4.6 简单的LR技术→∪Σεαβ∈∉

移进-归约算法 (栈, 输入缓冲区):

句型的句柄:

练习 1: S → S S + | S S \* | a

求 SSS + a \* +

S

SS

SSS

SSS+

SSS+a

SSS+a\*

SSS+a\*+

SS + a \* a +

aaa \* a ++

以上句型的句柄.

练习2: 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. 文法S → 0 S 1 | 0 1 的串 000111
2. 文法 S → S S + | S S \* | a的串aaa \* a ++

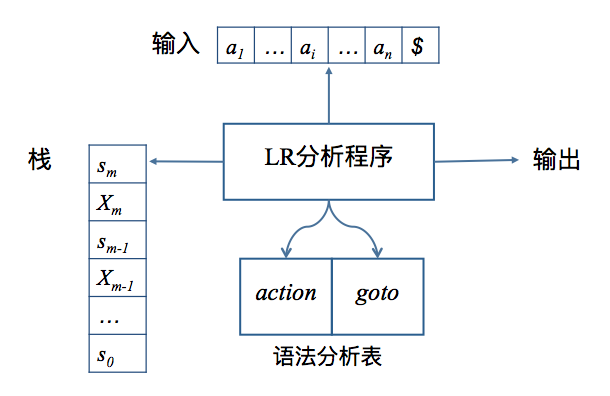
|  |  |  |
| --- | --- | --- |
| 栈 | 输入 | 动作 |
| $ | 000111$ | 移进 |
| $ 0 | 00111$ | 移进 |
| $ 0 0 | 0 111 $ | 移进 |
| $ 0 0 0 | 1 1 1 $ | 移进 |
| $ 0 0 0 1 | 1 1 $ | 归约 S 🡪 01 |
| $ 0 0 S | 1 1 $ | 移进 |
| $ 0 0 S 1 | 1 $ | 归约 S 🡪 0S1 |
| $ 0 S | 1 $ | 移进 |
| $ 0 S 1 | $ | 归约S🡪 0S1 |
| $ S | $ | accept |

|  |  |  |
| --- | --- | --- |
| 栈 | 输入 | 动作 |
| $ | aaa \* a ++$ | ?? |
| … | … | … |
| $ S | $ | accept |

LL (1) 文法: L 表示对输入进行从左到右的扫描；L构成出来的最左推导

LR(k)文法: L表示对输入进行从左至右的扫描; R表示(反向)构造出一个最右推导序列. k表示分析决定时向前看k个输入符号. 假设k=1, 即简写为LR文法.

4.6.3 LR语法分析算法



注: 栈存储形式为s0 X1 s1 X2 s2 … Xm sm的串，sm在栈顶。Xi是文法符号，si是叫做状态的符号，状态符号概括了栈中它下面部分所含的信息。栈顶的状态符号和当前的输入符号用来检索分析表，以决定移进-归约分析的动作。真正实现时，文法符号不必出现在栈里。

* 输入：一个输入串*w*和一个LR语法分析表。
* 输出：如果*w*在*L(G)*中，输出*w*的自底向上语法分析过程中的归约步骤；否则给出错误提示。
* 方法：最初，语法分析器栈中的内容为初试状态*S0*，输入缓冲区的内容为*w $*。然后，执行语法分析程序。

LR分析驱动程序 的行为是：它根据栈顶当前的状态sm和当前的输入符号ai，访问action[sm, ai], 它可能的4种值 如下：

(1) 移进s，其中s是一个状态

(2) 按文法产生式A → β归约

(3) 接受

(4) 出错

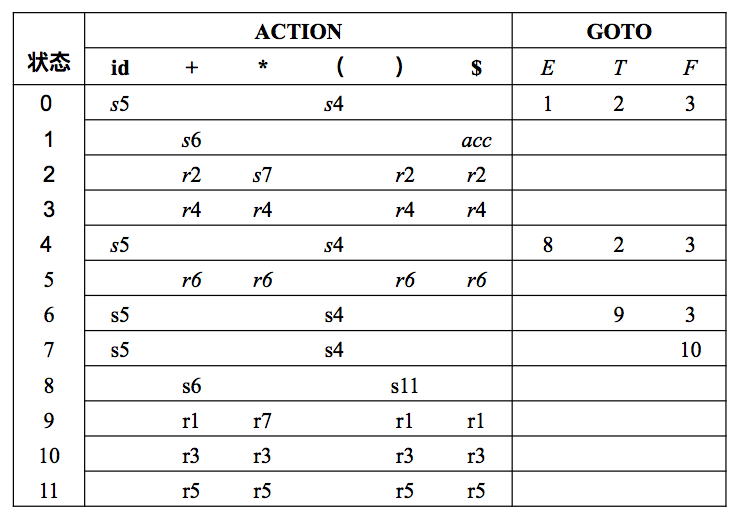
转移函数goto取状态和文法符号作为变元，产生一个状态。

例(1) 对于下列文法

(1) *E* → *E + T*  (2) *E* → *T*

(3) *T* → *T \* F* (4) *T* → *F*

(5) *F* → ( *E* ) (6) *F* → **id**



LR语法分析表

状态是什么？？？

ACTION 和 GOTO两个函数

ACTION[i, a], 其中i为状态, a为输入符号 (或者结束符$):

(1) sj: 移入状态j进状态栈, 将符号a移入符号栈

(2) rj: 使用第j个产生式A → β进行归约, 将状态栈的 |β| 个状态出栈, 并且将新的状态移入栈 (怎么求新的状态); 将符号栈的 |β| 个符号出栈, 将A移入符号栈;

(3) 接受 (acc)

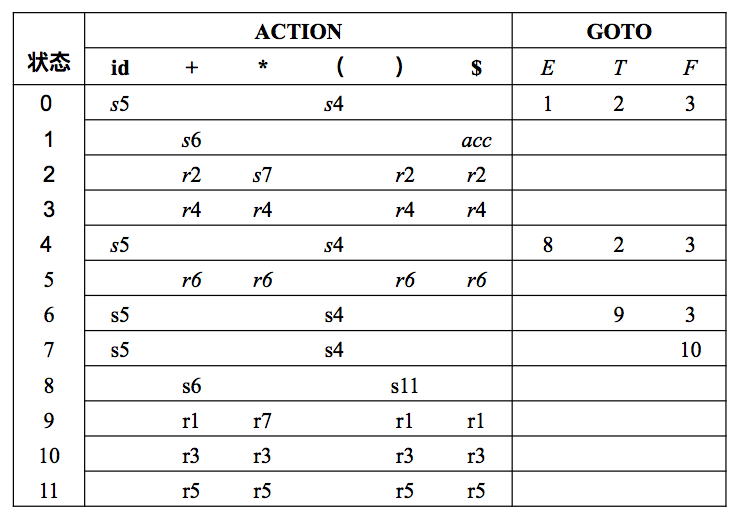
(4) 报错 (error)

(1) *E* → *E + T*  (2) *E* → *T*

(3) *T* → *T \* F* (4) *T* → *F*

(5) *F* → ( *E* ) (6) *F* → **id**

|  |  |  |  |
| --- | --- | --- | --- |
| 状态栈 | 符号栈 | 输入 | 动作 |
| $ 0 | $ | id + id \* id $ | s5 |
| $ 0 5 | $ id | + id \* id $ | r6(F->id) |
| $ 0 3 | $ F | + id \* id $ | r4 (T->F) |
| $ 0 2 | $ T | + id \* id $ | r2 (E->T) |
| $ 0 1 | $ E | + id \* id $ | s6 |
| $ 0 1 6 | $ E + | id \* id $ | s5 |
| $ 0 1 6 5 | $ E + id | \* id $ | r6 (F->id) |
| $ 0 1 6 3 | $ E + F | \* id $ | r4 (T->F) |
| $ 0 1 6 9 | $ E + T | \* id $ | s7 |
| $ 0 1 6 9 7 | $ E + T \* | id $ | s5 |
| $ 0 1 6 9 7 5 | $ E + T \* id | $ | r6 (F ->id) |
| $ 0 1 6 9 7 10 | $ E + T \* F | $ | r3 |
| $ 0 1 6 9 | $ E + T | $ | r1 |
| $ 0 1 | $ E | $ | acc |



id \* id + id的执行步骤

|  |  |  |  |
| --- | --- | --- | --- |
| 状态栈 | 符号栈 | 输入 | 动作 |
| $0 | $ | id \* id + id $ | s5 |
| $0 5 | $ id | \* id + id $ | r6 F🡪id |
| $ 0 3 | $ F | \* id + id $ | r4 T 🡪 F |
| $ 0 2 | $ T | \* id + id $ | S7 |
| $ 0 2 7 | $ T \* | id + id $ | S5 |
| $ 0 2 7 5 | $ T \* id | + id $ | R6 F->id |
| $ 0 2 7 10 | $ T \* F | + id $ | R3 |
| $ 0 2 | $ T | + id $ | R2 |
| $ 0 1 | $ E | + id $ | S6 |
| $ 0 1 6 | $ E + | id $ | S5 |
| $ 0 1 6 5 | $ E + id | $ | R6 |
| $ 0 1 6 3 | $ E + F | $ | R4 |
| $ 0 1 6 9 | $ E + T | $ | R1 |
| $ 0 1 | $ E | $ | acc |

id + id \* id的执行步骤

LR语法能分析特点:

概念：

可行前缀: 右句型的前缀，该前缀不超过最右句柄的右端

*S* ⇒\**rm γ A w*⇒*rm γ β w*

*γ β*的任何前缀（包括ε和*γ β*本身）都是可行前缀

假设

*E* ⇒\**rm F* \* **id** ⇒*rm* ( *E* ) \* **id**

可行前缀可以是 (，( *E*，( *E* )，但不会是( *E* ) \*，因为( *E* )是句柄( *F* → ( *E* ) )，语法分析器必须在移入\*之前将( *E* )归约成*F*。

LR文法：我们能为之构造出所有条目都唯一的LR分析表。

直观上说，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。