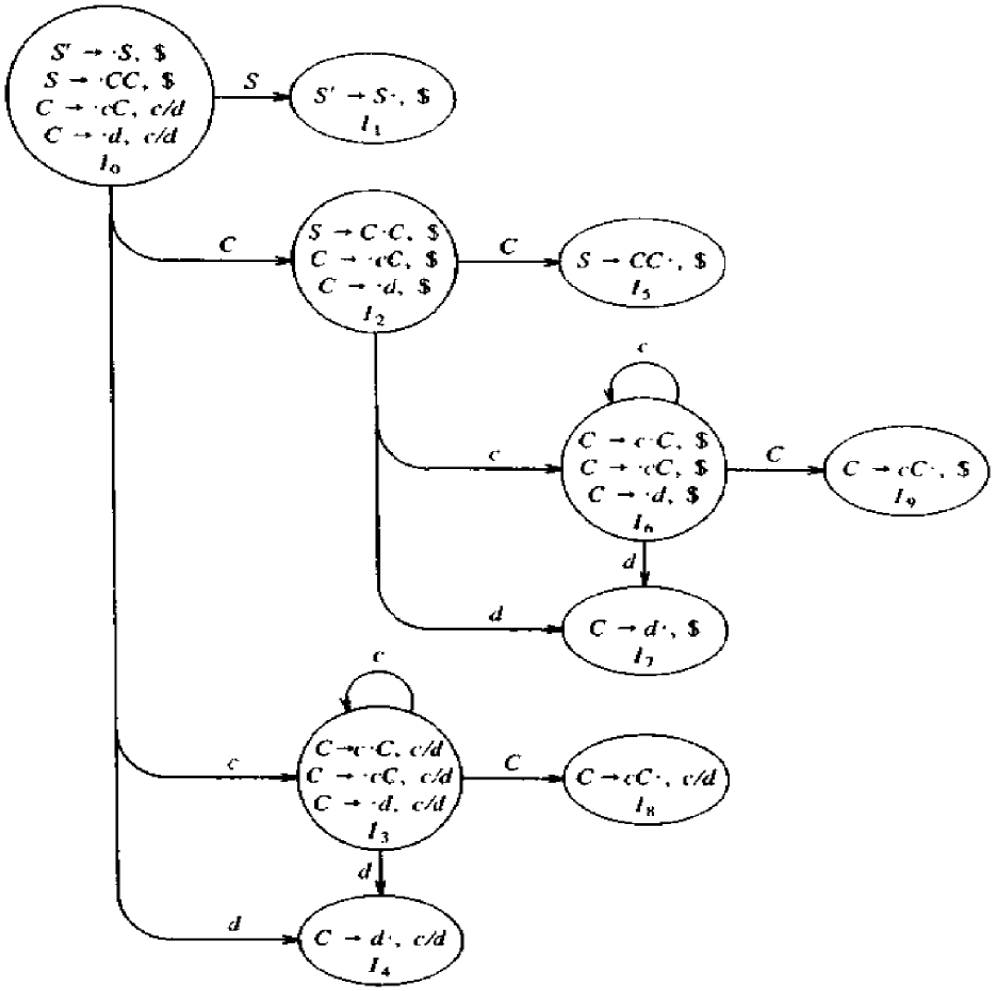


Fig. 4.39. The *goto* graph for grammar (4.21).

4.7.3 规范LR(1)语法分析表的构造算法

- *输入*：拓广文法G'。
- *输出*：文法G'的规范LR语法分析表函数action和goto。
- *方法*：

1、构造G'的LR(1)项目集规范族C={I_0，I_1,…，I_n}。

2、从I_i构造语法分析器的状态i，状态i的分析动作确定如下：

(a) 如果[A->α·aβ,b]在I_i中，并且goto(I_i,a)=I_j，
则置action[i,a]为s_j，即“移动j进栈”，这里a必须是终结符。

(b) 如果[A->α·,a]在I_i中且A≠S'，
则置action[i,a]为r_j，即按r_j归约，其中j是产生式A->α的序号。

(c) 如果[S'->S·,$]在$I_i$中，则置action[$I_i$,$]=acc，表示“接受”。

如果由上面的规则产生的动作有冲突，
则G不是LR(1)文法，就构造不出语法分析器。

3、对所有的非终结符A，使用下面的规则构造状态i的goto函数：
如果goto(I_i,A)=I_j，则goto[i,A]=j。

4、不能由规则(2)和(3)定义的表项都置为“出错”。

5、语法分析器的初始状态是从包含[S'->·S,$]的项目集构造出的状态。

例4.25

文法: S' -> S
S -> CC
C -> cC | d

的规范LR语法分析表如图4-40所示，产生式1、2和3分别是S->CC、C->cC和C->d。

- 注意：每个SLR(1)文法都是LR(1)文法，但对于SLR(1)文法，规范LR(1)文法可能具有更多的状态。上面的例子中的文法是SLR文法，它的SLR语法分析器只有7个状态，而图4-40却有10个状态。

STATE	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		

Fig. 4.40. Canonical parsing table for grammar (4.21).

4.7.4 构造LALR语法分析表

- 就语法分析器大小而言，SLR表和LALR表对同一个文法具有同样多的状态。构造SLR表和LALR表比构造规范LR表要经济得多。
- 我们可以寻找同心的（即第一分量相同）LR(1)项目集，并把这些同心的项目集合并成一个项目集。例如，图4-39中，I_4和I_7、I_3和I_6以及I_8和I_9。注意，一般而言，心是相应文法的一个LR(0)项目集；另外，LR(1)文法可能产生多个同心的项目集合。
- 同心集的合并不会引起新的移动-归约冲突；但可能产生新的归约-归约冲突。

例4.26

考虑如下文法：

S' -> S

S -> aAd | bBd | aBe | bAe

A -> c

B -> c

它只产生四个串：acd，ace，bcd和bce。通过构造该文法的LR(1)项目集，可以看出没有冲突，它是LR(1) 文法。

在它的项目集中，对活前缀ac有效的项目集为：

{[A->c·,d]，[B->c·,e]},

对bc有效的项目集为：

{[A->c·,e]，[B->c·,d]},

这两个集合都没有产生冲突且是同心的，然而他们合并后

A -> c·，d|e

B -> c·，d|e

产生归约-归约冲突。

一个简易但耗空间的LALR表构造算法

- *输入*：拓广文法G'。
- *输出*：G'的LALR语法分析表的action函数和goto函数。
- *方法*：

1、构造文法的LR(1)项目集规范族$C=\{I_0,I_1,\cdots,I_n\}$。

2、对出现在LR(1)项目集中的每个心，找出所有与之同心的项目集，用它们的并集代替它们。

3、令$C'=\{J_1,J_2,\cdots,J_n\}$是合并后的LR(1)项目集族。按照规范LR语法分析表的构造算法的方式从J_i来构造状态i的动作。如果分析动作出现冲突，算法无法产生分析表，说明该文法不是LR(1)文法。

4、goto表的构造如下：

如果J是一个或多个LR(1)项目集的并，即$J=I_1 \cup I_2 \cup \cdots \cup I_k$，那么$goto(I_1,X)$，$goto(I_2,X)$，…，$goto(I_k,X)$也同心。

记K为所有与$goto(I_1,X)$同心的项目集的并，则$goto(J,X)=K$。

例4.27

再次考虑文法文法: S' -> S 它的转移图如图 4-39所示。

S -> CC

C -> cC | d,

正如我们之前提到的，有3对项目集可以合并I_4和I_7、 I_3和I_6、 I_8和I_9、结果如图4-41所示。

STATE	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		

Fig. 4.41. LALR parsing table for grammar (4.21).

STATE	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		

Fig. 4.40. Canonical parsing table for grammar (4.21).

- 当输入串存在错误时，LALR语法分析器可能比LR语法分析器多做一些不必要的归约，而LR语法分析器则能立即报错。但LALR不会比LR多移进一个符号。
- 例如，若输入串是ccd（针对文法(4-21)）并跟随以$时，图4-34的LR语法分析器将把 0 c 3 c 3 d 4 压入栈，并在状态4发现错误，因为状态4面临$的动作是“出错”。

- 对于同一输入串，LALR语法分析器将产生相应的动作，即把

  0 c 36 c 36 d 47

压入栈。但状态47面临$的动作是归约C->d，栈的内容成为

  0 c 36 c 36 C 89

现在状态89面临$的动作是归约C->cC，这时栈的内容改为

  0 c 36 C 89

在经一次类似的归约，获得的栈为

  0 C 2

最后，状态2面临$的动作是“出错”。

作业

- （1）构造下面拓广文法的LR(0)和SLR（1）分析表：

  S’ -> S

  S -> CC

  C -> cC | d

- （2）构造下面拓广文法的规范LR（1）分析表：

  S->AS|b

  A->SA|a

作业答案

（1）构造下面拓广文法的LR(0)和SLR（1）分析表：

S' -> S

S -> CC

C -> cC | d

- 对各条产生式编号：

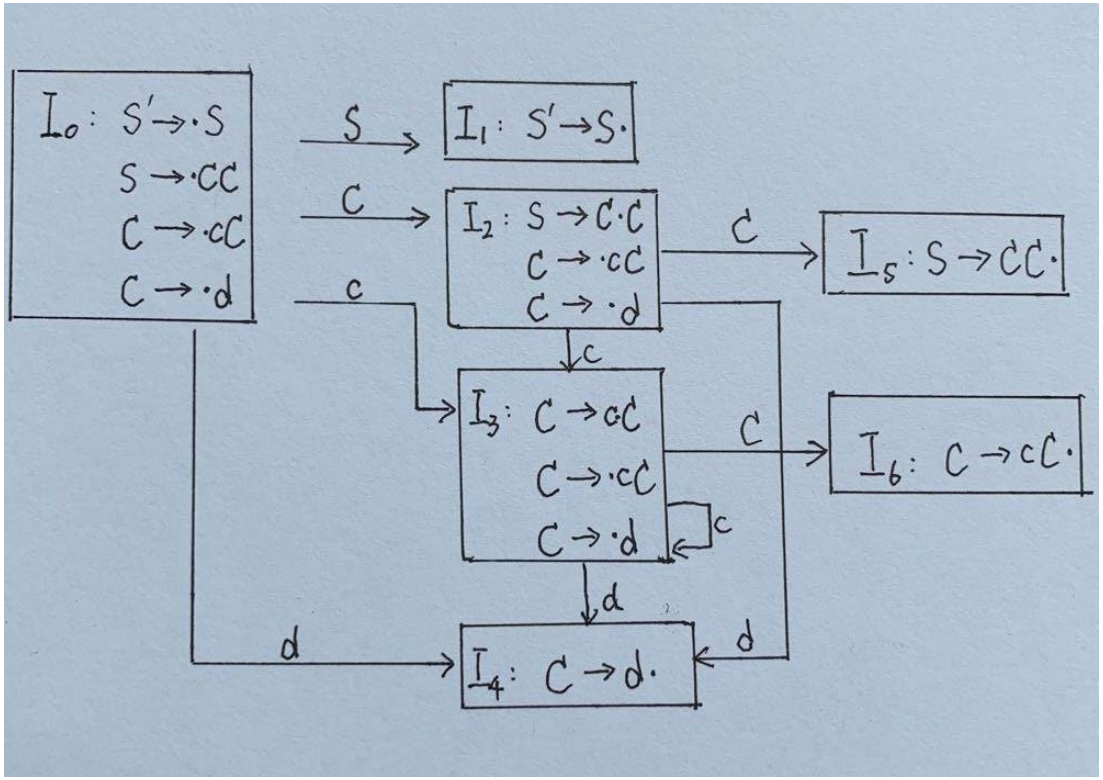
0: S' -> S

1: S -> CC

2: C -> cC

3: C -> d

- 然后，构造给定文法的 LR(0) 自动机，结果如图所示：



- 则 LR(0) 分析表结果如图所示：

状态	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

- 在此基础上，构造 SLR 分析表，则还需要计算文法的 Follow 集：
- Follow(S')={$}, Follow(S)={$} , Follow(C)={$,c,d}
- 则 SLR（1）分析表为：

状态	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5			r1		
6	r2	r2	r2		

（2）构造下面拓广文法的

规范LR（1）分析表： S->AS|b

A->SA|a

对各条产生式编号：

0: S' -> S

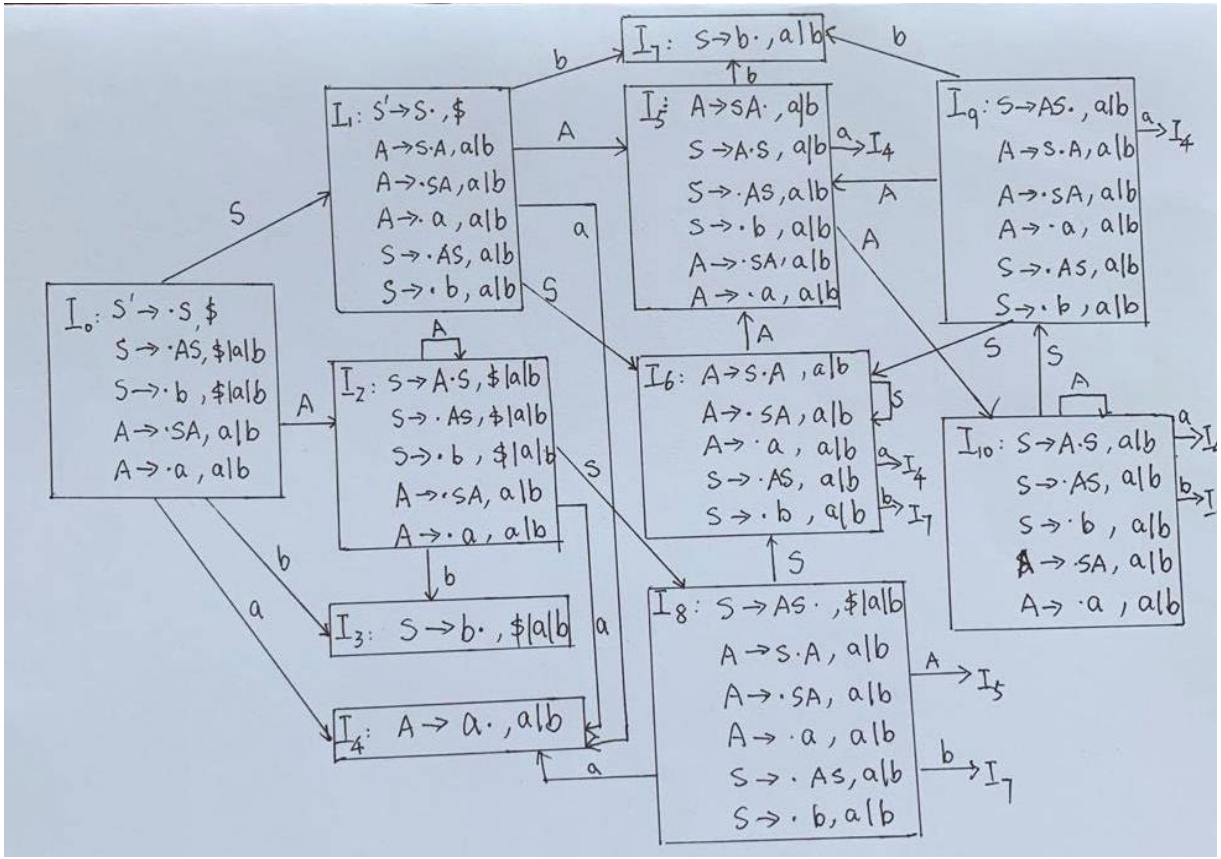
1: S -> AS

2: S -> b

3: A -> SA

4: A -> a

- 先计算文法的 First 集合，
- 结果为： First (A)={a,b), First(S)={a,b}
- 然后，构造给定文法的 LR(1) 自动机，计算过程用到了 First 集合，最终结果如图所示：



- 则 LR(1) 分析表结果如图所示:
- 从表中可以看到，存在冲突项，所以该文法不是 LR（1）的。

状态	ACTION			GOTO	
	a	b	$	S	A
0	s4	s3		1	2
1	s4	s7	acc	6	5
2	s4	s3		8	2
3	r2	r2	r2		
4	r4	r4			
5	S4/r3	s7/r3		9	10
6	s4	s7		6	5
7	r2	r2			
8	S4/r1	s7/r1	r1	6	5
9	S4/r1	s7/r1		6	5
10	s4	s7		9	10

附加：算符优先分析法

一、算符优先关系表

- 算符文法：上下文无关文法G，如果它不含ε－产生式，并且产生式右部均不含相连的非终结符，即没有 $P \to \varepsilon$ ，或 $P \to \cdots QR \cdots$ 形式的产生式，则G是一个算符文法。

例：

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

优先关系定义：算符文法G，对任何 $a,b \in V_T$，定义a与b之间的优先关系如下：

(1) a $\doteq$ b，G中有 $P \to \cdots ab \cdots$ 或 $P \to \cdots aQb \cdots$ 形式的产生式。

(2) a $\lessdot$ b，G中有 $P \to \cdots aQ \cdots$ 的产生式，且 $Q \overset{+}{\Rightarrow} b\cdots$ 或 $Q \overset{+}{\Rightarrow} Rb\cdots$ 。

(3) a $\gtrdot$ b，G中有 $P \to \cdots Qb \cdots$ 的产生式，且 $Q \overset{+}{\Rightarrow} \cdots a$ 或 $Q \overset{+}{\Rightarrow} \cdots aR$ 。

对 $E \to E+T$ ，

由 $T \overset{+}{\Rightarrow} i$ 得 + $\lessdot$ i

由 $T \overset{+}{\Rightarrow} T*F$ 得 + $\lessdot$ *

由 $T \overset{+}{\Rightarrow} (E)$ 得 + $\lessdot$ (

由 $E \overset{+}{\Rightarrow} i$ 得 i $\gtrdot$ +

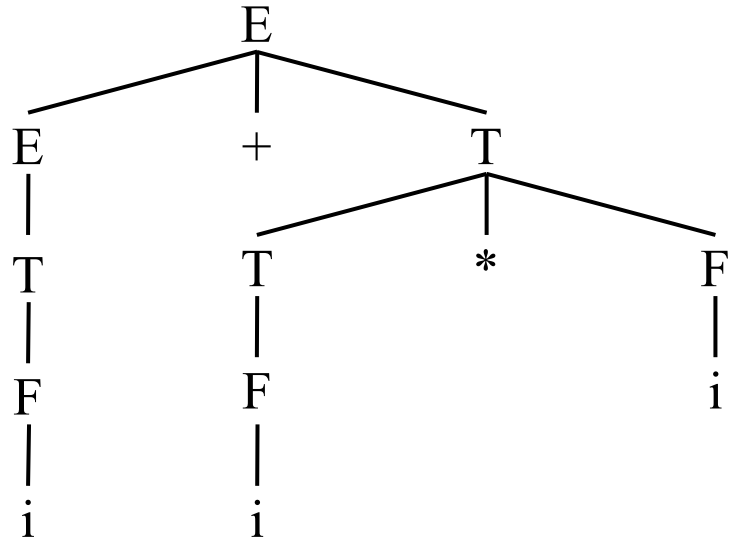
由 $E \overset{+}{\Rightarrow} (E)$ 得) $\gtrdot$ +

由 $E \Rightarrow E+T$ 得 + $\gtrdot$ +

	+	*	i	(	)	$
+	$\gtrdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
*	$\gtrdot$	$\gtrdot$	$\lessdot$			$\gtrdot$
i	$\gtrdot$	$\gtrdot$			$\gtrdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	
)	$\gtrdot$	$\gtrdot$			$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$		$\doteq$

二、算符优先分析方法

这也是一种表驱动的分析方法，除了优先关系表作为分析表外，还需要一个分析栈和一个分析控制程序，协同地进行语法分析。文法句子i+i*i的LR分析过程如下：



图： 相应的句柄规约

例子文法句子i+i*i的算法优先分析过程

分析栈	输入串	动作
$	i+i*i$	$<i
$i	+i*i$	i ＞ + 归约
$F	+i*i$	$ ＜ +
$F+	i*i$	+ ＜ i
$F+i	*i$	i ＞ * 归约
$F+F	*i$	+ ＜ *
$F+F*	i$	* ＜ i
$F+F*i	$	i ＞ $ 归约
$F+F*F	$	* ＞ $ 归约
$F+T	$	+ ＞ $ 归约
$E	$	结束

三、优先关系表的构造

$$FIRSTVT(P)=\{a \mid P\overset{+}{\Rightarrow}a\cdots 或 P\overset{+}{\Rightarrow}Qa\cdots, a\in V_T, Q\in V_N\}$$

$$LASTVT(P)=\{a \mid P\overset{+}{\Rightarrow}\cdots a 或 P\overset{+}{\Rightarrow}\cdots aQ, a\in V_T, Q\in V_N\}$$

构造FIRSTVT(P)可遵照下列两条规则:

- 若有产生式 $P\to a\cdots$ 或 $P\to Qa\cdots$ ，则 $a\in FIRSTVT(P)$ ；
- 若有产生式 $P\to Q\cdots$ 则 $FIRSTVT(Q)\subseteq FIRSTVT(P)$ 。

构造LASTVT(P)有类似的两条规则:

- 若有产生式 $P\to\cdots a$ 或 $P\to\cdots aQ$ ， 则 $a\in LASTVT(P)$;
- 若有产生式 $P\to\cdots Q$ 则 $LASTVT(Q)\subseteq LASTVT(P)$ 。

例

右面文法的FIRTVST集和LASTVT集，是这样得到的：

由F→ (E)|i，

FIRSTVT(F)={(, i }，LASTVT(F)={) , i }

由T→T*F|F，

FIRSTVT(T)={*, (, i }，LASTVT(T)={*,) , i }

由E→E+T|T，

FIRSTVT(E)={+, *, (, i }，LASTVT(E)={+, *,), i }

参照前节FIRST集的成员表表示法，本例的FIRST集、LASTVT集也可用成员表表示，其中用√表示FIRST集的成员，○表示LASTVT集的成员，则有下图：

$$E \to E+T \mid T$$

$$T \to T*F \mid F$$

$$F \to (E) \mid i$$

	+	*	(	)	i
E→E+T|T	√ ○	√ ○	√	○	√ ○
T→T*F|F		√ ○	√	○	√ ○
F→(E)|i			√	○	√ ○

于是，由 $E \to E+T$

有：+ ⋗ +，* ⋗ +，) ⋗ +，i ⋗ +；

+ ⋖ *，+ ⋖ (，+ ⋖ i；

由 $T \to T*F$

有：* ⋗ *，) ⋗ *，i ⋗ * ；

* ⋖ (，* ⋖ i；

由 $F \to (E)$

有：(≐)；

(⋖ +，(⋖ *，(⋖ (，(⋖ i；

+ ⋗)，* ⋗)，) ⋗)，i ⋗)

	+	*	i	(	)	$
+	⋗	⋖	⋖	⋖	⋗	⋗
*	⋗	⋗	⋖		⋗	⋗
i	⋗	⋗			⋗	⋗
(	⋖	⋖	⋖	⋖	≐	
)	⋗	⋗			⋗	⋗
$	⋖	⋖	⋖	⋖		≐

大作业

- 基本要求:从文本文件读入一个上下文无关的算符文法，构造算符优先分析表并以文本文件形式输出
  (M[a,b]=c ,其中: $a, b \in V_T \cup \$$,c='>'|'<'|'='|' ').
- 测试文法： 1.

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

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid i$$

2. E -> E + E | E * E | (E) | id

- 提交方式:canvas平台
- 提交时间:2020年6月21日

本章小结

- 语法分析器
- 上下文无关文法和文法设计,
- 推导、分析树和二义文法
- LL(1)语法分析器
- SLR语法分析器
- 规范LR语法分析器和LL(1)语法分析器
- 算符优先法

谢 谢!