



重庆大学
CHONGQING UNIVERSITY

第四讲 语法分析

重庆大学 计算机学院 张敏

2024年3月23日



pls program a compiler for C to RISC



I'm sorry, but programming a complete C-to-RISC compiler is not a task that can be accomplished through a single response on this platform. Developing a compiler is a complex task that requires significant expertise in compiler design, computer architecture, and programming languages.

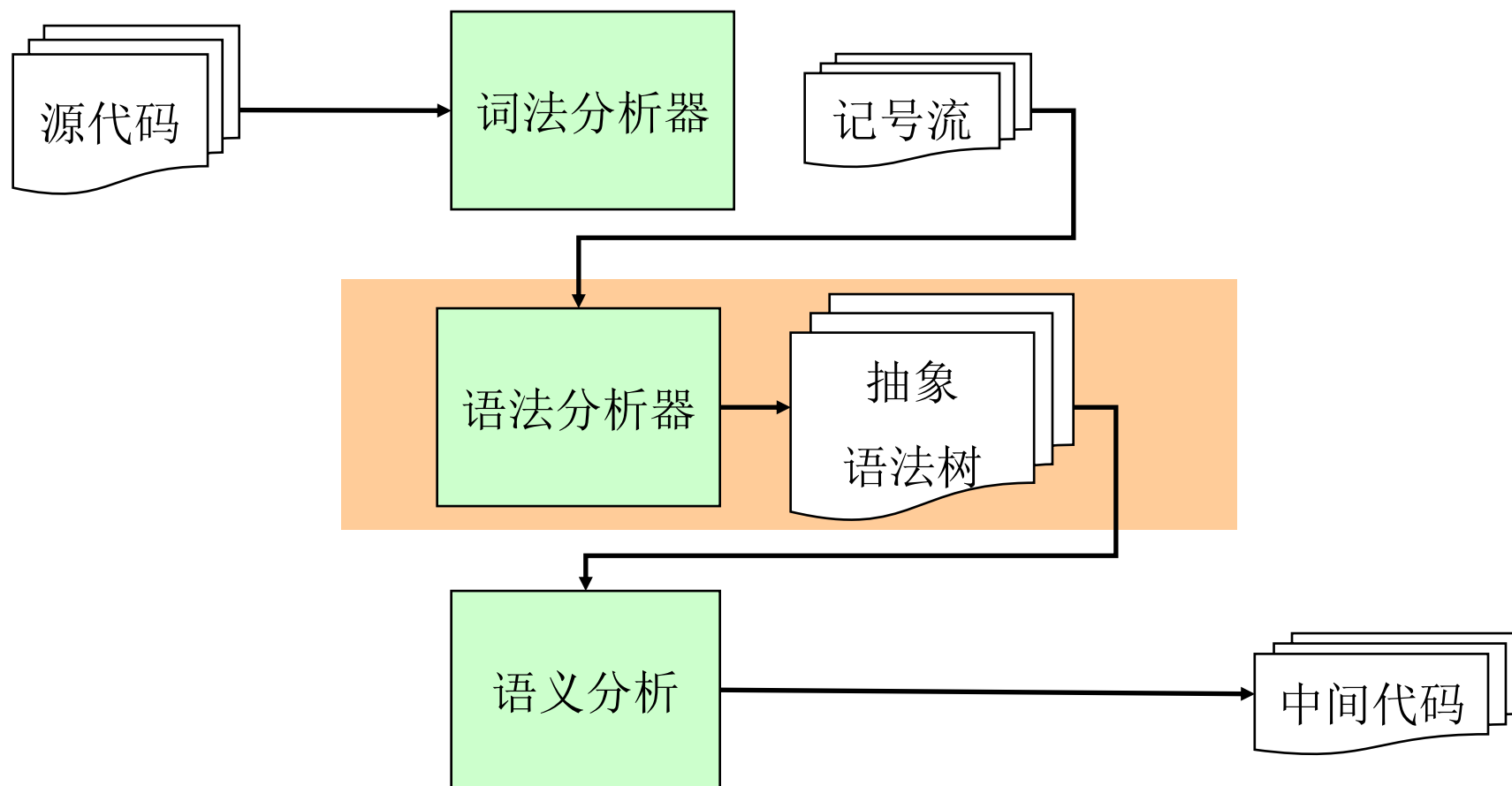


However, I can suggest some resources that may help you get started with the process:

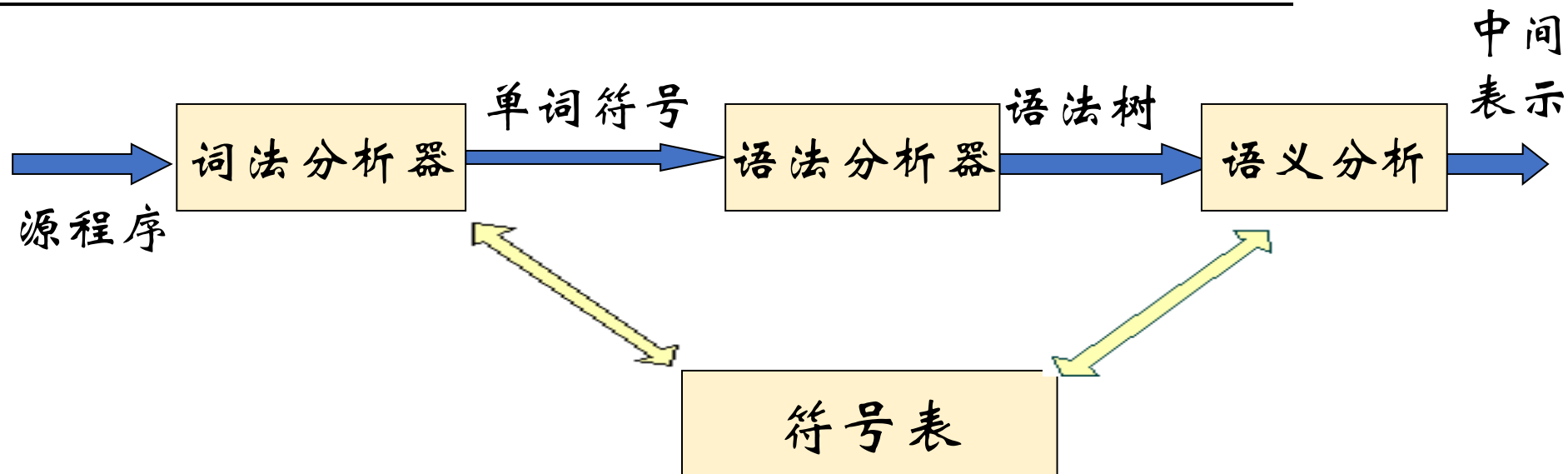
1. Start by learning about the RISC architecture and instruction set that you want to target. This will help you understand the limitations and capabilities of the platform and optimize your compiler accordingly.
2. Learn about the C language and its syntax, semantics, and structure. This will help you understand how to parse and analyze C code and generate corresponding RISC instructions.
3. Study compiler design and implementation techniques, such as lexical analysis, syntax analysis, code generation, and optimization. These techniques are fundamental to building a functional and efficient compiler.
4. Consider using an existing compiler framework or toolchain, such as LLVM or GCC, to simplify the process of building a C-to-RISC compiler. These tools provide pre-built libraries and modules that can be integrated into your compiler project.
5. Test and debug your compiler. Regenerate response hat it generates correct and efficient code. Use test suites and benchmarking tools to measure performance.

Can ChatGPT

4.1 语法分析器的任务



4.1 语法分析器的任务



- ①对词法分析器产生的单词符号进行处理，输出分析树
- ②与单词相关的信息记录到符号表中
- ③类型检查
- ④错误处理

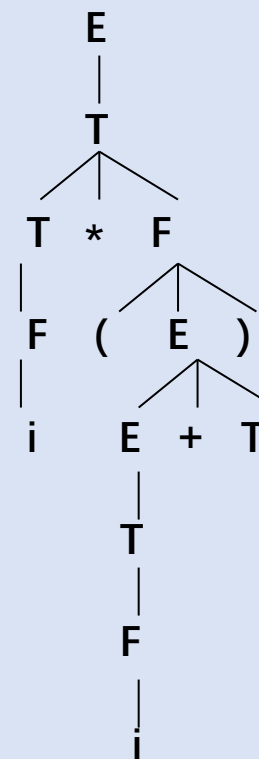
4.1.1 语法树构建

$G[E]: E \rightarrow E + T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow (E) \mid i$

句型: $i * (i + T)$

推导过程:

$E \Rightarrow T \Rightarrow T * F \Rightarrow T * (E) \Rightarrow F * (E) \Rightarrow i * (E)$
 $\Rightarrow i * (E + T) \Rightarrow i * (T + T) \Rightarrow i * (F + T) \Rightarrow i * (i + T)$





推导与语法分析树

语法分析树描绘了如何从文法的开始符号推导出一个句子的过程，与推导所用的顺序无关（最左、最右、其他）。

特点:

- 树中的每个内部节点代表非终结符
- 每个叶子节点代表终结符
- 每一步推导代表如何从双亲节点生成直接孩子节点
- 分析树的含义取决于树的后序遍历



4.1.2 二义性文法

给定文法 G ，如果存在句子 s ，它有两棵不同的分析树，那么称 G 是二义性文法。

从编译器角度，二义性文法存在以下问题：

- 同一个程序会有不同的含义
- 因此程序运行的结果不是唯一的

解决方案：文法的重写



4.1.2 二义性文法

$E \rightarrow \text{num}$
 $\quad | \text{id}$
 $\quad | E + E$
 $\quad | E * E$



$E \rightarrow E + T$
 $\quad | T$
 $T \rightarrow T * F$
 $\quad | F$
 $F \rightarrow \text{num}$
 $\quad | \text{id}$

Eg $1+2*3$ (运算的优先级)
 $1+2+3$ (加法的左结合性)



4.1.3 句型的分析

句型分析：识别一个符号串是否为某文法的句型，是某个推导的构造过程。

在语言的编译实现中，把完成句型分析的程序称为分析程序或识别程序 (Parser) 。

从左到右的分析算法，即总是从左到右地识别输入符号串，首先识别符号串中的最左符号，进而依次识别右边的一个符号。



4.1.4 语法错误处理

错误处理目标

- 清楚而准确地报告发现的错误，如错误的位置和性质。
- 迅速地从错误中恢复过来。
- 不应该明显地影响编译程序对正确程序的处理效率。

错误恢复策略

- 紧急恢复：简单，适用于大多数分析程序。
做法：一旦发现错误，分析程序每次抛弃一个输入记号，直到扫描到的记号属于某个指定的同步记号集合为止。同步记号通常是定界符，如语句结束符分号、块结束标识END等。
- 短语级恢复
- 出错产生式
- 全局纠正



4.2 语法分析方法

数学理论：上下文无关文法（CFG）：描述语言语法规则的数学工具

- 自顶向下分析

- 递归下降分析算法（预测分析算法）
- LL分析算法

- 自底向上分析

- LR 分析算法
- 算符优先分析法



1、上下文无关文法能描述程序设计语言的哪些部分？

- 单词
- 语句
-
- 表达式中嵌套的括号对、 if 语句

2、正规文法无法描述语言 $L=\{a^n b^n | n \geq 1\}$ ，上下文无关文法能否描述此类语言？请写出对应的文法。

$$G = (\{S\}, \{a, b\}, \{S \rightarrow aSb, S \rightarrow ab\}, S)$$



4.2.1 语法分析工具—上下文无关文法

如果在上下文无关文法 G 中存在一个非终结符 A ，并有推导：

$A \Rightarrow a_1 A a_2$ ，则称该文法具有自嵌套特性。

例： $G = (\{S, A\}, \{\text{begin}, \text{end}, [,]\}, P, S)$

$P:$ $S \rightarrow \text{begin } A \text{ end}$

$S \rightarrow \varepsilon$

$A \rightarrow [S]$

自嵌套特性区别了正规文法和上下文无关文法，其识别系统是不确定的下推自动机。

4.2.1 语法分析工具—下推自动机 (PDA)

PDA可以正式描述为7元组 $(Q, \Sigma, S, \delta, q_0, I, F)$

Q : 有限数量的状态

Σ : 输入字母

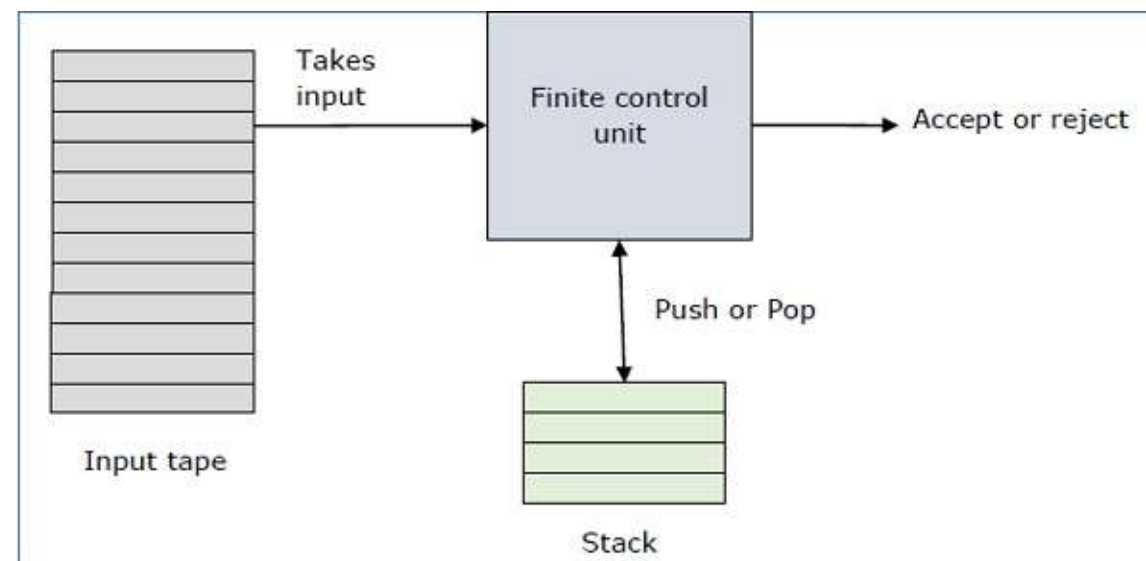
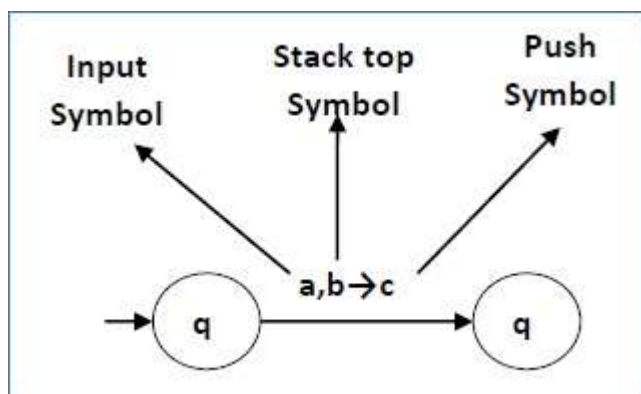
S : 堆栈符号

δ : 转移函数

q_0 : 初始状态 ($q_0 \in Q$)

I : 初始堆栈顶符号 ($I \in S$)

F : 接受状态 ($F \in Q$)





4.2.2 语法分析算法

自顶向下分析法:

从文法的开始符号出发, 反复使用各种产生式, 寻找与输入符号匹配的推导。

自底向上分析法:

输入符号串开始, 逐步进行归约, 直至归约到文法的开始符号。

两种方法反映了两种不同的语法树的构造过程



4.3 自顶向下分析法

语法分析：给定文法 G 和句子 s ，回答 s 是否能够从 G 推导出来？

基本算法思想：

从 G 的开始符号出发，随意推导出某个句子 t ，比较 t 和 s ：

- 若 $t=s$ ，则回答 “是”

- 若 $t \neq s$ ，则？

因为这是从开始符号出发推出句子，因此称为 自顶向下分析

- 对应于分析树自顶向下的构造顺序



4.3.1 自顶向下分析法示例

试分析输入串 $\omega = abbcde$ 是否为如下文法的一个句子

$$S \rightarrow aAcBe$$
$$A \rightarrow b \mid Ab$$
$$B \rightarrow d$$



4.3.1 自顶向下分析法示例

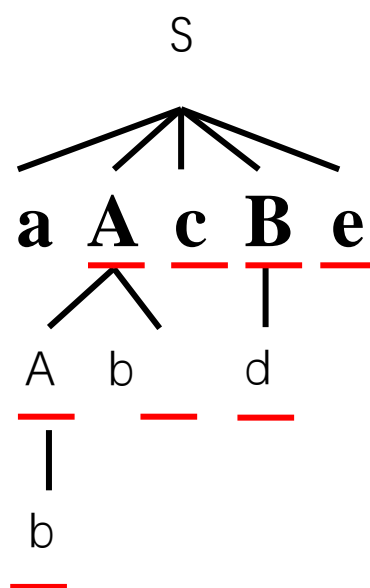
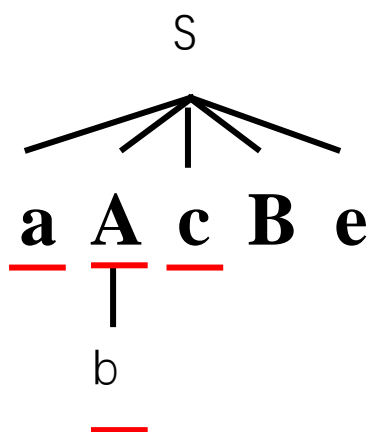
a b b c d e \$
↑

a b b c d e \$
↑

$S \rightarrow aAcBe$

$A \rightarrow b \mid Ab$

$B \rightarrow d$



$S \Rightarrow aAcBe \Rightarrow aAbcBe \Rightarrow abbcBe \Rightarrow abbcde$

为输入符号串建立一个最左推导序列的过程



4.3.1 自顶向下算法描述

```
tokens[]; // all tokens
i=0;
stack = [S] // S是开始符号
while (stack != [ ])
    if (stack[top] is a terminal t)
        if (t==tokens[i++])
            pop( );
        else backtrack( );
    else if (stack[top] is a nonterminal T)
        pop( );
        push (the next right hand side of T)
```



4.3.3 自顶向下算法讨论

算法需要用到回溯

给分析效率带来问题

编译器必须高效

编译上千万行的内核等程序

需要线性时间的算法

避免回溯



4.3.3.1 确定的自顶向下分析思想

从某文法的开始符号出发，对给定的输入符号串如何根据当前输入符号(前看符号)唯一确定选用哪个产生式替换相应非终结符往下推导或构造一棵相应的语法树。

核心问题：选择使用哪个产生式进行推导？

假定要被替换的最左非终结符号是 V ，且左部为 V 的规则有 n 条：
 $V \rightarrow A_1 | A_2 | \dots | A_n$ ，那么如何确定用哪个右部去替换 V ？

确定的自顶向下分析示例1

若有文法G1[S]:

$S \rightarrow pA \mid qB$

$A \rightarrow cAd \mid a$

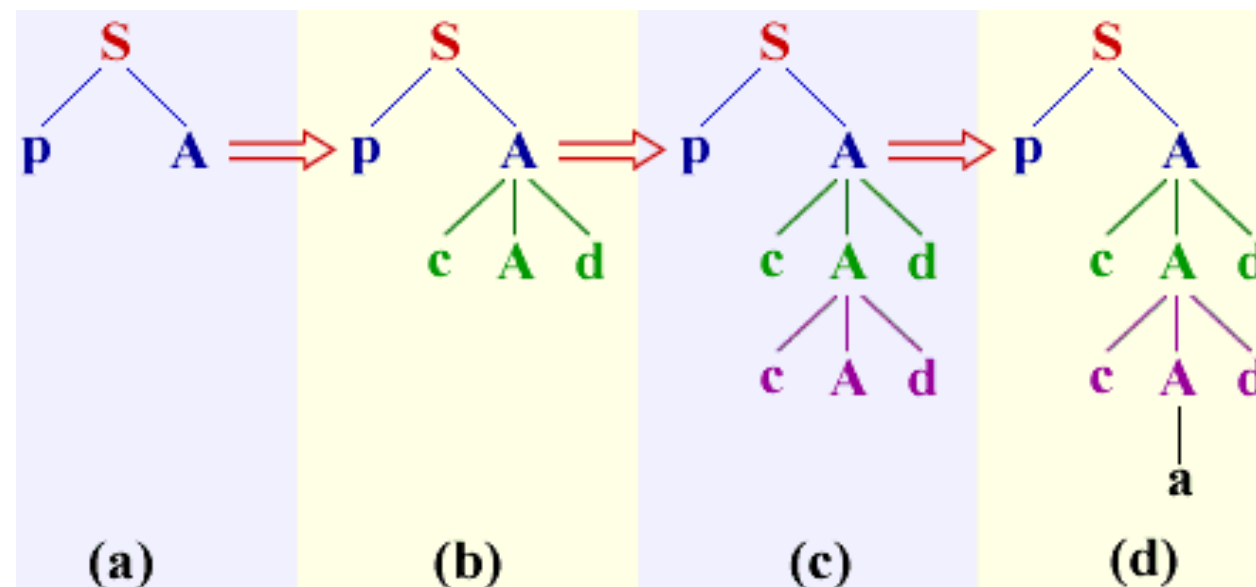
$B \rightarrow dB \mid c$

识别输入串 $w = pccadd$ 是否是G1[S]的句子

试探推导过程:

$S \Rightarrow pA \Rightarrow pcAd \Rightarrow pccAdd \Rightarrow pccadd$

试探成功。



这个文法有以下两个特点:

- ① 每个产生式的右部都由终结符号开始。
- ② 如果两个产生式有相同的左部, 那么它们的右部由不同的终结符开始。

确定的自顶向下分析示例2

若有文法G2[S]:

$S \rightarrow Ap \mid Bq$

$A \rightarrow a \mid cA$

$B \rightarrow b \mid dB$

识别输入串w=ccap是否是G2[S]的句子

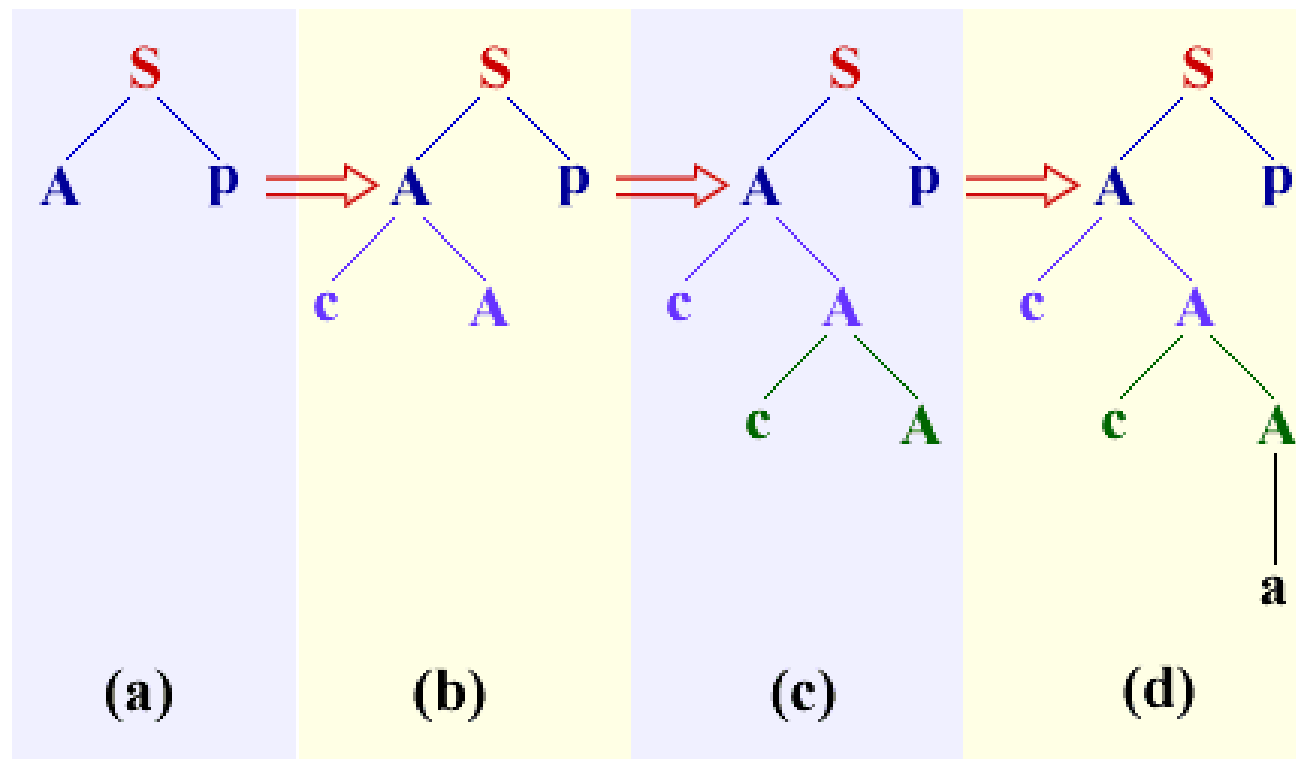
那么试探推出输入串的推导过程为：

$S \Rightarrow Ap \Rightarrow cAp \Rightarrow ccAp \Rightarrow ccap$

试探推导成功。

文法的特点是：

- ① 产生式的右部不全是由终结符开始。
- ② 如果两个产生式有相同的左部，它们的右部是由不同的终结符或非终结符开始。
- ③ 文法中空产生式。



确定的自顶向下分析示例3

若有文法G3[S]:

$S \rightarrow aA|d$

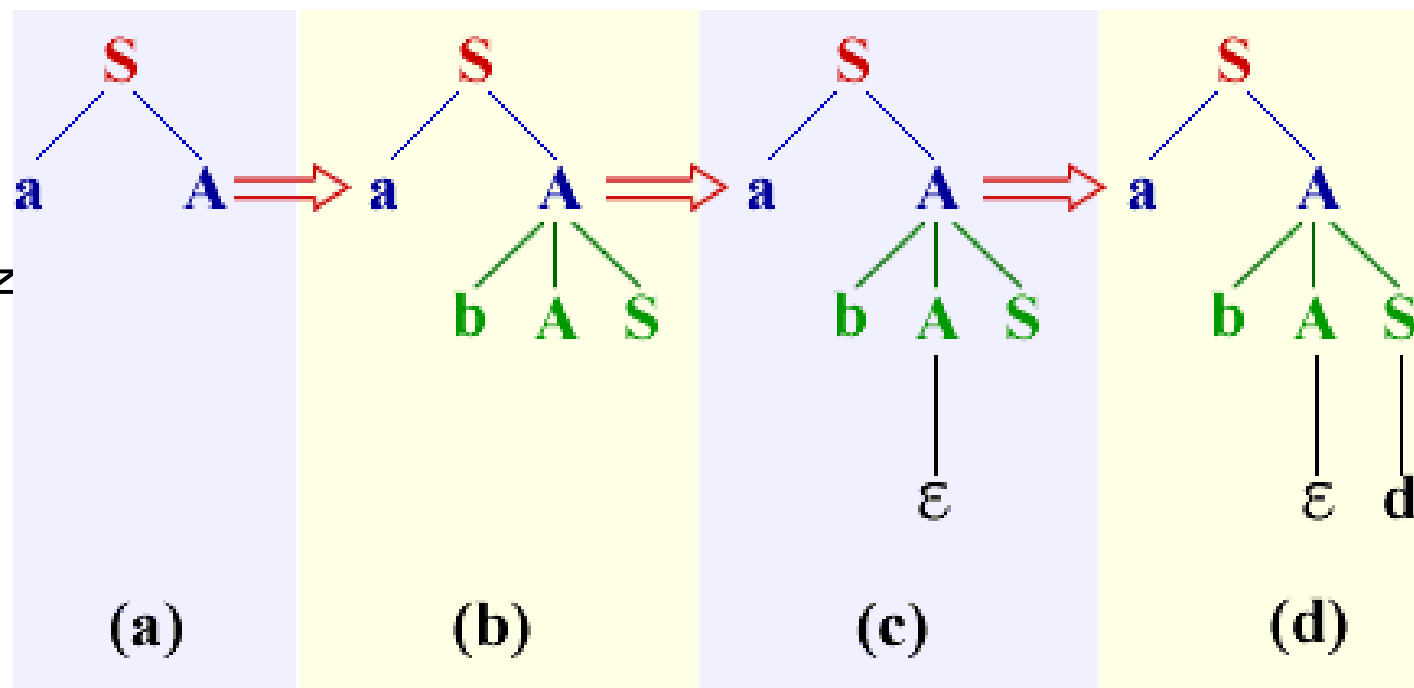
$A \rightarrow bAS|\varepsilon$

识别输入串 $w=abd$ 是否是G3[S]的句子

试探推导出abd的推导过程为:

$S \Rightarrow aA \Rightarrow abAS \Rightarrow abS \Rightarrow abd$

试探推导成功。



文法的特点是:

文法中含有空产生式。

由此可以看出, 当某一非终结符的产生式中含有空产生式时, 它的非空产生式右部的**首符号集**两两不相交, 并与在推导过程中**紧跟该非终结符后边可能出现的终结符集**也不相交, 则仍可构造确定的自顶向下分析。



4.4消除文法的不确定因素

哪些原因导致自顶向下分析的不确定性?
不确定性会导致的后果?

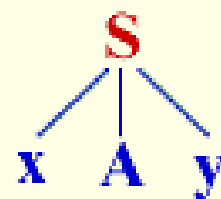
具有不确定性的文法示例1

若有文法 $G_4[S]$:

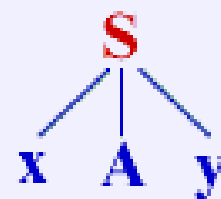
$S \rightarrow xAy$

$A \rightarrow ab|a$

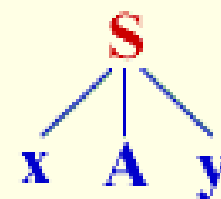
若当前输入串为 xay , 则可能的推导树为:



(a)



(b)



(c)

出现回溯的原因:

若文法中含有形如: $A \rightarrow \alpha\beta|\alpha\gamma$ 的产生式, 这导致了对相同左部的产生式其候选式的右部第一符号相同. 当不清楚如何选择时, 我们可以[通过改写产生式来推后这个决定](#)。



4.4.1 消除回溯提取左公因子

提“左公因子”

现将产生式 $A \rightarrow \alpha\beta | \alpha\gamma$ 进行等价变换为：

$$A \rightarrow \alpha (\beta | \gamma)$$

其中 '(..|..|..)' 为元符号，可进一步引进新非终结符 A' ，去掉 '(..|..|..)' 使产生式变换为：

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta | \gamma$$

写成一般形式为：

$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \dots | \alpha\beta_n$ ，提取左公共因子后变为：

$A \rightarrow \alpha (\beta_1 | \beta_2 | \dots | \beta_n)$ ，再引进非终结符 A' ，变为：

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$$

若在 β_i 、 β_j 、 $\beta_k \dots$ (其中 $1 \leq i, j, k \leq n$) 中仍含有左公共因子，这时可再次提取，这样反复进行提取直到引进新非终结符的有关产生式再无左公共因子为止。



提取左公因子示例

若文法G10的产生式为:

$$(1) S \rightarrow aSb \quad (2) S \rightarrow aS \quad (3) S \rightarrow \varepsilon$$

请提取文法中的左公因子。

对产生式(1)、(2)提取左公因子后得:

$$S \rightarrow aS(b|\varepsilon) \quad S \rightarrow \varepsilon$$

进一步变换为文法G' 10:

$$S \rightarrow aSA$$

$$A \rightarrow b$$

$$A \rightarrow \varepsilon$$

$$S \rightarrow \varepsilon$$



间接左公因子示例

若文法G11的产生式为：

(1) $A \rightarrow ad$

(2) $A \rightarrow Bc$

(3) $B \rightarrow aA$

(4) $B \rightarrow bB$

请问该文法有何特点？该如何处理？

产生式(2)的右部以非终结符开始，因此左公共因子可能是隐式的，所以这种情况下对右部以非终结符开始的产生式，用其相同左部而右部以终结符开始的产生式进行相应替换，对文法G11分别用(3)、(4)的右部替换(2)中的B

可得：

(1) $A \rightarrow ad$

(2) $A \rightarrow aAc$

(3) $A \rightarrow bBc$

(4) $B \rightarrow aA$

(5) $B \rightarrow bB$



间接左公因子示例

提取产生式(1)、(2)的左公共因子得：

$$A \rightarrow a(d|Ac)$$

$$A \rightarrow bBc$$

$$B \rightarrow aA$$

$$B \rightarrow bB$$

• 引进新非终结符 A' 得 G' 11为：

$$(1) A \rightarrow aA'$$

$$(2) A \rightarrow bBc$$

$$(3) A' \rightarrow d$$

$$(4) A' \rightarrow Ac$$

$$(5) B \rightarrow aA$$

$$(6) B \rightarrow bB$$



提取左公因子后化简

值得注意的是对文法进行提取左公共因子变换后，有时会使某些产生式变成无用产生式，在这种情况下必须对文法重新压缩(或化简)。

- 示例：若有文法G12的产生式为：

$$(1) S \rightarrow aSd \quad (2) S \rightarrow Ac \quad (3) A \rightarrow aS \quad (4) A \rightarrow b$$

请消除该文法中的左公因子

引入新非终结符 A' 后变为：

$$(1) S \rightarrow aSA'$$

$$(2) S \rightarrow bc$$

$$(3) A' \rightarrow d \mid c$$

$$(4) A \rightarrow aS$$

$$(5) A \rightarrow b$$

显然，原文法G12中非终结符A变成不可到达的符号，产生式(4)、(5)也就变为无用产生式，所以应删除。



课堂讨论



重庆大学
CHONGQING UNIVERSITY

不一定每个文法的左公共因子都能在有限的步骤内替换成无左公共因子的文法。

请思考：文法中产生式的哪些形式会导致此种情况？



具有不确定性的文法示例2

若有文法G6[S]:

(1) $S \rightarrow Sa$

(2) $S \rightarrow b$

请判断句子baa是否为文法所描述语言中的句子

具有不确定性的文法示例2

若有文法G6[S]:

(1) $S \rightarrow Sa$

(2) $S \rightarrow b$

请判断句子baa是否
为文法所描述语言中的句子

S
|
a

(a)

S
/ \
S a

(b)

S
/ \
S a
|
b

(c)

S
/ \
S a
/ \
S a

(d)

S
/ \
S a
/ \
S a
|
b

(e)

含有左递归的文法不能用确定的自顶向下分析，但可用带回溯的自顶向下分析。

左递归文法

直接左递归

设一个文法含有下列形式的产生式: 1) $A \rightarrow A\beta$ $A \in VN, \beta \in V^*$
称含1)中产生式的文法为含有左递归的规则或称直接左递归文法。

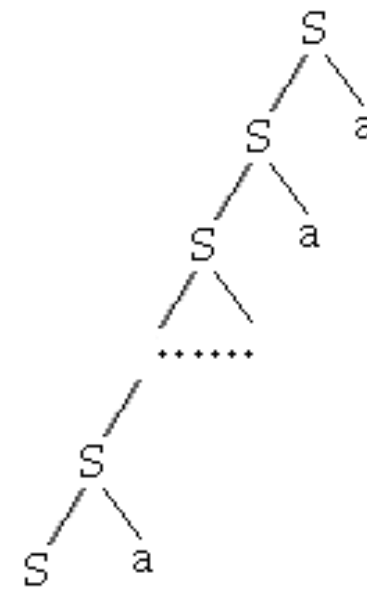
间接左递归文法

设一个文法含有下列形式的产生式。

2) $A \rightarrow B\beta$

$$B \rightarrow A\alpha \quad A, B \in VN, \alpha, \beta \in V^*$$

可见含2)中产生式的文法有 $A \Rightarrow \dots \Rightarrow A\dots$,则称文法中含有左递归或间接左递归

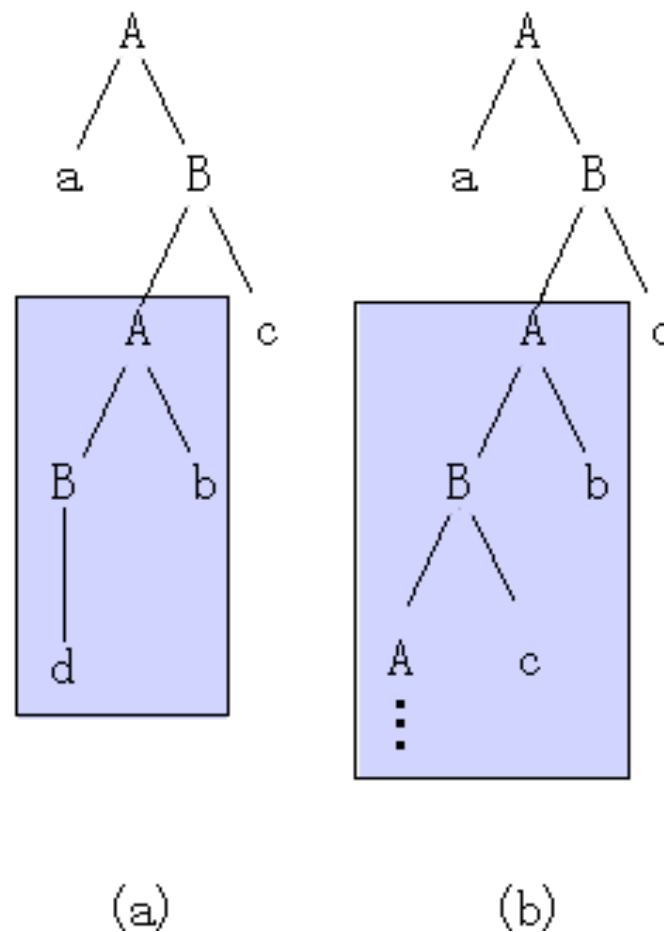


间接左递归文法示例

文法G7为:

- (1) $A \rightarrow aB$
- (2) $A \rightarrow Bb$
- (3) $B \rightarrow Ac$
- (4) $B \rightarrow d$

若有输入串为adbcbcbbc, 请给出分析过程。





4.4.2 消除左递归

消除直接左递归：把直接左递归改写为右递归。

改写方法：

首先将含左递归的产生式进行分组如下：

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

其中 β 都不以 A 开头，将 A 产生式替换为：

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

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

示例：

$G[S]: S \rightarrow Sa \quad S \rightarrow b$ 可改写为 $G[S]: S \rightarrow bS' \quad S' \rightarrow aS' \mid \varepsilon$

改写后文法和原文法产生的语言句子集都为： $\{ba^n \mid n \geq 0\}$ 。

文法中左递归的消除

消除间接左递归

- 1) 对于间接左递归的消除需先将间接左递归变为直接左递归，然后再消除直接左递归。
- 2) 改写文法

一个文法是左递归的，如果它有非终结符号A，对某个文法符号串 α ，存在推导： $A \xRightarrow{+} A\alpha$ 若存在某个 $\alpha=\varepsilon$ ，则称该文法是有环路的。



文法中左递归的消除

消除文法中左递归算法:

要求文法中不含环路

满足这个要求的充分条件是, 文法中不包含形如 $A \rightarrow A$ 的和 $A \rightarrow \varepsilon$ 的产生式

输入: 无环路、无 ε -产生式的文法G

输出: 不带有左递归的、与G等价的文法G'

方法:

(1) 把文法G的所有非终结符号按某种顺序排列成 A_1, A_2, \dots, A_n

(2) for (i=1; i ≤ n; i++)

 for (j=1; j ≤ i-1; j++)

 if ($A_j \rightarrow \delta_1 | \delta_2 | \dots | \delta_k$ 是关于当前 A_j 的所有产生式) {

 把每个形如 $A_i \rightarrow A_j \gamma$ 的产生式改写为: $A_i \rightarrow \delta_1 \gamma | \delta_2 \gamma | \dots | \delta_k \gamma$;

 消除关于 A_i 的产生式中的直接左递归;

 }

(3) 化简第(2)步得到的文法, 即去除无用的非终结符号和产生式。



文法中左递归的消除

文法G的产生式为：

(1) $S \rightarrow Qc | c$

(2) $Q \rightarrow Rb | b$

(3) $R \rightarrow Sa | a$

请消除该文法可能出现的左递归。



文法中左递归的消除

若非终结符排序为S、Q、R:

则左部为S的产生式(1)无直接左递归, (2)中右部不含S, 所以把(1)右部代入(3)得:

$$(4) R \rightarrow Qca|ca|a$$

再将(2)的右部代入(4)得:

$$(5) R \rightarrow Rbca|bca|ca|a$$

对(5)消除直接左递归得:

$$R \rightarrow (bca|ca|a)R'$$

$$R' \rightarrow bcaR'|\epsilon$$

最终文法变为:

$$S \rightarrow Qc|c$$

$$Q \rightarrow Rb|b$$

$$R \rightarrow (bca|ca|a)R'$$

$$R' \rightarrow bcaR'|\epsilon$$



文法中左递归的消除

若非终结符的排序为R、Q、S:

则把(3)代入(2)得: $Q \rightarrow Sab|ab|b$

再将此代入(1)得: $S \rightarrow Sabc|abc|bc|c$

消除该产生式的左递归后, 最终文法变为:

$S \rightarrow (abc|bc|c)S'$

$S' \rightarrow abcS'|\epsilon$

$Q \rightarrow Rb|b$

$R \rightarrow Sa|a$

此时Q、R为不可到达的非终结符, 所以以Q、R为左部及包含Q、R的产生式应删除。

当非终结符的排序不同时, 最后结果的产生式形式不同, 但它们是等价的。



具有不确定性的文法示例3

若有文法 $G_4[S]$:

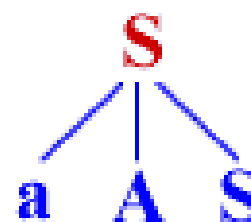
(1) $S \rightarrow aAS$

(2) $S \rightarrow b$

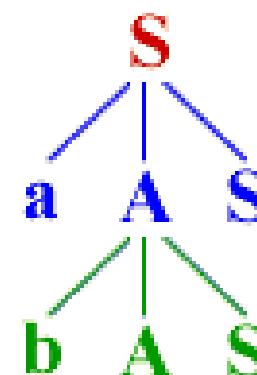
(3) $A \rightarrow bAS$

(4) $A \rightarrow \varepsilon$

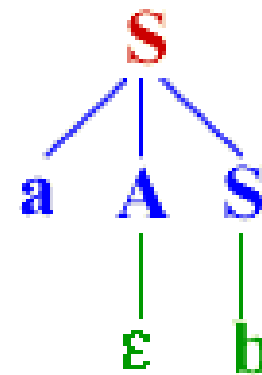
请判断输入串 ab 是否为文法描述语言的句子



(a)



(b)



(c)



4.4.3 改造 ε -无关文法的方法

若产生式 $A \rightarrow X_1 X_2 \dots X_n \in \varphi$ 则把产生式 $A \rightarrow \alpha_1 \alpha_2 \dots \alpha_n$ 加入 φ'

其中: X_i 、 α_i 为文法符号, 即 X_i 、 $\alpha_i \in (V_T \cup V_N)$

若 X_i 不能产生 ε , 则 $\alpha_i = X_i$

若 X_i 能产生 ε , 则 $\alpha_i = X_i$ 或 $\alpha_i = \varepsilon$

注意: 不能所有的 α_i 都取 ε

文法 G' 满足:

一个文法是 ε -无关的, 如果它没有 ε -产生式 (即形如 $A \rightarrow \varepsilon$ 的产生式), 或者只有一个 ε -产生式, 即 $S \rightarrow \varepsilon$, 并且文法的开始符号 S 不出现在任何产生式的右部。



示例

消除下面文法中的 ε 产生式

$$S \rightarrow Aa|b \quad A \rightarrow Ac|Sd|\varepsilon$$

A可以为 ε ，因此改写产生式得到：

$$S \rightarrow Aa|a|b$$

$$A \rightarrow Ac|c|Sd$$



4.5 自顶向下语法分析器的构造

- 递归下降分析法
- LL(1)分析法



4.5.1 递归下降分析

从文法的开始符号出发，进行推导，试图推出要分析的输入串的过程。本质上是一种试探过程，反复使用不同产生式谋求匹配输入串的过程。

算法基本思想：

- 每个非终结符构造一个分析函数
- 用前看符号指导产生式规则的选择

特点

- 容易实现（方便手工编码）
- 错误定位和诊断信息准确



4.5.1 递归下降分析

```
parse_S()  
  parse_N()  
  parse_V()  
  parse_N()
```

```
parse_N()  
  token = tokens[i++]  
  if (token==s||token==t||  
      token==g||token==w)  
    return;  
  error( "..." );
```

```
parse_V() token = tokens[i++] ...// leave this part to you
```

$S \rightarrow N V N$

$N \rightarrow s$

| t

| g

| w

$V \rightarrow e \mid d$



4.5.2 递归调用预测分析

一种确定的、不带回溯的递归下降分析方法

- 一、如何克服回溯?
- 二、对文法的要求
- 三、预测分析程序的构造



4.5.2 递归调用预测分析

如何克服回溯?

能够根据所面临的输入符号准确地指派一个候选式去执行任务。
该选择的工作结果是确信无疑的。

即:

$$A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_i | \dots | \alpha_n$$

当前输入符号: a

指派 α_i 去匹配输入符号串



4.5.2 递归调用预测分析

对文法的要求?

1. 不含左递归 $A \Rightarrow \dots \Rightarrow A\alpha$

2. $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \phi \quad (i \neq j)$

$$A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$$

非终结符号A的所有候选式的**开头终结符号集**两两互不相交

缺席匹配：当递归下降分析程序没有适当候选式时，可以用一个 ϵ -候选式，表示其它候选式可以缺席。



4.5.2 递归调用预测分析语法分析

示例:

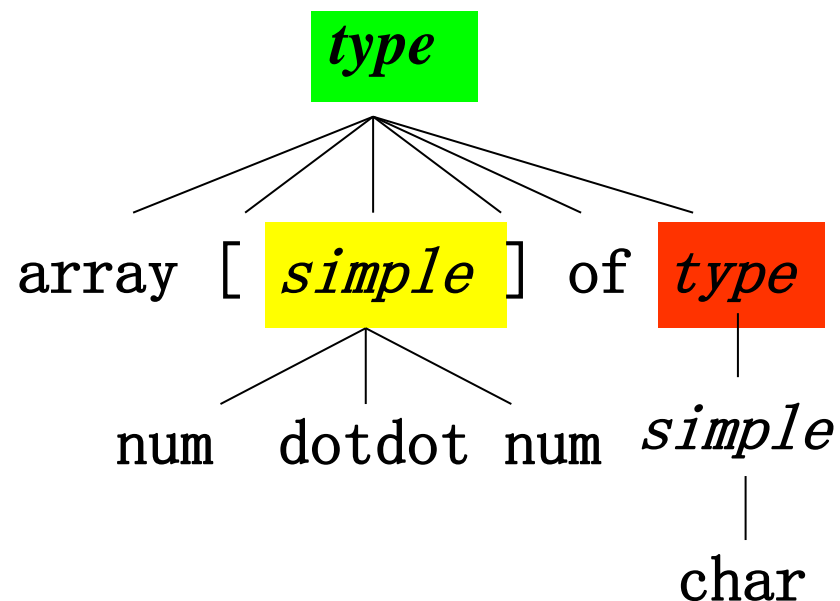
输入: array [num dotdot num] of char

有如下产生PASCAL类型子集的文法:

$type \rightarrow simple \mid \uparrow id \mid array[simple] \text{ of } type$

$simple \rightarrow integer \mid char \mid num \text{ dotdot } num$

树:



$type$ 的三个候选式, 有:

$FIRST(simple) = \{ integer, char, num \}$

$FIRST(\uparrow id) = \{ \uparrow \}$

$FIRST(array[simple] \text{ of } type) = \{ array \}$

$simple$ 的三个候选式, 有:

$FIRST(integer) = \{ integer \}$

$FIRST(char) = \{ char \}$

$FIRST(num \text{ dotdot } num) = \{ num \}$



预测分析程序的构造

预测分析程序的转换图
转换图的工作过程
转换图的化简
预测分析程序的实现



预测分析程序的转换图

从文法构造转换图

- 改写文法：
重写文法/消除左递归/ 提取左公因子
- 对每一个非终结符号A，做如下工作：
创建一个初始状态和一个终结状态。
对每一个产生式 $A \rightarrow X_1X_2...X_n$ 创建一条从初态到终态的路径，有向边的标记依次为 $X_1, X_2, ..., X_n$ 。

为预测分析程序建立转换图作为其实现蓝图

- 每一个非终结符号有一张图
- 边的标记可以是终结符号，也可以是非终结符号。
- 在一个非终结符号A上的转移意味着对相应A的过程的调用。
- 在一个终结符号a上的转移，意味着下一个输入符号若为a，则应做此转移



预测分析程序的转换图

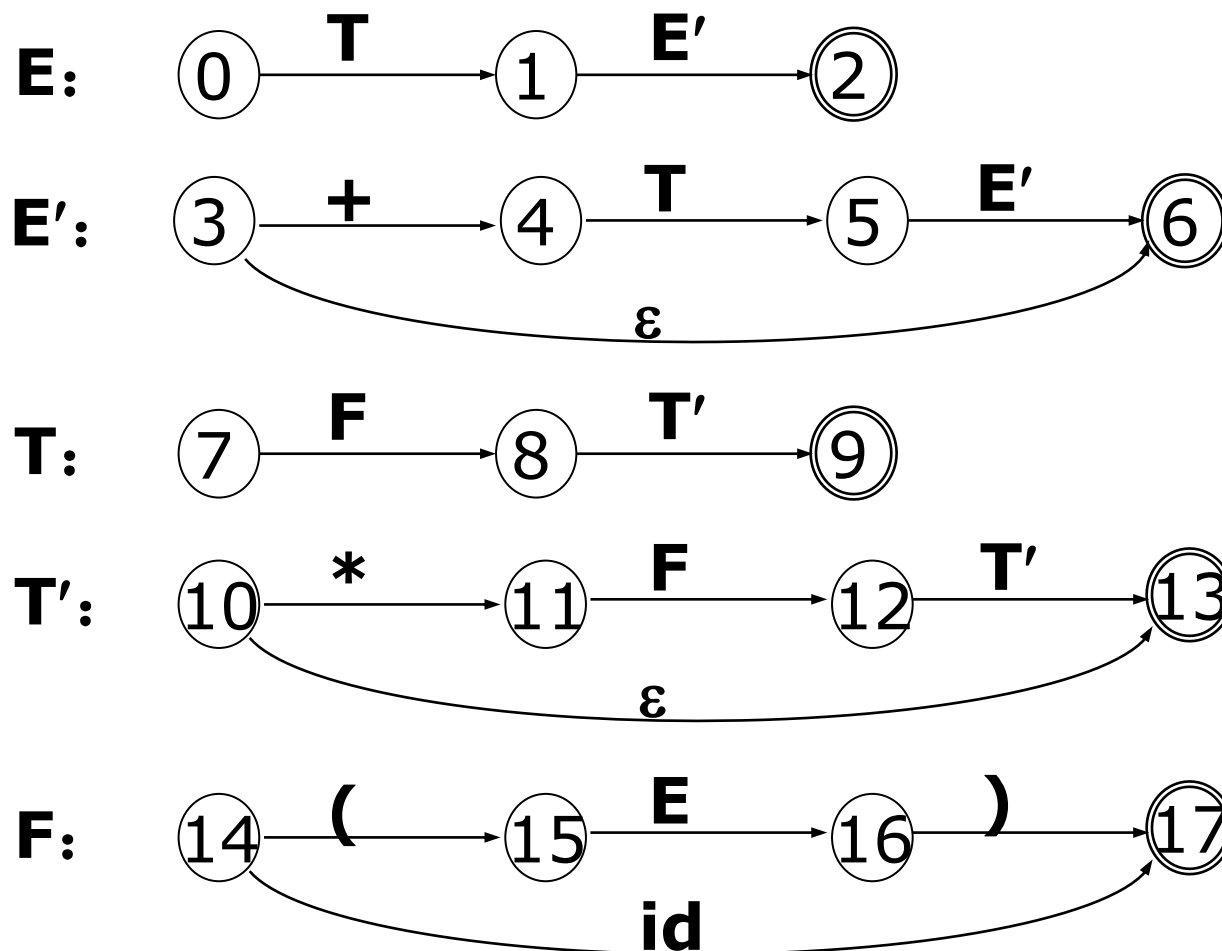
为如下文法构造预测分析程序转换图

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

消除文法中存在的左递归，得到

$E \rightarrow TE'$
 $E' \rightarrow +TE' \mid \epsilon$
 $T \rightarrow FT'$
 $T' \rightarrow *FT' \mid \epsilon$
 $F \rightarrow (E) \mid id$

为每个非终结符号构造转换图：





确定的转换图

定义4.2

如果在一个非终结符号的状态转换图中，每个状态结点都不存在同名的射出边、不存在标号为 ϵ 的边，而且如果有以非终结符号标记的射出边，则此边是该状态结点的唯一射出边，则称该状态转换图是“确定的”。

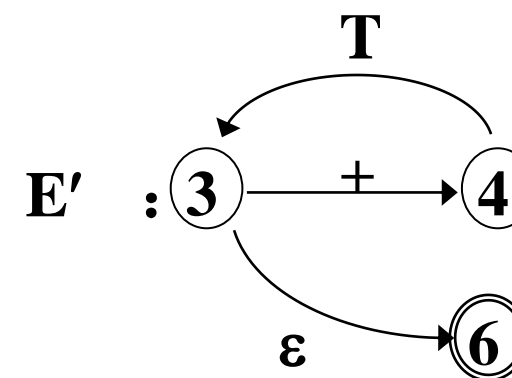
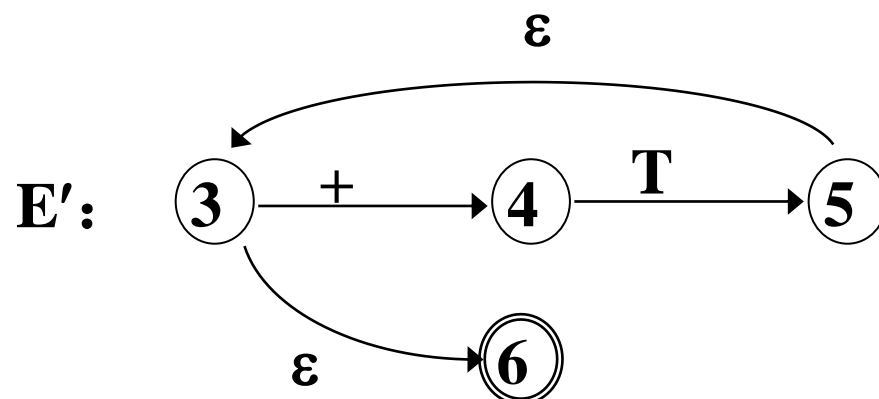
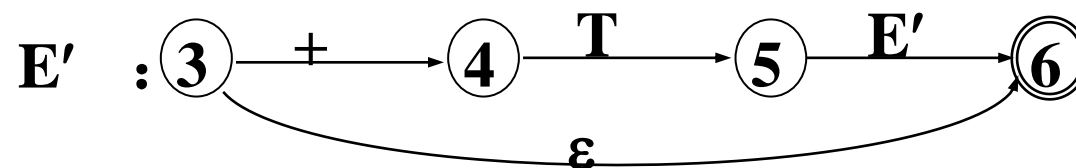
如果一个文法的所有非终结符的状态转换图都是确定的，则可以此基础构建预测分析程序。

如果转换图存在不确定性，可以尝试消除；不能消除不确定性，则不能构造预测分析程序，只能采用带回溯的递归下降分析。



转换图的化简

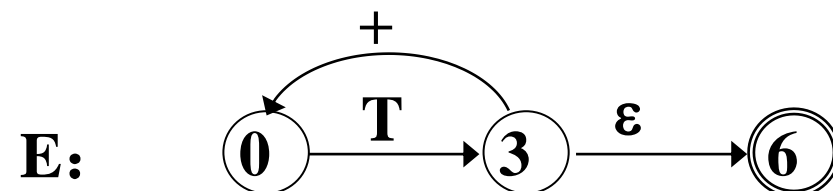
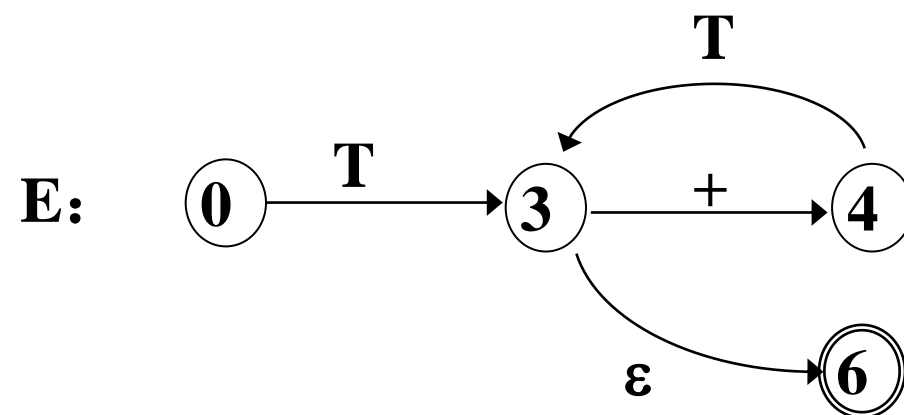
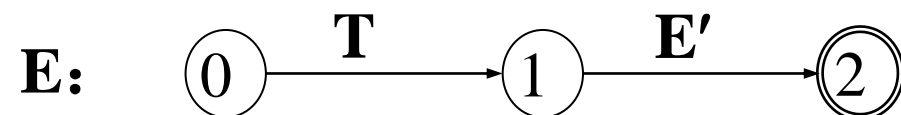
用代入的方法进行化简





转换图的化简

把E'的转换图代入E的转换图：



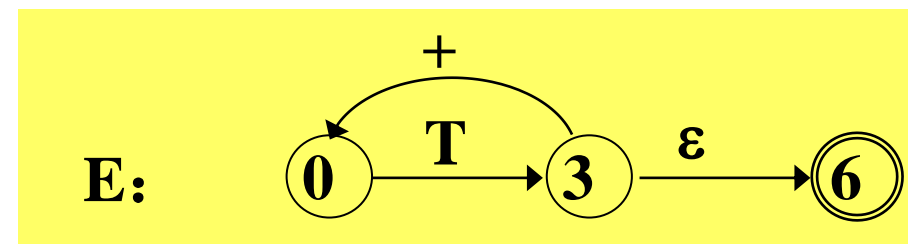


预测分析程序的实现

要求用来描述预测分析程序的语言允许递归调用

E的过程:

```
void procE(void)
{
    procT();
    if (char == '+') {
        forward pointer;
        procE();
    }
}
```



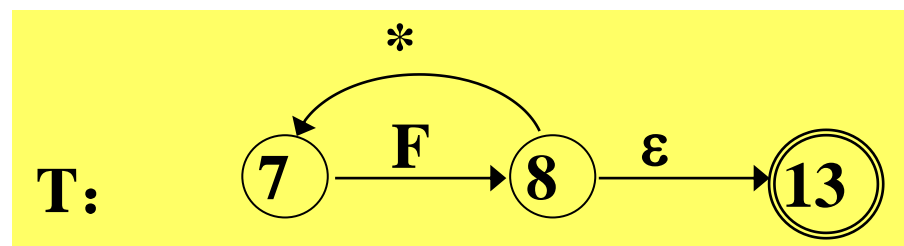


预测分析程序的实现

要求用来描述预测分析程序的语言允许递归调用

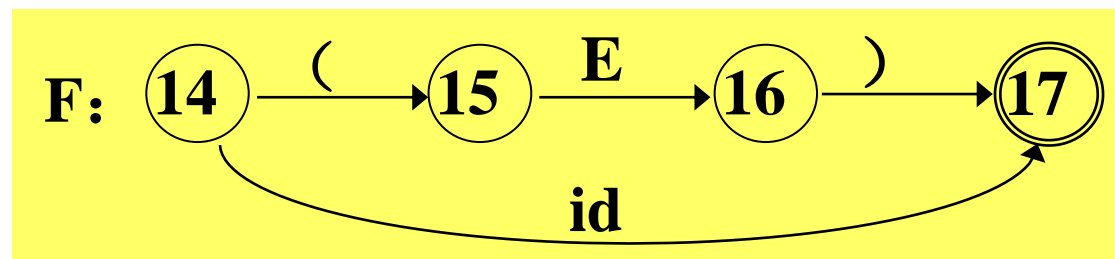
T的过程:

```
void procT(void)
{
    procF();
    if (char == '*') {
        forward pointer;
        procT();
    }
}
```



F的过程:

```
void procF(void)
{
    if (char == '(') {
        forward pointer;
        procE();
        if (char == ')') {
            forward pointer;
        };
        else error();
    };
    else if (char == 'id') {
        forward pointer;
        else error();
    }
}
```





4.5.3 非递归预测分析-LL(1)分析法

基于表驱动的分析法，使用一张分析表和一个栈联合控制，实现对输入符号串的自顶向下分析。

- 预测分析程序的模型及工作过程
- 预测分析表的构造
- LL(1)文法
- 预测分析方法中的错误处理示例



4.5.3.1 预测分析程序的模型及工作过程

输入缓冲区:

存放被分析的输入符号串，串后随右尾标志符\$。

符号栈:

存放一系列文法符号，\$存于栈底。分析开始时，先将\$入栈，以标识栈底，然后再将文法的开始符号入栈。

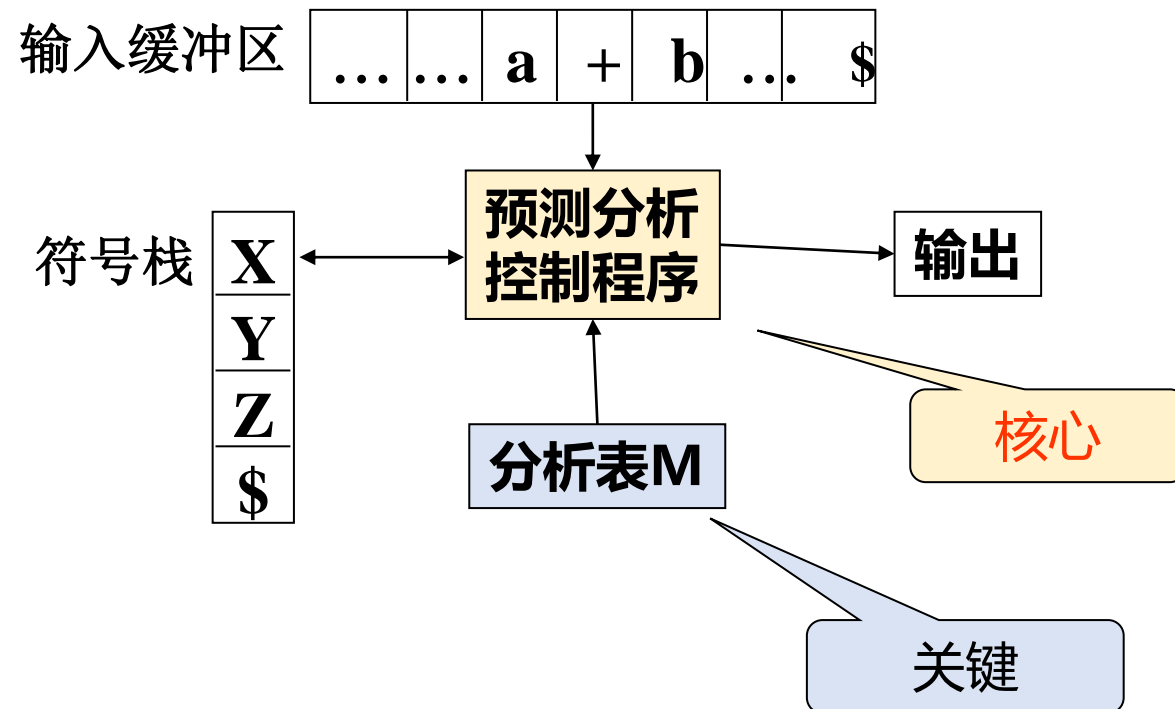
分析表:

二维数组 $M[A, a]$ ， $A \in V_N$ ， $a \in V_T \cup \{\$ \}$ 。

根据给定的A和a，在分析表M中找到将被调用的产生式。

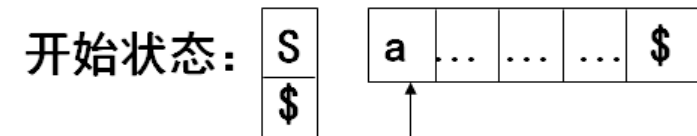
输出流:

分析过程中不断产生的产生式序列。

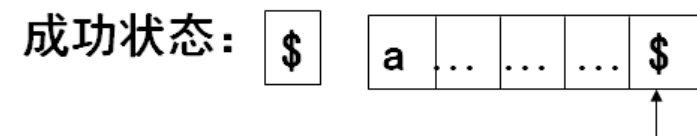




4.5.3.2 预测分析控制程序



根据栈顶符号X和当前输入符号a，决定分析动作有4种可能：



- (1) $X=a=\$$ ，宣告分析成功，停止分析；
- (2) $X=a\neq \$$ ，从栈顶弹出X，输入指针前移一个位置；
- (3) $X\in V_T$ ，但 $X\neq a$ ，报告发现错误，调用错误处理程序，以报告错误及进行错误恢复；

- (4) 若 $X\in V_N$ ，访问分析表 $M[X, a]$

$$M[X, a]=X\rightarrow Y_1Y_2\cdots Y_n$$

先将X从栈顶弹出，然后把产生式的右部符号串按反序
推入栈中（即按 Y_n 、 \cdots 、 Y_{n-1} 、 Y_2 、 Y_1 的顺序）；

$$M[X, a]=X\rightarrow\epsilon \quad \text{从栈顶弹出X；}$$

$$M[X, a]=\text{error} \quad \text{调用出错处理程序}$$



4.5.3.3 算法：非递归预测分析方法

输入：输入符号串 ω ，文法 G 的一张预测分析表 M 。

输出：若 ω 在 $L(G)$ 中，则输出 ω 的最左推导，否则报告错误。

方法：分析开始时， $\$$ 在栈底，文法开始符号 S 在栈顶， $\omega\$$ 在输入缓冲区中置 ip 指向 $\omega\$$ 的第一个符号；

```
do {  
    令 $X$ 是栈顶符号， $a$ 是 $ip$ 所指向的符号；  
    if ( $X$ 是终结符号或 $\$$ ) {  
        if ( $X = a$ ) {  
            从栈顶弹出 $X$ ； $ip$ 前移一个位置；  
        };  
        else error();  
    else /*  $X$ 是非终结符号 */  
    if ( $M[X,a] = X \rightarrow Y_1 Y_2 \dots Y_k$ ) {  
        从栈顶弹出 $X$ ；  
        把 $Y_k, Y_{k-1}, \dots, Y_2, Y_1$ 压入栈， $Y_1$ 在栈顶；  
        输出产生式 $X \rightarrow Y_1 Y_2 \dots Y_k$ ；  
    };  
    else error();  
}while( $X \neq \$$ ) /* 栈不空，继续 */
```



非递归预测分析方法示例

示例：文法4.4的预测分析表M，试分析输入串 $\text{id}+\text{id}*\text{id}$ 。

	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		



栈	输入	输出	左句型
\$ E	id + id * id \$		E
\$ E' T	id + id * id \$	$E \rightarrow TE'$	T E'
\$ E' T' F	id + id * id \$	$T \rightarrow FT'$	F T' E'
\$ E' T' id	id + id * id \$	$F \rightarrow id$	id T' E'
\$ E' T'	+ id * id \$		id T' E'
\$ E'	+ id * id \$	$T' \rightarrow \epsilon$	id E'
\$ E' T +	+ id * id \$	$E' \rightarrow +TE'$	id + T E'
\$ E' T	id * id \$		id + T E'
\$ E' T' F	id * id \$	$T \rightarrow FT'$	id + F T' E'
\$ E' T' id	id * id \$	$F \rightarrow id$	id + id T' E'
\$ E' T'	* id \$		id + id T' E'
\$ E' T' F *	* id \$	$T' \rightarrow *FT'$	id + id * F T' E'
\$ E' T' F	id \$		id + id * F T' E'
\$ E' T' id	id \$	$F \rightarrow id$	id + id * id T' E'
\$ E' T'	\$		id + id * id T' E'
\$ E'	\$	$T' \rightarrow \epsilon$	id + id * id E'
\$	\$	$E' \rightarrow \epsilon$	id + id * id



4.5.3.4 预测分析表的构造

改写文法

FIRST集合及其构造

FOLLOW集合及其构造

预测分析表的构造



FIRST集合及其构造

FIRST集合

定义：对任何文法符号串 $\alpha \in (V_T \cup V_N)^*$ ，
FIRST(α)是 α 可以推导出的开头终结符号集合

描述为：

$$\text{FIRST}(\alpha) = \{ a \mid \alpha \xRightarrow{*} a \cdots, a \in V_T \}$$

若 $\alpha \xRightarrow{*} \varepsilon$ ，则 $\varepsilon \in \text{FIRST}(\alpha)$ 。



FIRST集合及其构造

构造每个文法符号 $X \in V_T \cup V_N$ 的FIRST(X)

- 若 $X \in V_T$, 则 $\text{FIRST}(X) = \{X\}$;
- 若 $X \in V_N$, 且有产生式 $X \rightarrow a...$, 其中 $a \in V_T$, 则把 a 加入到 $\text{FIRST}(X)$ 中;
- 若 $X \rightarrow \varepsilon$ 也是产生式, 则 ε 也加入到 $\text{FIRST}(X)$ 中。
- 若 $X \rightarrow Y...$ 是产生式, 且 $Y \in V_N$, 则把 $\text{FIRST}(Y)$ 中的所有非 ε 元素加入到 $\text{FIRST}(X)$ 中;
- 若 $X \rightarrow Y_1Y_2...Y_k$ 是产生式, 如果对某个 i ,
 $\text{FIRST}(Y_1)$ 、 $\text{FIRST}(Y_2)$ 、...、 $\text{FIRST}(Y_{i-1})$ 都含有 ε ,
即 $Y_1Y_2...Y_{i-1} \Rightarrow \varepsilon$,
则把 $\text{FIRST}(Y_i)$ 中的所有非 ε 元素加入到 $\text{FIRST}(X)$ 中;
- 若所有 $\text{FIRST}(Y_i)$ 均含有 ε , 其中 $i=1, 2, ..., k$, 则把 ε 加入到 $\text{FIRST}(X)$ 中。



FOLLOW集合及其构造

FOLLOW集合

定义：假定S是文法G的开始符号，对于G的任何非终结符号A，集合 FOLLOW(A)是在所有句型中，紧跟A之后出现的终结符号或\$组成的集合。

描述为：

$$\text{FOLLOW}(A) = \{ a \mid S \xRightarrow{*} \dots Aa \dots, a \in V_T \}$$

特别地，若 $S \xRightarrow{*} \dots A$ ，则规定 $\$ \in \text{FOLLOW}(A)$ 。



构造每个非终结符号A的集合FOLLOW(A)

- 对文法开始符号S，置\$于FOLLOW(S)中，\$为输入符号串的右尾标志。
- 若 $A \rightarrow \alpha B \beta$ 是产生式，则把FIRST(β)中的所有非 ϵ 元素加入到FOLLOW(B)中。
- 若 $A \rightarrow \alpha B$ 是产生式，或 $A \rightarrow \alpha B \beta$ 是产生式并且 $\beta \xRightarrow{*} \epsilon$ ，则把FOLLOW(A)中的所有元素加入到FOLLOW(B)中。
- 重复此过程，直到所有集合不再变化为止。



构造每个非终结符号A的集合FOLLOW(A)

示例：构造文法4.4中每个非终结符号的FIRST集合和FOLLOW集合

	FIRST	FOLLOW
E	(, id	\$,)
E'	+, ϵ	\$,)
T	(, id	\$,), +
T'	*, ϵ	\$,), +
F	(, id	\$,), +, *

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$



算法4.2 预测分析表的构造方法

输入：文法G

输出：文法G的预测分析表M

方法：

```
for (文法G的每个产生式 $A \rightarrow \alpha$ ) {  
    for (每个终结符号 $a \in \text{FIRST}(\alpha)$ )  
        把 $A \rightarrow \alpha$ 放入 $M[A, a]$ 中;  
    if ( $\epsilon \in \text{FIRST}(\alpha)$ )  
        for (任何 $b \in \text{FOLLOW}(A)$ )  
            把 $A \rightarrow \alpha$ 放入 $M[A, b]$ 中;  
};  
for (所有无定义的 $M[A, a]$ ) 标上错误标志.
```



示例：为文法构造预测分析表

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

	FIRST	FOLLOW
E	(, id	\$,)
E'	+, ε	\$,)
T	(, id	\$,), +
T'	*, ε	\$,), +
F	(, id	\$,), +, *

	id	+	*	()	\$
E						
E'						
T						
T'						
F						



示例：为文法构造预测分析表

	id	+	*	()	\$
E	$E \rightarrow TE'$	err	err	$E \rightarrow TE'$	err	err
E'	err	$E' \rightarrow +TE'$	err	err	$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$	err	err	$T \rightarrow FT'$	err	err
T'	err	$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$	err	$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$	err	err	$F \rightarrow (E)$	err	err

4.5.3.5 LL(1)文法

例：考虑如下映射程序设计语言中if语句的文法

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \varepsilon$$

$$E \rightarrow b$$

(文法4.5)

构造各非终结符号的FIRST和FOLLOW集合：

	S	S'	E
FIRST	i, a	e, ε	b
FOLLOW	\$, e	\$, e	t

应用算法4.2，构造该文法的分析表：

	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow eS$ $S' \rightarrow \varepsilon$			$S' \rightarrow \varepsilon$
E		$E \rightarrow b$				



4.5.3.5 LL(1)文法

如果一个文法的预测分析表M不含多重定义的表项，则称该文法为LL(1)文法。

LL(1)的含义：

- 第一个L表示从左至右扫描输入符号串

- 第二个L表示生成输入串的一个最左推导

- 1表示在决定分析程序的每步动作时，向前看一个符号

LL(1)分析法的特点：

- 分析高效（线性时间）

- 错误定位和诊断信息准确

- 很多开源或商业的生成工具，ANTLR，...



LL(1)文法的判断

根据文法产生式判断:

一个文法是LL(1)文法, 当且仅当它的每一个产生式 $A \rightarrow \alpha | \beta$, 满足:

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \phi \quad \text{并且若} \beta \text{推导出} \varepsilon, \text{ 则} \text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \phi$$

根据分析表判断:

如果利用算法4.2构造出的分析表中不含多重定义的表项, 则文法是LL(1)文法。



4.5.3.6 预测分析方法中的错误处理示例

分析过程中，有两种情况可以发现源程序中的语法错误：

- (1) $X \in V_T$, 但 $X \neq a$;
- (2) $X \in V_N$, 但 $M[X, a]$ 为空。

错误处理方法：

第(1)种情况，弹出栈顶的终结符号；

第(2)种情况，跳过剩余输入符号串中的若干个符号，直到可以继续进行分析（同步符号）为止。



同步符号的选择

- 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;
- 可以把表示语句开始的一些关键字加入到同步记号集中;
- 如果栈顶的终结符不能被匹配，就可以弹出该终结符，此时相当于把所有的符号都看作同步符号



带有同步化信息的分析表

带有同步化信息的分析表的构造

对于 $A \in V_N$, $b \in \text{FOLLOW}(A)$, 若 $M[A,b]$ 为空, 则加入 “synch”

带有同步化信息的分析表的使用

- 若 $M[A,b]$ 为空, 则跳过a;
- 若 $M[A,b]$ 为synch, 则弹出A。

示例：带有同步化信息的分析表

构造文法4.4的带有同步化信息的分析表

$E \rightarrow TE'$
 $E' \rightarrow +TE' \mid \epsilon$
 $T \rightarrow FT'$
 $T' \rightarrow *FT' \mid \epsilon$
 $F \rightarrow (E) \mid id$

	FIRST	FOLLOW
E	(, id	\$,)
E'	+, ε	\$,)
T	(, id	\$,), +
T'	*, ε	\$,), +
F	(, id	\$,), +, *

	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	synch	synch	$F \rightarrow (E)$	synch	synch



栈	输入	输出
\$E	*id*+id\$	出错, $M[E,*]$ =空白, 跳过*
\$E	id*+id\$	$E \rightarrow TE'$
\$E'T	id*+id\$	$T \rightarrow FT'$
\$E'T'F	id*+id\$	$F \rightarrow id$
\$E'T'id	id*+id\$	
\$E'T'	*+id\$	$T' \rightarrow *FT'$
\$E'T'F*	*+id\$	
\$E'T'F	+id\$	出错, $M[F,+]$ =synch, 弹出F
\$E'T'	+id\$	$T' \rightarrow \epsilon$
\$E'	+id\$	$E' \rightarrow +TE'$
\$E'T+	+id\$	
\$E'T	id\$	$T \rightarrow FT'$
\$E'T'F	id\$	$F \rightarrow id$
\$E'T'id	id\$	
\$E'T'	\$	$T' \rightarrow \epsilon$
\$E'	\$	$E' \rightarrow \epsilon$
\$	\$	

	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	synch	synch	$F \rightarrow (E)$	synch	synch

■ 带有同步化信息的分析表的使用

- 若 $M[A,b]$ 为空, 则跳过a;
- 若 $M[A,b]$ 为synch, 则弹出A。

- $X \in V_T$, 但 $X \neq a$ 弹出栈顶的终结符号;



可选集(SELECT)

定义:在自顶向下的分析中, 为每个产生式定义一个可选集 (SELECT)。
若 $A \rightarrow \alpha$ 为文法 G 的产生式, 且编号为 i , 则它的可选集 $\text{SELECT}(i)$ 的定义如下:

- 如果 $\alpha \neq \varepsilon$, 且 α 是不可空的, 则 $\text{SELECT}(i) = \text{FIRST}(\alpha)$;
- 如果 $\alpha \neq \varepsilon$, 且 α 是可空的, 则 $\text{SELECT}(i) = \text{FIRST}(\alpha) \cup \text{FOLLOW}(A)$
- 如果 $\alpha = \varepsilon$, 则 $\text{SELECT}(i) = \text{FOLLOW}(A)$

$$\text{SELECT}(i) = \text{FIRST_S}(\alpha)$$

可选集的作用

只要文法的某一非终结符 A 的各个可选集互不相交, 那么语法分析程序就可以根据当前输入符号 A 和 A 的可选集来正确选择 A 的某个候选式



LL(1)文法与可选集

定义：

一个上下文无关的文法称为是LL(1)文法，当且仅当同一非终结符的各个产生式的**可选集** (SELECT) 互不相交

LL(1)的含义：

L—自左向右的扫描输入串

L—用最左推导方法分析句子

1—向前查看一个输入符号，LL(k)



思考



重慶大學
CHONGQING UNIVERSITY

是否所有语言都有LL(1)文法?
能否将非LL(1) 文法变换为等价的LL(1)文法?



思考



如何识别列表中出现赋值语句的情况？例如：[a,b=c,[d,e]]

```
list      : ' [ ' elements ' ] ' ;  
elements : element ( ' , ' element ) * ;  
element  : NAME '=' NAME  
          | NAME  
          | list  
          ;
```

可以用LL (1) 分析法解析吗？怎样实现？



思考



如何识别列表中出现赋值语句的情况？例如：[a,b=c,[d,e]]

```
list      : ' [ ' elements ' ] ' ;  
elements : element ( ' , ' element ) * ;  
element  : NAME ' = ' NAME  
          | NAME  
          | list  
          ;
```

```
element : NAME ( ' = ' NAME ) ?  
        | list  
        ;
```



思考



如何识别列表中出现赋值语句的情况？例如：[a,b=c,[d,e]]

```
list      : ' [ ' elements ' ] ' ;  
elements : element ( ' , ' element ) * ;  
element  : NAME '=' NAME  
          | NAME  
          | list  
          ;
```

还有其它方法增加解析器的向前看能力吗？



思考



重庆大学
CHONGQING UNIVERSITY

课堂作业：如何实现LL（2）解析？请你给出思路和方法。



思考



重庆大学
CHONGQING UNIVERSITY

课堂作业：如何实现LL（K）解析？请你给出思路和方法。

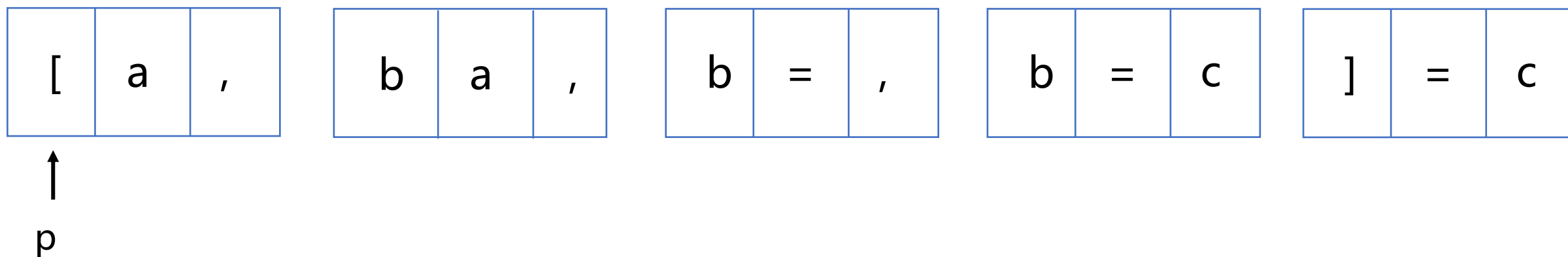


扩展：LL (K) 递归下降的语法解析器

构建环形向前看缓冲区

词法单元缓冲区：大小为K

下标p：沿着已知缓冲区移动





扩展：LL (K) 递归下降的语法解析器模板(1)



```
Lexer input;           //待处理词法单元的来源
Token[ ] lookahead ;   //环形缓冲区
int k;                 //向前看符号的个数
int p ;               //环形缓冲区中装填下一个词法单元的位置
```

```
public Parser(Lexer input, int k) {
    this.input = input;
    this.k=k;
    lookahead = new Token[k] ;           //开辟向前看缓冲区
    for (int i=1; i<=k; i++) consume( ); //用k个向前看符号初始化缓冲区
```

```
Public void consume ( ) {
    lookahead[p] = input.nextToken( ); //在下一个位置上放入词法单元
    p= (p+1) % k;                      //自增下标
}
```

//更多代码细节，请查阅 [http://www.antlr.org](http://wwwantlr.org)



扩展：LL (K) 递归下降的语法解析器模板(2)

```
public Token LT (int i) {  
    return lookahead[ (p+i-1)%k];    //环式取值  
}  
public int LA (int i){  
    return LT(i).type;  
}
```

```
public void match(int x) {  
    if ( LA(1) ==x ) consume();  
    else throw new Error( "expecting" + input.getTokenName(x)+" ; found" +LT(1));  
}
```

//更多代码细节，请查阅 [http://www.antlr.org](http://wwwantlr.org)



扩展：LL (K) 递归下降的语法解析器模板(3)



```
/** element : NAME ' = ' NAME | NAME | list ; assignment, NAME or list*/
```

```
void element( ){  
    if (LA(1)==LookaheadLexer.NAME && LA(2)==LookaheadLexer.EQUALS) {  
        match(LookaheadLexer.NAME);  
        match(LookaheadLexer.EQUALS);  
        match(LookaheadLexer.NAME);  
    }  
    else if ( LA(1) == LookaheadLexer.NAME ) match(LookaheadLexer.NAME );  
    else if ( LA(1) == LookaheadLexer.LBRACK ) list( );  
    else throw new Error( "expecting name or list; found" +LT(1));  
}
```

```
LookaheadLexer lexer = new LookaheadLexer(args[0]); //命令行参数为待解析语句  
LookaheadParser parser = new LookaheadParser(lexer,2);  
Parser.list();      // 从规则列表开始解析  
//更多代码细节，请查阅 http://www.antlr.org
```




思考



请问以下的C语言函数定义和声明语句，进行向前看解析，会存在什么问题？

```
void bar() {...} // 函数定义  
void bar();      // 函数声明（前向声明）
```

```
function : def | decl ;  
def: functionHead '{ ' body '}' ; //如 "void bar() {}"  
decl: functionHead ';'           //如 "void bar() ;"  
functionHead: ...;               //如 "int (*foo) (int *f[],float)"
```



扩展：利用任意多的向前看符号进行解析

带有**回溯机制**的解析器进行**推演解析**

例如下面的function规则解析函数：

```
Void function ( ) {  
    if ( <<推演结果和def吻合>> ) def( ) ;  
    else if ( <<推演结果和decl吻合>> ) decl( ) ;  
    else throw new RecognitionException( "expecting funtion" );  
}
```

！！ 推演匹配里隐含了对解析选项的排序。

回溯解析器会按照一定的次序对各个解析选项进行尝试



思考



重庆大学
CHONGQING UNIVERSITY

请思考：回溯推演解析需要考虑哪些问题，如何改进？

- 1、基本框架
- 2、缓冲区管理
- 3、回溯时执行操作



扩展：利用任意多的向前看符号进行解析

1、基本框架，采用语法错误（抛出的异常）来引导解析过程。

```
public void <<rule>> throws RecognitionException{
    if ( speculate_ <<alt1>>() ) {                //尝试alt1
        <<匹配-alt1>>
    }
    else if ( speculate_ <<alt2>>() ) {            //尝试alt2
        <<匹配-alt2>>
    }
    ...
    else if ( speculate_ <<altN>>() ) {            //尝试altN
        <<匹配-alt1N>>
    }
    // 出错，无匹配选项
    else throw new NoViableException( "expecting <<rule>>" )
}
//更多代码细节，请查阅 http://www.antlr.org
```



扩展：利用任意多的向前看符号进行解析

2、固定大小的缓冲区，需要时扩展

```
public Boolean speculate_<<alt>>() {  
    boolean success = true;  
    mark();                //标记当前的位置，以便将来放回  
    try{ <<匹配-alt>>}    //尝试匹配选项  
    catch (RecognitionException e) {success =false; }  
    release();             //不管成功与否，都要放回  
    return success;  
}
```

//更多代码细节，请查阅 <http://www.antlr.org>

用栈存放缓冲区下标，以处理嵌套的回溯



扩展：利用任意多的向前看符号进行解析

3、回溯时执行操作

完成特定功能的解析器，在解析中可能会执行一些与解析无关的操作。

- 禁止执行此类操作；
- 只禁止副作用的操作；
- 重复解析成功匹配的选项；



扩展：利用任意多的向前看符号进行解析

记忆解析器

额外使用少量内存，记录回溯过程中不完整的解析结果，将整个解析过程的时间复杂度降到线性水平。

带有记忆机制的递归下降解析器又称为packrat parser，Bryan Ford 在《Packrat parsing:: simple,powerful,lazy,linear time,functional pearl》提出。

请大家思考，可以如何实现记忆解析器？请给出你的设计思路 and 方案。



扩展：记忆解析器

```
s      : expr '!' //假设回溯解析器先尝试这个选项
      | expr ';' //再尝试这个选项
      ;
expr : ... //匹配形如 " (3+4) " 的输入
```

为记录这种不完整的解析结果，每个文法规则都要有自己的记忆映射表，记录词法单元缓冲区的位置下标当前的状态。状态可以分成三种情况：unknown、failed 和 succeeded。



扩展：记忆解析器

把解析器增强为记忆解析器后，需要给每个规则方法增加一个负责记录的方法，带有记忆机制的规则解析模板如下：

```
Map<Integer,Integer> <<rule>>_memo = new HashMap<Integer,Integer>();
public void<<rule>>() throws RecognitionException{
    boolean failed = false;
    int startTokenindex = index() ;
    if (isSpeculating() && alreadyParsedRule(<<rule>>_memo) ) return;
    //那么之前没有解析过，现在来解析
    try {_<<rule>>();}
    catch(RecognitionException re) {failed =true; throw re;}
    finally{
        //不管解析是否成功，回溯时都必须记录解析结果
        if (isSpeculating() )
            memorize(<<rule>>_memo,startTokenIndex,failed);
    }
}
```

//更多代码细节，请查阅 <http://www.antlr.org>



扩展：记忆解析器

实现方法alreadyParsedRule:

```
public boolean alreadyParsedRule(Map<Integer,Integer> memorization)
    throws PreviousParserFailedException
{
    Integer memoI = memorization.get(index());
    if (memoI ==null ) return false;
    int memo = memoI.inValue();
    System.out.println( "Parsed list before at index" + index() +
                        ";skip ahead to      token index" + memo+"":
                        +lookahead.get(memo).text);
    if (memo==FAILED) throw new PreviousParserFailedException();
    seek(memo);
    return true;
}

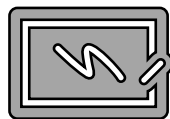
//更多代码细节，请查阅 http://www.antlr.org
```



扩展：记忆解析器

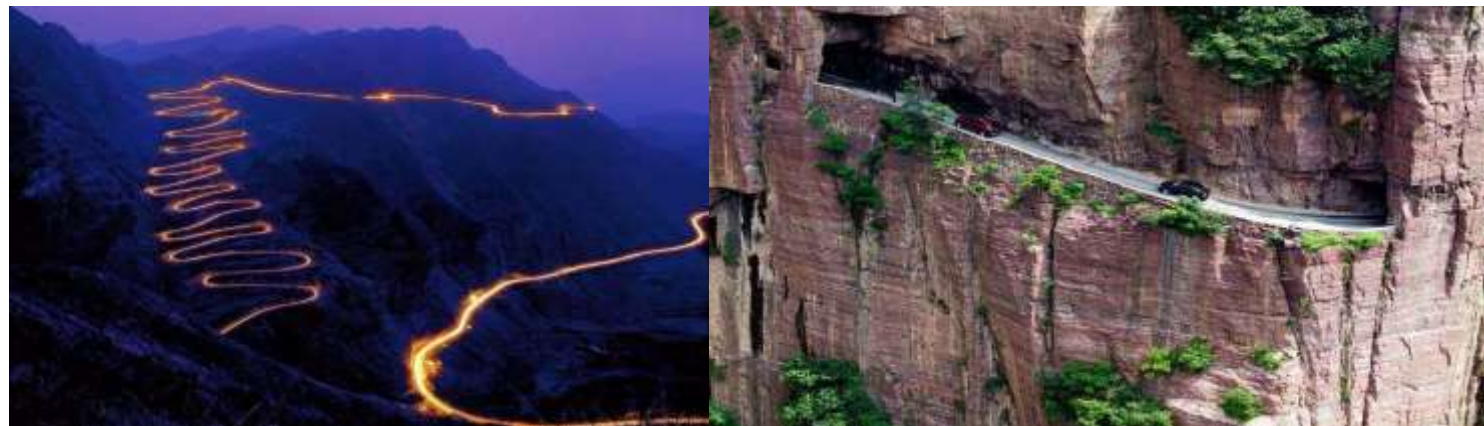
实现方法memoize:

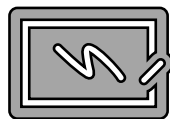
```
public void memoize(Map<Integer,Integer> memorization,  
                    int startTokenIndex, Boolean failed)  
{  
    int stopTokenIndex = failed ? FAILED : index();  
    memorization.put(startTokenIndex, stopTokenIndex);  
}  
//更多代码细节, 请查阅 http://www.antlr.org
```



课后作业 1

- 1、请改造文法，消除if语句文法的不确定性
 <IF语句> \rightarrow if E then S1 else S2
 <IF语句> \rightarrow if E then S1
- 2、请对下面的文法进行消除左递归等价变换
 $S \rightarrow (L) \mid a$
 $L \rightarrow L, S \mid S$



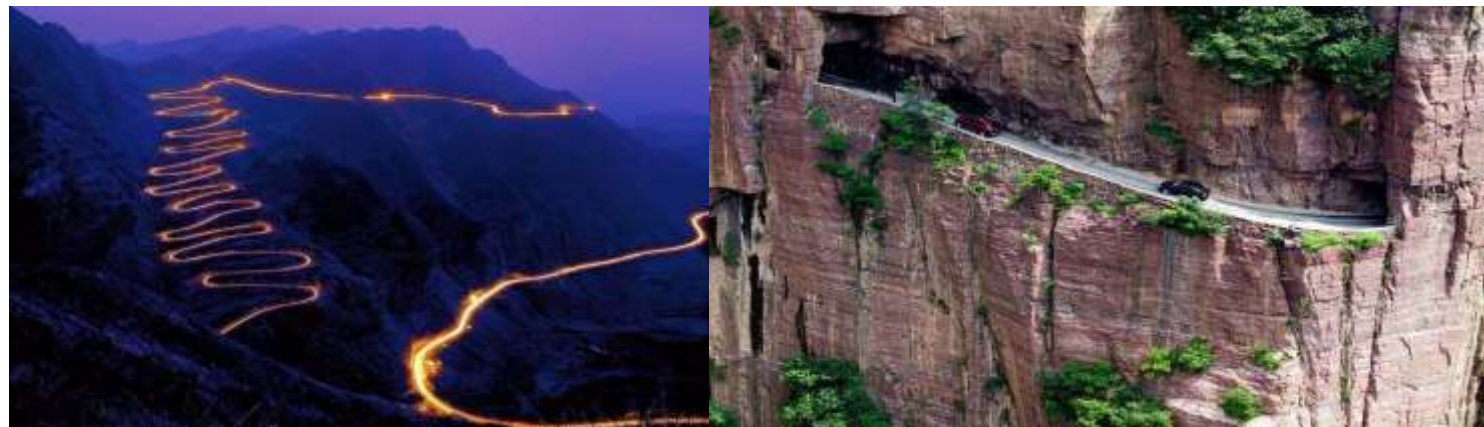


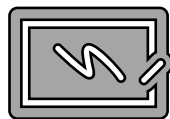
课后作业 2

考虑如下文法G:

$E \rightarrow A|B$ $A \rightarrow \text{num} | \text{id}$ $B \rightarrow (L)$ $L \rightarrow LE|E$

- (1) 消除文法中的左递归
- (2) 为改写后文法中的非终结符号构造FIRST集合及FOLLOW集合
- (3) 说明改写后的文法是LL (1) 文法, 并构造LL (1) 分析表
- (4) 给出输入符号串(a(b(2))(c)的预测分析过程





扩展

阅读相应材料，思考：

- 1、可以如何改进LL(1)?
- 2、LL(2), LL(k)应该如何实现?
- 3、请自行查阅资料，了解记忆解析器、谓词解析器等方法

