编译原理

Compiler Construction Principles

朱 青

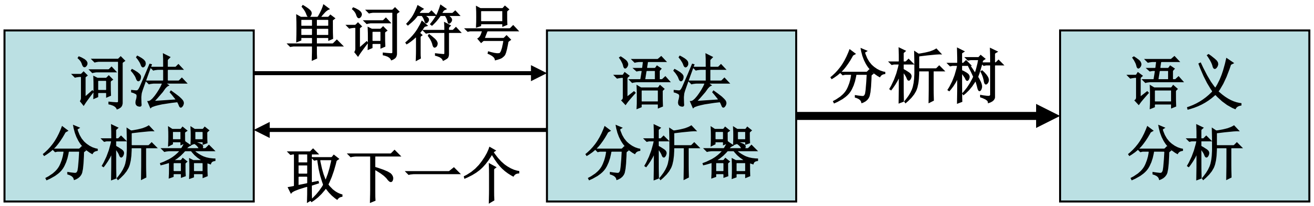
信息学院计算机系，

中国人民大学，

zqruc2012@aliyun.com

第3章: 程序语言的语法描述与分析

语法分析的重要性:



语法分析方法:

1)自顶向下分析法.
2)自底向上分析法.

第3章: 程序语言的语法描述与分析

- ⌘3.1 上下文无关文法
- ⌘3.2 自顶向下分析法
- ⌘3.3 递归下降分析法
- ⌘3.4 LL(1)分析法
- ⌘3.5 LL(1)分析器

3.1上下文无关文法

3.1.1文法与语言

文法是描述语言的语法结构的形式规则(即语法规则)

上下文有关文法:

上下文无关文法:

-----在编译中通指此文法.

一.上下文无关文法:

例:只含*, +和( )的算术表达式的文法

$$E \longrightarrow E+E|E*E|(E)|a \quad (3.1)$$

定义:一个上下文无关文法是一个四元式(V_T, V_N, S, P)

其中:1) V_T是非空有限集, 每个元素是一个终结符.

2) V_N是非空有限集, 每个元素是一个非终结符.

3) S是一个非终结符,是开始符号.

4) P是产生式的集合:它的每一个产生式的形式为 $P \longrightarrow \alpha$.

其中：P属于V_N，α属于$(V_N \cup V_T)^*$.

S在产生式的左部必须至少出现一次.

为简单起见，文法可以只给出产生式集合，并指出开始符号.

例如：

$E \longrightarrow EAE \mid a$　　　$V_T = \{a, +, *\}$

$A \longrightarrow + \mid *$　　　$V_N = \{E, A\}$

E为开始符号.　　　$S = \{E\}$

缩写为：$E \longrightarrow EAE \mid a$

$A \longrightarrow + \mid *$

左部为非终结符号，右部为后选式.

二.由文法产生语言:

文法(3.1)可以生成句子:(a*a+a)

E==>(E)==>(E*E)==>(a*E)

==>(a*E+E)==>(a*a+E)==>(a*a+a)

==>:表示推导.

- 基本术语定义:

1) 称$\alpha A\beta$ 直接推导出$\alpha\gamma\beta$,仅当有$A \longrightarrow \gamma$ 且 $\alpha,\beta \in (V_T \cup V_N)^*$.

表示为$\alpha A\beta \Rightarrow \alpha\gamma\beta$.

2) 若有$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$,则称<u>从α_1推导出α_n .</u>

3) $\alpha_1 \overset{*}{\Rightarrow} \alpha_n$ 表示从α_1 经过0次或若干次可推导出α_n.

$\alpha_1 \overset{+}{\Rightarrow} \alpha_n$ 表示从α_1 经过1次或若干次可推导出α_n.

若$\alpha \overset{*}{\Rightarrow} \beta$ 表示 或者$\alpha = \beta$ 或$\alpha \overset{+}{\Rightarrow} \beta$.

4) 若G是一个文法,S是开始符号,若有$S \overset{*}{\Rightarrow} \alpha$,则称α为句型.仅含终结符 的句型叫句子.

文法G所生成的语言为:

$L(G)=\{\alpha \mid S \overset{+}{\Rightarrow} \alpha$ 且 α 属于 $V_T^*\}$

5)最左(右)推导

每次直接推导的是最左(右)的非终结符.

6)短语,直接短语:

若G是一个文法,S是开始符号,

假定αβχ是文法G的一个句型,如果有

$$S \overset{*}{\Longrightarrow} \alpha A \chi \quad 且 \quad A \overset{+}{\Longrightarrow} \beta$$

则称β是一个关于非终结符号A的,句型αβχ的<u>短语</u>.

如果有

$$S \overset{*}{\Longrightarrow} \alpha A \chi \quad 且 \quad A \Longrightarrow \beta$$

则称β是<u>直接短语.</u>

一个句型的最左直接短语称为该句型的句柄.

例如:考虑一个文法G=({a,b},{S},S,P)

其中P:

$$S \longrightarrow aSb$$

$$S \longrightarrow ab$$

$$S \Longrightarrow aSb \Longrightarrow aaSbb$$

$$\Longrightarrow a^3Sb^3 \Longrightarrow \ldots$$

$$\Longrightarrow a^{n-1}Sb^{n-1}$$

$$\Longrightarrow a^nb^n (n>=1)$$

$$L(G)=\{a^nb^n \mid n>=1\}$$

例如: 以下述文法为例说明(a*a+a)

是文法的句子,及术语举例.

$$E \longrightarrow T|E+T$$

$$T \longrightarrow F|T*F \qquad (3.2)$$

$$F \longrightarrow (E)|a$$

解: E=>T=>F=>(E)=>(E+T)

=> (T+T) =>(T*F+T)   (3.3)

=>(F*F+T)=>(a*F+T)

=>(a*a+T)=>(a*a+F)=>(a*a+a)

所以 (a*a+a)是文法G的<u>句子</u>.

而 T, F,(E),(E+T) , (T+T),(T*F+T),

(F*F+T),(a*F+T),(a*a+T),(a*a+F),

都是文法G的<u>句型</u>.

<u>最右推导</u>: E=>T=>F=>(E)=>(E+T)

=> (E+F) =>(E+a)

=>(T*F+a)=>(T*a+a)

=>(F*a+a)=>(a*a+a)

考虑文法(3.2)的一个句型a1*a2+a3:

- a1,a2,a3,a1*a2 都是句型a1*a2+a3的短语,
- a1,a2和a3是直接短语,
- a1是最左直接短语,
- a2+a3不是句型a1*a2+a3短语,因为有E=>a2+a3但不存在从文法的开始符号E到a1*E的推导.

3.1.2 语法树和二义性

上述概念可以用一张图(语法树)表示.

如文法(3.1)的如下推导的语法树可表示: E=>(E)

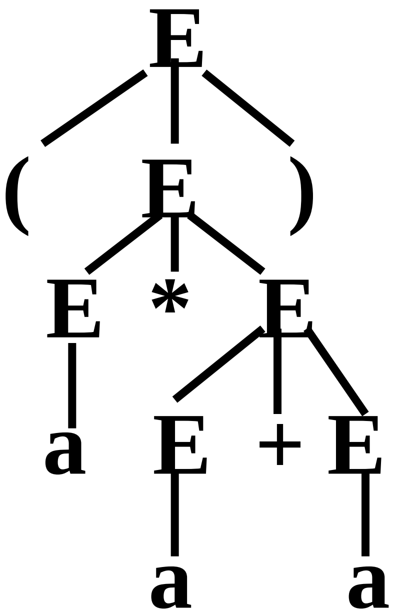
=>(E*E)

=>(a*E)

=>(a*E+E)

=>(a*a+E)

=>(a*a+a)



(3.4)

文法(3.1)的如下推导的语法树可表示:

E==>(E)

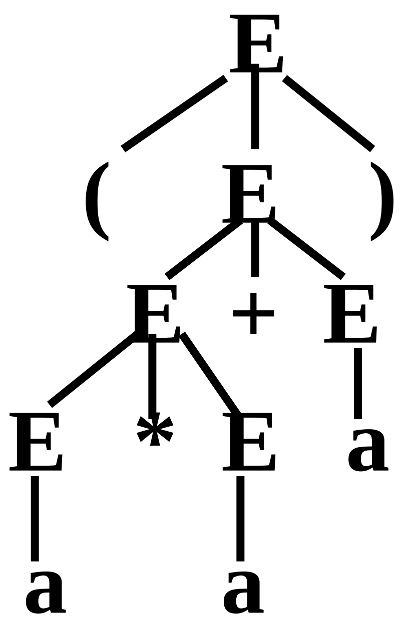
==>(E+E)

==>(E*E+E)

==>(a*E+E)

==>(a*a+E)

==>(a*a+a)



(3.5)

若某文法存在一个句子,对应两个不同的语法树,则称该文法是二义的.

已经证明,文法的二义性是不可判定的.可以为某些二义性找到充分条件,使其变成非二义的.

例如: 上述文法规定先乘后加,

同级运算左结合,即:

$$E \longrightarrow T|E+T$$

$$T \longrightarrow F|T*F$$

$$F \longrightarrow (E)|a$$

此文法无二义性.

例如：**PASCAL**语言关于**IF**语句的文法有下列规则:

(E--表达式**,S--**语句**,other--**非**IF**语句**)**

S ⟶ if E then S

| if E then S else S

|other

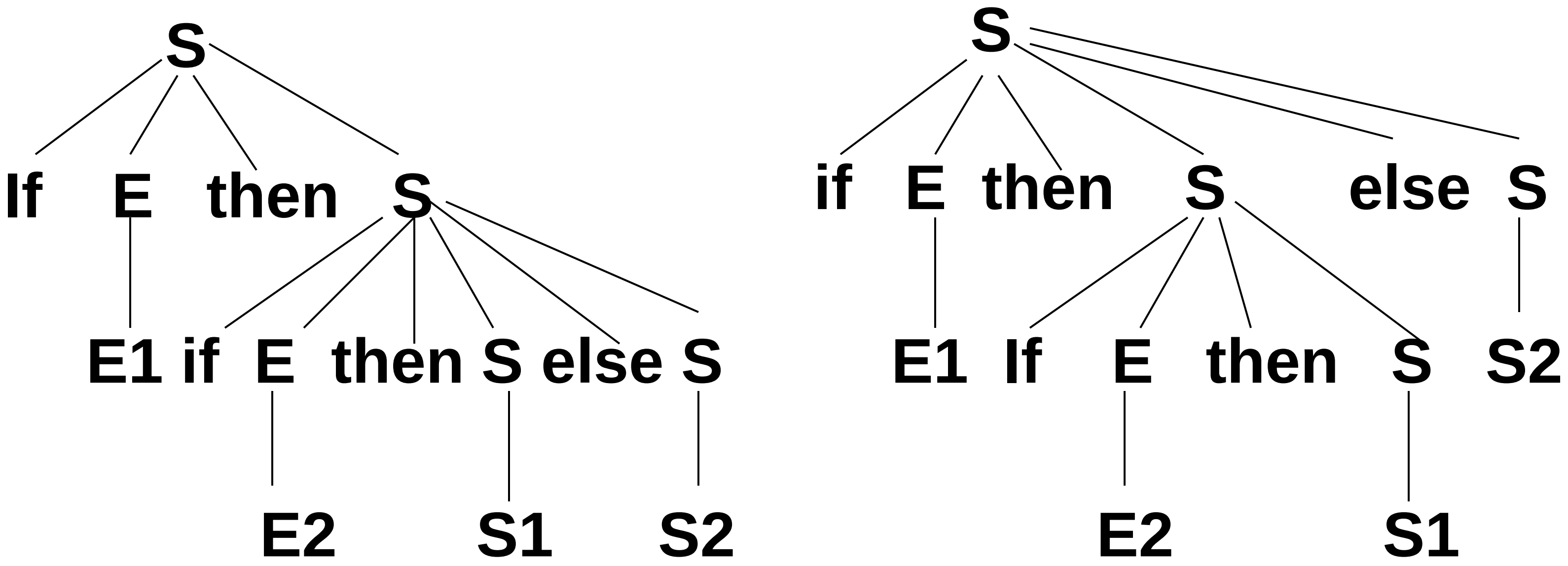
判断此文法是否为二义文法.

解:　　对复合条件语句:

if E1 then if E2 then S1 else S2

有两棵语法树(如下所示)

故此文法是二义的.



改写文法为下列的无二义性文法:

(M_S为匹配的, UN_S为非匹配的)

S ⟶ M_S

| UN_S

M_S ⟶ if E then M_S else M_S

| other

UN_S ⟶ if E then S

| if E then M_S else UN_S

3.1.3 形式语言概观

通常将文法分成四类:

0型,1型,2型,3型.

0型>1型>2型>3型. (‘>’---强于)

0型语言:递归可枚举语言.

1型语言:上下文有关语言.

2型语言:上下文无关语言.

3型语言:正规语言.

文献已经证明:这四类语言分别可被四种自动机所识别.

<u>图灵机</u>**(Turing machine---TM)**识别**0**型语言**(**递归可枚举语言**).**

<u>线性界限自动机</u>**(linear bounded automata---LBA)**识别上下文有关语言**.**

下推自动机 (push down automata---PDA) 识别上下文无关语言.

有限自动机 (finite automata ---FA) 识别正规语言.

文献已经证明:各类文法所产生的语言恰好与各类自动机所识别的语言相同.

定义: **(0型文法)**

$G=(V_N,V_T,S,P)$是一个**0**型文法,如果他的每个产生式

$$\alpha \longrightarrow \beta$$

是这样的一种结构: α属于$(V_N \cup V_T)^*$且至少含一个非终结符而β属于$(V_N \cup V_T)^*$.

如果对0型文法加上以下的第i条限制,就得到i型文法.

1. G的任何产生式$\alpha \rightarrow \beta$均满足$|\alpha|<|\beta|$或$|\alpha|=|\beta|$；仅仅$S \rightarrow \varepsilon$例外，但S不得出现在任何产生式的右部.

2. G的任何产生式为$A \rightarrow \beta$,
   A属于V_N 且β属于$(V_N \cup V_T)^*$.

3. G的任何产生式为A⟶αB或A→α,

其中, α属于V_T*, A,B属于V_N.

0型文法：称短语文法(无限制文法).

1型文法：称上下文有关文法.

2型文法：称上下文无关文法.

3型文法：称右线性文法,正规文法.

文法与语言举例:

例1: 3型文法G:

含有下列产生式的集合:

$S \longrightarrow 0|C0$

$C \longrightarrow B1$

$B \longrightarrow 0|C0$

$D \longrightarrow 1|C1|D0|D1|B0$

此文法构造的NFA能够且只能够识别所产生的正规语言.

例2: 上下文无关文法:

$$S \longrightarrow aSb|ab$$

此文法产生的语言:

$$L = \{ a^n b^n \mid n >= 1 \}$$

例3:上下文有关文法:

$S \longrightarrow aSAB$

$S \longrightarrow abB$

$BA \longrightarrow BA'$

$BA' \longrightarrow AA'$

$AA' \longrightarrow AB$

$bA \longrightarrow bb$

$bB \longrightarrow bc$

$cB \longrightarrow cc$

此文法产生的语言:

$$L = \{ a^n b^n c^n \mid n >= 1 \}$$

例4: 0型语言

$$L_0=\{ \alpha c \alpha \mid \alpha \in (a|b)^* \}$$

只有0型文法才能生成。

第3章: 程序语言的语法描述与分析

- ⌘3.1 上下文无关文法
- ⌘3.2 自顶向下分析法
- ⌘3.3 递归下降分析法
- ⌘3.4 LL(1)分析法
- ⌘3.5 LL(1)分析器

3.2 自顶向下分析法

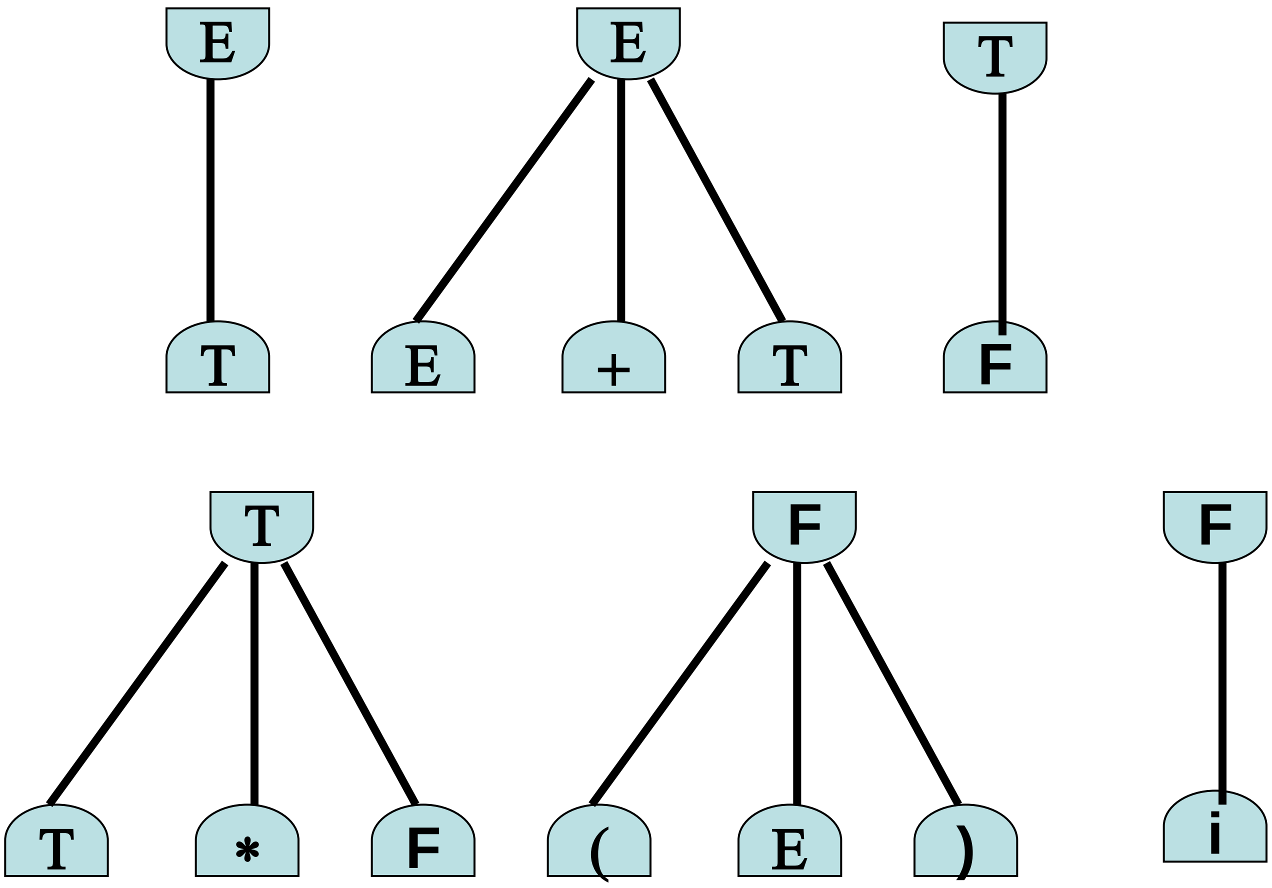
语法分析由两类:自顶向下分析法和自底向上分析法.下面用股子游戏来演示.如:文法

$$E \longrightarrow T|E+T$$

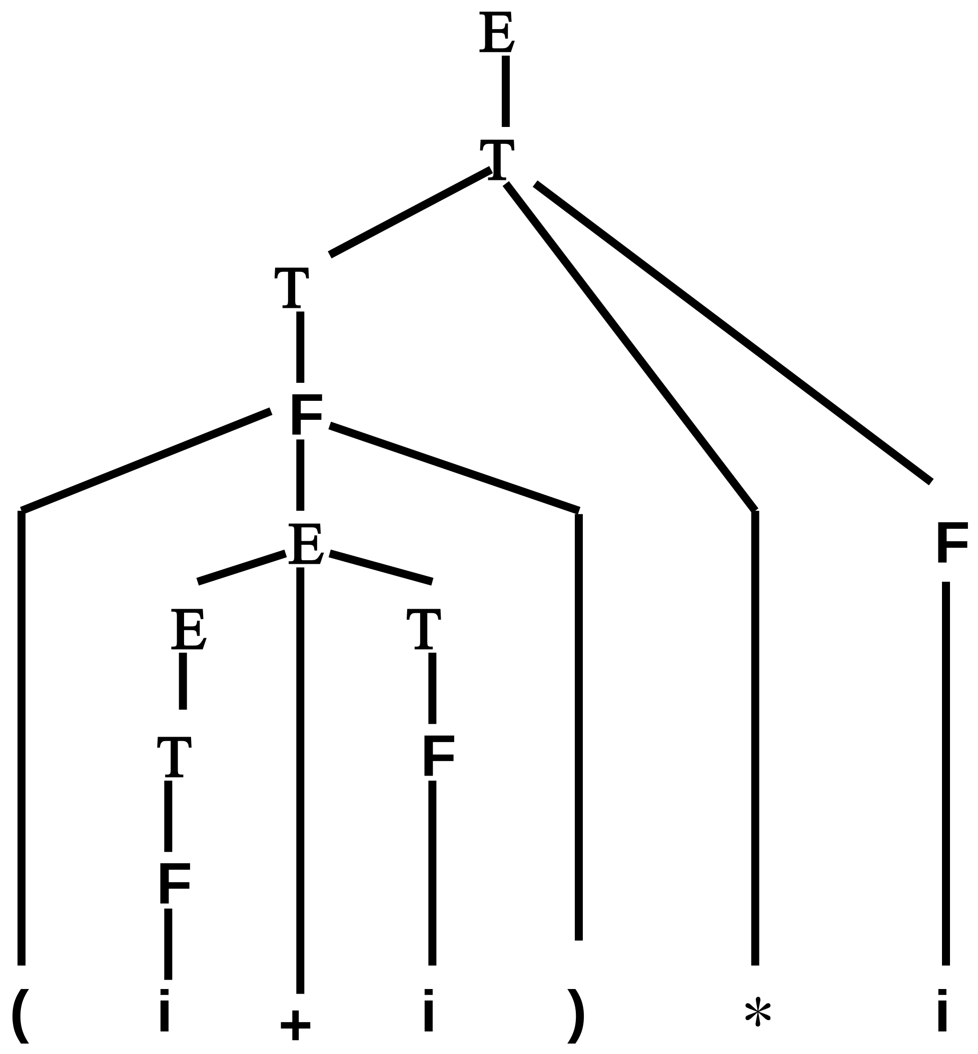
$$T \longrightarrow F|T*F$$

$$F \longrightarrow (E)|\ i$$

所有的股子为:



若要分析符号串 (i+i)*i初始态为:



自上而下的分析法:从句子符号到句子,
自下而上的分析法: 从句子到句子符号.

分析开始时,句子从左向右读,则最自然,也可以从右向左读,则也同样方便.

注意:不替换已经使用的产生式.如果替换,就是回退.叫回溯.

所谓确定与非确定的分析,取决于是否发生回溯.

第3章: 程序语言的语法描述与分析

- ⌘3.1 上下文无关文法
- ⌘3.2 自顶向下分析法
- ⌘3.3 递归下降分析法
- ⌘3.4 LL(1)分析法
- ⌘3.5 LL(1)分析器

3.3 递归下降分析法

是一种易于实现的自上而下的分析法.可以从适当的文法出发写一个语法程序.

考虑下面的文法生成的语言:

PROGRAM → begin DECLIST comma STATELIST end

DECLIST → d semi DECLIST

DECLIST → d

STATELIST → s semi STATELIST

STATELIST → s

(这个语言的形式结构: **begin d;d;…;s;s;…;s end**).

这个文法稍加改造,可得:

PROGRAM $\longrightarrow$ begin DECLIST comma STATELIST end
DECLIST $\longrightarrow$ dx
X $\longrightarrow$ semi DECLIST
X $\longrightarrow$ ε
STATELIST $\longrightarrow$ sY
Y $\longrightarrow$ semi STATELIST
Y $\longrightarrow$ ε

设:有以下程序:

1. error是一个出错处理程序.
2. **semi, comma**是终结符.
3. **Lexical**是一个词法分析程序,它提供下一个单词.

4. 语法分析程序执行之前,已经执行过:

symbol:=lexical;

用递归下降分析法写语法分析程序时,只需给每个非终结符写一个过程.

```
PROC PROGRAM;
BEGIN
    IF symbol <> begin  THEN  error;
    symbol:=lexical;    DECLIST;
    IF symbol <> comma  THEN  error;
    symbol:=lexical;    STATELIST;
    IF symbol <> end  THEN  error
END;
```

```
PROC DECLIST;
BEGIN
     IF symbol <> d  THEN  error;
     symbol:=lexical;   X;
END;

PROC  X;
BEGIN
     IF symbol =semi  THEN
     BEGIN   symbol:=lexical;    DECLIST  END
     ELSE  IF symbol = comma
               THEN  return
               ELSE error;
END;
```

```
PROC STATELIST;
BEGIN
    IF symbol <> s  THEN  error;
    symbol:=lexical;   Y;
END;

PROC  Y;
BEGIN
    IF symbol =semi  THEN
    BEGIN   symbol:=lexical;
              STATELIST       END
    ELSE  IF symbol = end
            THEN  return
            ELSE error;
END;
```

用这样的语法分析句子,因多次的递归调用,效率很低,可以用重复来代替递归.

DECLIST ⟶ d(semi d)*

STATELIST ⟶ s (semi s)*

递归下降分析程序改写:

PROC PROGRAM 保留.

```
PROC  DECLIST;
BEGIN
    IF   symbol &lt;&gt; d  THEN error;
```

```
    symbol := lexical;
    WHILE  symbol =semi  DO
    BEGIN
       symbol := lexical;
       IF symbol <> d THEN error;
       symbol := lexical;
    END
END;
```

```
PROC   STATELIST;
  BEGIN
      IF   symbol <> s  THEN error;
     symbol := lexical;
      WHILE  symbol =semi  DO
      BEGIN
        symbol := lexical;
        IF symbol <> s THEN error;
        symbol := lexical;
     END
 END;
```

用重复代替递归可较前的方法提高效 率即改善可读性.

<u>文法的另一种表示法:</u>

- 在LEX中规定辅助定义式:Di ⟶Ri的右部Ri只允许出现Σ中的字符和已定义过的简名D1,…,Di-1,即Di不能递归调用.去掉这一条,允许递归调用.这种定义系统就可以表示上下文无关文法.

现在许多程序设计语言的语法是用这种定义系统来描述的.

某些符号的记法:(用重复代替递归)

1)用花括号$\{a\}$表示闭包运算a^*.

2)用$\{a\}_0^n$表示a可任意重复0次至n次.

$\{a\}_0^0=a^0=\varepsilon$.

3)用方括号[a]表示$\{a\}_0^1$,即表示a的出现可有可无(等价于$a|\varepsilon$).

例如:文法

$E \longrightarrow T|E+T$ 可表示为:$E \longrightarrow T\{+T\}$

$T \longrightarrow F|T*F$ $T \longrightarrow F\{*F\}$

```
PROCEDURE  E;
BEGIN
    T;
   WHILE  SYM='+' DO
   BEGIN   ADVANCE; T END;
END;

PROCEDURE  T;
BEGIN
    F;
   WHILE  SYM='*' DO
   BEGIN   ADVANCE; F END;
END;
```

(ADVANCE 指向下一个输入符号).

递归下降分析法:

优点: 1) 从适当的文法出发写一个递归下降分析器很容易.

2) 分析器和文法很相似,故分析器的可靠性很高.

缺点: 1)因不断的递归调用,分析速度慢.

2)分析器本身比较大.

3)会引起分析过程中插入代码生成,而导致编译的不同阶段混在一起.给软件维护带来困难,并过早地依赖于机器.

左递归与公共左因子

1.消除左递归

<u>定义:</u> (左递归)

如果文法有一个非终结符号**A,** 存在推导 **A==>A** α,α为某个符号串,该文法是*左递归的.

自顶向下文法不能处理左递归文法.

简单情况下,如文法的非终结符号A的产生式为

$$A \longrightarrow A\alpha|\beta$$

其中β不以α打头,那么可以把A的两个规则改写如下的非左递归产生式:

$$A \longrightarrow \beta A'$$

$$A' \longrightarrow \alpha A'|\varepsilon$$

这里A是直接左递归的.

考虑下述表达式文法:

$$E \longrightarrow E+T \mid T$$
$$T \longrightarrow T*F \mid F \qquad (3.10)$$
$$F \longrightarrow (E) \mid id$$

经消除直接左递归后变成:

$$E \longrightarrow TE'$$
$$E' \longrightarrow +TE' \mid \varepsilon$$
$$T \longrightarrow FT'$$
$$T' \longrightarrow *FT' \mid \varepsilon$$
$$F \longrightarrow (E) \mid id$$

一般而言,假定关于A的全部产生式是:

$A \longrightarrow A\alpha 1|A\alpha 2|\ldots|A\alpha m|\beta 1|\beta 2|\ldots|\beta n$

其中 βi(I=1,2,…,n)不以A打头;

αj(j=1,2,…m)不等于ε,

那么可以把上述产生式改写为:

$A \longrightarrow \beta 1\ A'|\beta 2\ A'\ |\ldots|\beta n\ A'$

$A' \longrightarrow \alpha 1A'|\alpha 2A'|\ldots|\alpha mA'|\varepsilon$

此法易于消除见于表面的直接左递归，但不意味着已经消除整个文法的左递归性。例如文法：

$$S \longrightarrow Aa \mid b$$

$$A \longrightarrow Ac \mid Sd \mid \varepsilon$$

非终结符号S是左递归的，因为有 S==>Aa==>Sda，但它不是直接左递归的。

算法**3.1:**消除左递归**:**

输入**:** 无环路无ε**-**产生式的文法**G.**

输出**:** 不带有左递归的一个等价文法**.**

1. 把文法**G**的所有非终结符号按一种顺序排列成 **A1,A2, … ,An.**

2. for i:= 1 to n do

for j:=1 to i-1 do

begin

把每一个形如**Ai→Ajγ**的规则改写成

$$Ai \longrightarrow \delta 1\gamma \mid \delta 2\gamma \mid \ldots \mid \delta k\gamma$$

其中 $Aj \longrightarrow \delta 1 \mid \delta 2 \mid \ldots \mid \delta \kappa$ 是关于当前Aj的所有产生式;

消除关于Ai-产生式的直接左递归性.

end

3.化简由(2)所得到的文法,即去除那些从开始符号出发永远无法到达的非终结符号的产生式.

例如:文法如下,消除左递归.

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd \mid \varepsilon$$

解: 改写文法为ε–无关文法.

$S \rightarrow Aa$	$S \rightarrow Aa \mid \varepsilon a$ (即 $S \rightarrow Aa \mid a$)
$S \rightarrow b$	$S \rightarrow b$
$A \rightarrow Ac$	$A \rightarrow Ac \mid \varepsilon c$
$A \rightarrow Sd$	$A \rightarrow Sd$
$A \rightarrow \varepsilon$	无

现在考虑对改写后的文法:

$S \longrightarrow Aa|a|b$

$A \longrightarrow Ac|c|Sd$

根据算法4.1

1)非终结符安排为 S,A

2) S不存在左递归, 故i=1不做工作.

3)i=2时, 将$S \longrightarrow Aa|a|b$代入到A的有关候选式后, 得到

$A \longrightarrow Ac \mid c \mid Aad \mid ad \mid bd$

再消除**A-**产生式中的直接左递归**:**

$S \rightarrow Aa|a|b$

$A \rightarrow cA'|adA'|bdA'$

$A' \rightarrow cA'|adA'|\varepsilon$

例如:消除文法(3.17)的左递归

$S \rightarrow Qc|c$

$Q \rightarrow Rb|b$

$R \rightarrow Sa|a$

(P83)

2. 提取公共左因子:

提取公共左因子是对文法改写的一种方法,为适应预测分析法.

例如:考虑以下两个产生式:

stmt $\longrightarrow$ if expr then stmt else stmt

| if expr then stmt

当输入 **if** 时,选择哪个产生式展开 **stmt**. 故提取公共左因子.

一般地,如果有产生式 $A \longrightarrow \alpha\beta1|\alpha\beta2$ 是两个产生式,那么可以提取左因子.

产生式变为:

$$A \longrightarrow \alpha A'$$

$$A' \longrightarrow \beta1|\beta2$$

同样,若有 $A \longrightarrow \alpha\beta1|\alpha\beta2| \ \ldots \ |\alpha\beta n|\gamma$,

其中 γ是不以α开头的候选式,那么这些产生式可用如下的式子代替:

$$A \longrightarrow \alpha A' | \gamma$$

$$A' \longrightarrow \beta 1 | \beta 2 | \ldots | \beta n$$

A’是一个新的非终结符号.

例如: 下面的文法是“未定的else”文法的简写形式.

$$S \longrightarrow iEtS | iEtSeS | a$$

$$E \longrightarrow b \qquad (3.13)$$

其中: i,t,e分别表示if then else,

提取公共左因子后,文法为:

$$S \longrightarrow iEtSS'|a$$

$$S' \longrightarrow eS|\varepsilon \qquad (3.14)$$

$$E \longrightarrow b$$

文法(3.13) 和 (3.14)都是二义的,需进一步处理.

例如: 文法:

$$S \longrightarrow aSb$$

$$S \longrightarrow aSc$$

$$S \longrightarrow \varepsilon$$

提取公共左因子后文法形式

$$S \rightarrow aSX$$
$$X \rightarrow b$$
$$X \rightarrow c$$
$$S \rightarrow \varepsilon$$

第3章: 程序语言的语法描述与分析

- ⌘3.1 上下文无关文法
- ⌘3.2 自顶向下分析法
- ⌘3.3 递归下降分析法
- ⌘3.4 LL(1)分析法
- ⌘3.5 LL(1)分析器

3.4 LL(1)分析法

LL(1)文法是上下文无关文法中的一种.用它可以写确定的自上而下的语法分析器.LL(1)是S-文法的一般化.一个S-文法满足下面的二个条件:

1)每个产生式右部的第一个符号为终结符.

2)若每个非终结符有若干个候选式,

则这些候选式的第一个符号彼此不同.

(这可以帮助我们做确定的分析.)

例如:看下面的文法是否为S-文法.

$$S \longrightarrow pX$$
$$S \longrightarrow qY$$
$$X \longrightarrow aXb$$
$$X \longrightarrow x$$
$$Y \longrightarrow aYd$$
$$Y \longrightarrow y$$

是S-文法.

例如: 下面的文法不是S-文法:

$$
\begin{aligned}
S &\longrightarrow R \\
S &\longrightarrow T \\
R &\longrightarrow pX \\
T &\longrightarrow q \\
X &\longrightarrow x \\
Y &\longrightarrow aYd \\
Y &\longrightarrow y
\end{aligned}
$$

不是S-文法.

虽然他们产生的是同一种语言.但不是同一个文法.

在某些情况下,一个非**S-**文法可以改造成**S-**文法,而不影响所生成的语言.

下面对串**paaaxbbb**进行分析,看是否为上述文法的句子:

S=>pX=>paXb=>paaXbb

=>paaaXbbb=>paaaxbbb.

可见,用**S-**文法可以很容易地写确定的自上而下的语法分析器.

LL(1)文法虽然是**S-**文法的一般化**,** 但从**LL(1)**文法写确定的自上而下的语法分析器也很容易**.**

LL(1) :L--从左向右读符号串**.**

L--左推导**.**

1--在分析时**,**只要超前搜索一个符号**,**来决定要选着哪个候选式**.**

在LL(1)文法中,某些产生式右部的第一个符号虽然不是终结符,但从右部的第一个 符号(非终结符)可推出一组以终结符开头的串来,这些终结符组成一个集合.

例如: 对文法

$$S \longrightarrow Ry$$

$$S \longrightarrow Tz$$

$$R \longrightarrow pacb$$

$$T \longrightarrow qayd$$

当从S出发推导时,

1)若在被分析的第一个符号是p,则应使用 $S \rightarrow Ry$.

2)若在被分析的第一个符号是q,则应使用 $S \rightarrow Tz$.

定义: (非终结符的首符号集合)

a属于first(A) $\Longleftrightarrow$ $A \overset{*}{\Rightarrow} a\alpha$

其中: first(A)是A的首符号集合,A为非终结符, α为字串.

例如:文法:

$P \longrightarrow Ac$

$P \longrightarrow Bd$

$A \longrightarrow a$

$A \longrightarrow Aa$

$B \longrightarrow bB$

first(P)={a, b}

<u>定义</u>:(一个串的首符号集合)

a属于first(α) <==> $\alpha \stackrel{*}{\Longrightarrow} a\beta$

判断LL(1)文法的必要条件是:

一个非终结符,若有多余一个的候选式,则这些后选式的首符号集合,彼此不相交.

注意: 当某些串中能推出ε时要特别留神.

例如:

$P \longrightarrow AB$

$P \longrightarrow BG$ 不是LL(1)文法.

$A \longrightarrow Aa$ 因为

$A \longrightarrow \varepsilon$ S(AB)={a,c,b}

$B \longrightarrow c$ S(BG)={c,b}

$B \longrightarrow bB$

再考虑文法:

$T \longrightarrow AB$

$A \longrightarrow PQ$

$A \longrightarrow BC$

$P \longrightarrow pP$

$P \longrightarrow \varepsilon$

$Q \longrightarrow qQ$

$Q \longrightarrow \varepsilon$

$B \longrightarrow bB$

$B \longrightarrow e$

$C \longrightarrow d$

$C \longrightarrow f$

S(PQ)={p,q}

S(BC)={e,b}

看上去彼此不相交,但当分析的串的首符号为:b,e时推导就不唯一.

T=>AB=>BCB=>...

T=>AB=>PQB=>B...

都可以成立.这就是不确定性.是不确定的分析.

定义: (引导符号集)

一个非终结符P的一个给定候选α的引导符号集为:

$$DS(P,\alpha)=\{a|a\in first(\alpha)\ \ or\ (\alpha=>\varepsilon,\ a\in F(P))\}$$

其中：F(P)为文法所有产生式中能跟在P后面的符号集合。如上例中

DS（A，PQ）={p，q，b，e}

DS（A，BC）={b，e}

定义**LL(1)**文法:

若一个文法,它的每一个非终结符,有多余一个的候选式,则这些候选式的引导符号集,彼此不相交.这样的文法就是**LL(1)**文法.

LL(1)文法能进行确定的自上而下的文法分析.

下面给出形式化描述及判定**LL(1)**的算法:

判定LL(1)的算法:

步骤:

1)求文法符号串的 首符号集first.

2)求文法符号串的 后随符号集follow.

3)求文法产生式的引导符号集(选用集)select.

4)判定一个文法是否为LL(1)的.

首先,形式化定义首符号集first 和后随符号集follow.

定义1:(首符号集first)

假定α是文法G的任意符号串,定义

$$\text{first}(\alpha)=\{a \mid \alpha \stackrel{*}{\Rightarrow} a\ldots,\ \ a \in V_T\}$$

特别是，若$a \stackrel{*}{\Rightarrow} \varepsilon$则规定$\varepsilon \in \text{first}(a)$.

定义2:(后随符号集follow)

假定S是文法G的开始符号,对于G的任何非终结符A,定义

$$\text{follow}(A)=\{a \mid S \stackrel{*}{\Rightarrow} \ldots Aa\ldots,\ \ a \in V_T\}$$

特别是,若$S \stackrel{*}{\Rightarrow} \ldots A$,则规定$\# \in \text{follow}(A)$.

一.求文法符号串的 首符号集first 的算法:

应用下列规则,直到再没有任何终结符号或ε能加到该首符号集为止.

(1)如果X是终结符,则first(X)={X}.

(2)如果$X \in V_N$,且有产生式$X \longrightarrow a\ldots$,则把a加入到first(X)中;若$X \longrightarrow \varepsilon$也是一个产生式,则$\varepsilon \in first(X)$.

(3)若$X \longrightarrow Y \ldots$是一个产生式且X是非终结符,则把first(Y)中的所有非ε-元素都加到first(X)中;

若$X \longrightarrow Y1Y2\ldots Yi$是一个产生式, Y1,...,Yi-1都是非终结符,而且对任何j, 1<=j<=i-1 ，first (Yj) 都含有ε (即 Y1Y2…Yi-1==>ε) , 则把first(Yj)中的所有非ε-元素都加到first(X)中;

特别是,若所有的first(Yj)均含有ε , j=1,2,…,k, 则把ε加到first(X)中.

例如: 给定文法,求其各非终结符相关的串的首符号集.

$$
\begin{aligned}
E &\longrightarrow TE' \\
E' &\longrightarrow +TE' \mid \varepsilon \\
T &\longrightarrow FT' \qquad\qquad (3.16) \\
T' &\longrightarrow *FT' \mid \varepsilon \\
F &\longrightarrow (\ E\ ) \mid id
\end{aligned}
$$

解: 按上述规则:

first(F)={ (, id }   first(T')={ * , ε }

first(E')={ + , ε }

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

二.求文法符号串的 后随符号集**follow** 的算法:

应用下列规则,直到每个后随符号集**follow**不再增大为止.

1) 对于文法的开始符号**S**,置**#**于**follow(S)**中;

2)若$A \rightarrow \alpha B\beta$是一个产生式,则把 $first(\beta) \setminus \{\varepsilon\}$加至**follow(B)**中;

3)若$A \rightarrow \alpha B$是一个产生式,或$A \rightarrow \alpha B\beta$是一个产生式而$\beta \Rightarrow \varepsilon$ (即$\varepsilon \in first(\beta)$),则把**follow(A)**加至**follow(B)**中.

例如: 对文法(3.16),求每个非终结符的后随符号集.

解: (1)求 follow(E):E为开始符号,所以#属于follow(E);由F $\longrightarrow$ (E) , 得) $\in$ follow(E), 故follow(E)= {) , # }.

(2) 求follow(E’): 由产生式 E $\longrightarrow$ TE’,得

follow(E) $\in$ follow(E’) ,故

follow(E’)= follow(E)={) , # }

(3) 求follow(T): 由产生式E $\longrightarrow$ TE’和

E’ $\longrightarrow$ ε,得

follow(T)= first(E’) U follow(E),

即 follow(T) ={ + ,) , # };

(4) 求follow(T’): 由产生式T$\longrightarrow$FT’,得

follow(T) $\in$ follow(T’) ,故

follow(T’)= follow(T)= {) ,+, # }

(5) 求follow(F): 由产生式T $\longrightarrow$FT’和

T’ $\longrightarrow$ ε , 得

follow(F)= first(T’)\εUfollow(T),

即 follow(F) ={ * , + ,) , # };

三. 产生式**A** $\longrightarrow$α的选用集定义如下:

(1) 若α**<>**ε**,**且不存在推导α$\overset{+}{\Rightarrow}$ε**,**则产生式 **A** $\longrightarrow$ α 的选用集

select(A$\longrightarrow$α) = first(α)

(2) 若α<>ε,且存在推导$\alpha \overset{+}{\Rightarrow} \varepsilon$,则产生式 $A \longrightarrow \alpha$ 的选用集

select($A \longrightarrow \alpha$) = first(α)Ufollow(A)

(3) 若α=ε, 即产生式 $A \longrightarrow \varepsilon$ 的选用集

select($A \longrightarrow \varepsilon$) = follow(A)

例如: 求文法(3.16)各产生式的选用集.

select($E \longrightarrow TE'$)=first(TE') = { (,id}

select($E' \longrightarrow +TE'$)=first(+TE') = { + }

select($E' \longrightarrow \varepsilon$)=follow(E')={ # ,) } select(F $\rightarrow$ (E))=first((E)) = { (}

select(F $\rightarrow$ id) = {id)}

select(T $\rightarrow$ FT')=first(FT') = {(,id}

select(T' $\rightarrow$ *FT')=first(*FT') = { * }

select(T' $\rightarrow$ ε)=follow(T')={ # ,),+ }

四.LL(1)文法的判定.

设有文法G,如果其中任意产生式

A $\rightarrow$ α | β , 存在

select(A $\rightarrow$ α) n select(A $\rightarrow$ β)=Φ

则称G为LL(1)文法.

LL(1)文法是一种无二义性,非左递归的上下文无关文法,并且关于任意非终结符的不同产生式,其选用集不相交.

例如: 文法(3.16)

$$\mathrm{select}(E' \rightarrow +TE') \cap \mathrm{select}(E' \rightarrow \varepsilon) = \Phi$$

$$\mathrm{select}(T' \rightarrow *FT') \cap \mathrm{select}(T' \rightarrow \varepsilon) = \Phi$$

$$\mathrm{select}(F \rightarrow (E)) \cap \mathrm{select}(F \rightarrow id) = \Phi$$

所以,文法(3.16)是**LL(1)**文法.

例如: 验证下列文法是否为LL(1)文法.

(1) $S \longrightarrow$ if E then S S'

(2) $S \longrightarrow$ others

(3) $S' \longrightarrow$ else S

(4) $S' \longrightarrow \varepsilon$

(5) $E \longrightarrow$ expression

解: 求非终结符的首符号集和后随符号集

first(S)={if,other}　　follow(S)={#,else}

first(S')={else,ε}　　follow(S')={#,else}

first(E)={expression}　　follow(E)={ then}

求每个产生式的选用集

select(1)={if}　　select(2)={others}

select(3)={else}　select(4)={ # , else }

select(5)={expression}

S的产生式的两个选用集

$$select(1) \cap select(2) = \Phi$$

S'的产生式的两个选用集

$$select(3) \cap select(4) = \{else\}$$

由于文法同一终结符的两个产生式的选用集相交,所以,该文法不是**LL(1)**的.

LL(1)语言

　　已经可以判断一个文法是否是**LL(1)**的,但有两个问题需解决.

(1)是否每个语言都有一个**LL(1)**文法?

　　　答案是否定的.

例如: 语言**L(G)**

　　$L(G)=\{aa^t \mid a \in \{0,1\}^* \text{ 中}\}$. ($a^t$表示a的转置). 由文法G生成.

$S \longrightarrow 0S0$

$S \longrightarrow 1S1$　　　不是**LL(1)**的.

$S \longrightarrow \varepsilon$

(2)能否找一个算法判断一个语言是否是LL(1)的?

答案是否定的.

上述问题的本质,在于它说明不是所有的程序设计语言都对应一个LL(1)文法;但是,程序设计语言的决大部分上下文无关性质可被LL(1)文法描述.

问题的关键: 对于给定的非LL(1)文法,如何找出一个等价的LL(1)文法.从上述结论知:实现此事无通用的算法.但在大多情况下,

文法的转换并不困难.有经验的编译工作者可做手工转换.

文法转换涉及的问题:

1).消除左递归:

消除直接左递归

消除循环左递归

2).提取公共左因子:

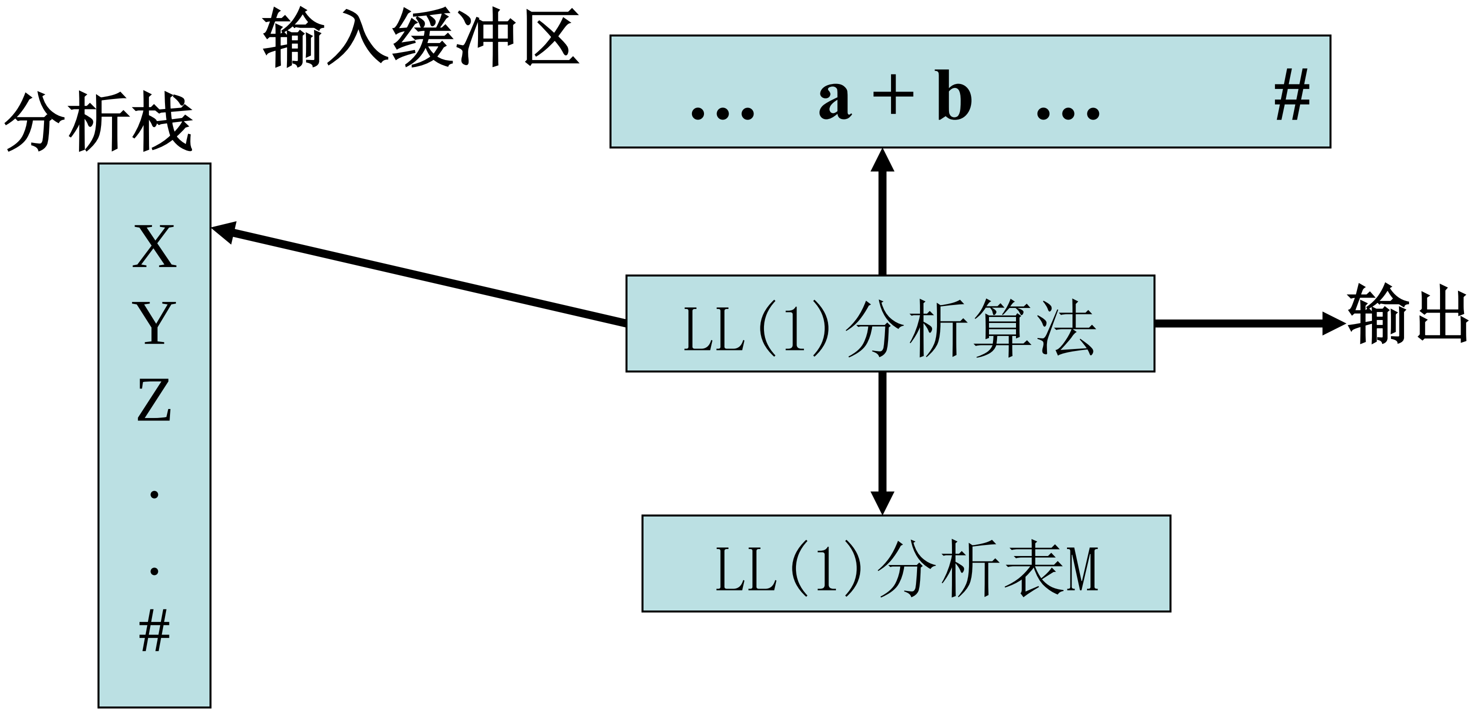
但上述方法没有普遍性.否则,于判断一个语言是否为LL(1)的问题相矛盾.

第3章: 程序语言的语法描述与分析

- ⌘3.1 上下文无关文法
- ⌘3.2 自顶向下分析法
- ⌘3.3 递归下降分析法
- ⌘3.4 LL(1)分析法
- ⌘3.5 LL(1)分析器

3.5 LL(1)分析器

LL(1)分析器的模型.



在LL(1)分析模型中.

输入缓冲区:存放要分析的符号串,在串的末尾加符号#表示输入结束.

分析栈:存放当前句型中尚待分析的文法符号,其中包括终结符号和非终结符号.栈底用#标记结束.初始化时,把#和文法的开始符号放在栈顶.

分析表M:是一个矩阵.其每行对应文法的一个非终结符号A,其每一列对应文法的一个终结符号a,结束标记符#作为一个终结符号

矩阵M[A,a]表示当前栈顶为A输入符号为a时, 应该选用的产生式, 或者为空. 构造分析表时, 按照文法的选用集, 若有a∈select(A→α),则产生式A→α填入M[A, a]；最后，把某些空白项作为处理错误的入口。

	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)		

LL(1)分析算法控制分析器的执行。初态，分析栈顶为文法的开始符号，输入缓冲指向输入串的第一个符号。对当前的栈顶符号X，输入符号a，算法执行的操作如下：

(1)如果X=a=#，算法结束 并且报告分析成功,接受输入串.

(2)如果X=a<>#，则从栈中弹出X,并使缓冲区指针前进到下一个输入符号.

(3)如果**X**是一个非终结符**,**则查分析表**;**若**M[A,a]**为**X**的产生式**,**用该产生式右部的逆替换栈顶符号**,**否则出错**.**

<u>算法**: LL(1)**分析算法</u>

输入**:**串**w,**文法**G**的分析表**M.**

输出**:**如果**w**∈**L(G),**则产生**w**的最左推导**,**否则输出错误信息**.**

方法**:** 初态时**,**分析栈为**#S,**栈顶**S**是文法的开始符号**;**缓冲区为**w#.**分析器按下列操作进行语法分析**:**

```
(1) push # ,S;指针ip指向串w#的第一个符号;
(2)repeat
(3)       令X为栈顶符号,a为ip所指的符号;
(4)        if  X 为终结符或#   then
(5)                if    X=a=#  then  接受w
(6)                else  if  X =a<>#  then 弹出X, 并使ip前进
(7)                        else    error
(8)        else  /* X为非终结符 */
(9)                if   M[X,a]=X   Y1Y2…Yk then
                     begin                        →
(10)                    弹出X;
```

(11)　　　push Yk,…,Y2,Y1　/* Y1在栈顶 */

(12)　　end

(13)　　else　error

(14)　until Ｘ=#;　/* 栈为空 */

例如:　按照前面的文法所示的LL(1)分析表,
对输入串id+id*id进行语法分析.

解:　分析器分析过程中,栈中的符号,剩余的
输入串和选用的产生式,如下表所列:

分析栈(顶)	输入串	产生式
#E	id+id*id#	E → TE’
#E’T	id+id*id#	T → FT’
#E’T’F	id+id*id#	F → id
#E’T’id	id+id*id#	
#E’T’	+id*id#	T’ → ε
#E’	+id*id#	E’ → +TE’
#E’T+	+id*id#	
#E’T	id*id#	T → FT’
#E’T’F	id*id#	F → id
#E’T’id	id*id#	

分析栈(顶)	输入串	产生式
#E'T'	*id #	T' → *FT'
#E'T'F*	*id #	
#E'T'F	id #	F → id
#E'T'id	id #	
#E'T'	#	T' → ε
#E'	#	E' → ε
#	#	

LL(1)分析表的优点**:**

1)是确定的**,**永远不需要回溯**.**

2)分析表的体积比其他方法小**.**

3)LL(1)分析表使用广泛**.**

可以把**LL(1)** 文法推广到**LL(K)**文法**,**即**:**超前搜索**K**个符号**,**来决定采用哪个候选式**.**一般地**,**很少使用**.**

LL(1)分析中的错误处理

- 发现错误
  - 栈顶的终结符与当前输入符不匹配
  - 非终结符A于栈顶，面临的输入符为a，但分析表M的M[A,a]为空
- 应急恢复策略
  - 跳过输入串中的一些符号直至遇到“同步符号”为止。

LL(1)分析中的错误处理

- 同步符号集的选择
  - 把FOLLOW(A)中的所有符号作为A的同步符号。跳过输入串中的一些符号直至遇到这些“同步符号”，把A从栈中弹出，可使分析继续
  - 把FIRST(A)中的符号加到A的同步符号集，当FIRST(A)中的符号在输入中出现时，可根据A恢复分析应急恢复策略

LL(1)分析中的错误处理

文法G[E]:

(1) E → TE’

(2) E’ → +TE’

(3) E’ → ε

(4) T → FT’

(5) T’ → *FT’

(6) T’ → ε

(7) F → (E)

(8) F → i

	i	+	*	(	)	#
E	(1)			(1)	syn	syn
E’		(2)			(3)	(3)
T	(4)	syn		(4)	syn	syn
T’		(6)	(5)		(6)	(6)
F	(8)	syn	syn	(7)	syn	syn