**实验项目：LR分析器的设计**

**计算机224 2208010402 陈大地**

**​1. 语法分析的含义**

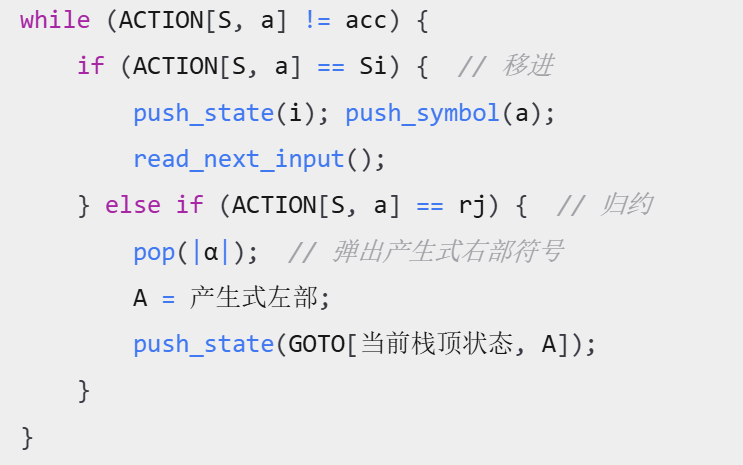
**理论背景**：  
语法分析是编译器的核心阶段，负责验证词法分析输出的符号序列是否符合源语言的语法结构，并构建语法树（或中间表示）。实现 LR (1) 语法分析器，处理移进 - 归约冲突，通过向前查看符号解决冲突。

* ​**自底向上分析（LR分析法）​**：
  + 从输入符号串出发，通过逐步归约（反向构造最右推导）生成语法树。
  + ​**关键操作**：
    - ​**移进（Shift）​**：将输入符号压入符号栈，并转移到新状态。
    - ​**归约（Reduce）​**：用产生式左部替换栈顶的符号序列，并根据GOTO表转移状态。
  + ​**优势**：能处理更复杂的文法（如含左递归、二义性文法需特殊处理）。

**实验实现**：

* ​**数据结构**：
  + 状态栈 a[] 和符号栈 b[] 分别记录当前分析状态和已识别的符号。
  + action 和 goto 表预定义状态转移规则。：构造 ACTION 表和 GOTO 表，实现自下而上语法分析。
* ​**核心代码逻辑**：

cpp:



**​2. 活前缀、项目集的含义及求解**

**理论背景**：

* ​**活前缀（Viable Prefix）​**：
  + 规范句型的前缀，且该前缀不包含句柄后的任何符号。
  + ​**意义**：确保在分析过程中不会错过任何可能的归约。
* ​**项目集（Item Set）​**：
  + 形式为 A → α·β, a，其中 · 表示当前分析位置，a 是向前搜索符（LR(1)特有）。
  + ​**闭包（Closure）​**：若存在 A → α·Bβ, a，需添加所有 B → ·γ, b（b ∈ FIRST(βa)）。
  + ​**转移（Goto）​**：根据符号 X 将项目集从状态 I 转移到状态 J。

**实验实现**：

* ​**预定义项目集规范族**：
  + 实验中通过 action 和 goto 表隐式构造项目集（如状态0到状态9的转移）。
  + ​**示例**：

cpp:

​

**活前缀的动态维护**：

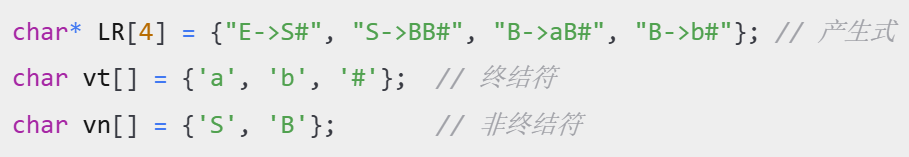
* + 状态栈和符号栈共同构成当前活前缀。例如，状态栈 0 3 6 和符号栈 a B a 对应的活前缀为 a B a。

**​3. LR分析表的构造**

**理论步骤**：

1. ​**构建项目集规范族**：通过闭包和转移生成所有可能的状态。
2. ​**填充ACTION表**：
   * 移进：若项目为 A → α·aβ, b，则 ACTION[i, a] = Sj（j为转移后的状态）。
   * 归约：若项目为 A → α·, a，则 ACTION[i, a] = rk（k为产生式编号）。
3. ​**填充GOTO表**：若从状态 i 通过非终结符 B 转移到状态 j，则 GOTO[i, B] = j。

**实验实现**：

* ​**预定义文法**：
* ​**ACTION/GOTO表解析**：
  + action[0][0] = "S3#"：状态0下输入 a 时移进到状态3。
  + goto1[0][1] = 2：状态0下归约出 B 后转移到状态2。

**​4. 语法范畴的自下而上分析过程**

**详细步骤**：

1. ​**初始化**：状态栈为 [0]，符号栈为 [#]，输入串为 a b #。
2. ​**移进与归约**：
   * ​**状态0**：输入 a，查表 ACTION[0][a] = S3，压入状态3和符号 a。
   * ​**状态3**：输入 b，查表 ACTION[3][b] = r3，按 B→b 归约，弹出 b，压入 B 并转移到 GOTO[0][B] = 2。
3. ​**接受状态**：最终 ACTION[1][#] = acc，分析成功。

**核心代码实现（C++）**

Cpp:

// LR(1)语法分析器核心代码

#include<stdio.h>

#include<string.h>

char \*action[10][3] = {"S3#", "S4#", NULL, // ACTION表

NULL, NULL, "acc",

"S6#", "S7#", NULL,

"S3#", "S4#", NULL,

"r3#", "r3#", NULL,

NULL, NULL, "r1#",

"S6#", "S7#", NULL,

NULL, NULL, "r3#",

"r2#", "r2#", NULL,

NULL, NULL, "r2#"};

int goto1[10][2] = {1, 2, // GOTO表

0, 0,

0, 5,

0, 8,

0, 0,

0, 0,

0, 9,

0, 0,

0, 0,

0, 0};

char vt[3] = {'a', 'b', '#'}; // 终结符

char vn[2] = {'S', 'B'}; // 非终结符

char \*LR[4] = {"E->S#", "S->BB#", "B->aB#", "B->b#"}; // 产生式

void main() {

int a[10], top1=0, top2=0, top3=0, top=0;

char b[10], c[10], c1;

// 输入处理、状态栈与符号栈操作...

while (action[y][j] != "acc") {

// 移进处理：Sx#

if (copy[0] == 'S') { z = copy[1] - '0'; /\* 入栈 \*/ }

// 归约处理：rx#

if (copy[0] == 'r') { /\* 按产生式归约 \*/ }

}

}

**实验输出示例**：

步骤 | 状态栈 | 符号栈 | 输入串 | ACTION | GOTO

1 | 0 | # | a b # | S3 | -

2 | 0 3 | # a | b # | r3 | 2

3 | 0 2 | # B | # | acc | -

输入 aabb#

步骤 状态栈 符号栈 input串 ACTION GOTO

1 0 # aabb# S3#

2 03 #a abb# S3#

3 033 #aa bb# S4#

4 0334 #aab b# r3# B->b

5 0338 #aaB b# S7#

6 03387 #aaBb # r3# B->b

7 03385 #aaBB # r1# S->BB

8 031 #aS # acc

**实验过程记录**

输入测试串：aabb#

分析步骤：

状态栈从 0 开始，符号栈#，输入aabb#

执行S3移进，状态栈变为 03，符号栈#a

归约B->b，状态栈更新为 02，符号栈#B

最终到达acc状态

**实验总结**

LR (1) 分析能力强于 LL (1)，但手工构造表复杂。实验中发现对长串分析时状态转移易出错，需优化错误处理机制。掌握 LR (1) 语法分析原理，理解项目集规范族构造 实现基于 ACTION/GOTO 表的自下而上分析 处理移进 - 归约冲突，验证 LR (1) 分析能力