

第二章

PL/0编译程序的实现

主要内容



- ❖ **PL/0**语言的规定、举例
- ❖ **PL/0**语言的语法描述图 and 文法
- ❖ **PL/0**编译程序的结构、总体框架等
- ❖ **PL/0**编译程序的词法分析
- ❖ **PL/0**编译程序的语法语义分析
- ❖ **PL/0** 编译程序的语法语义错误处理
- ❖ **PL/0**编译程序的目标代码结构和代码生成

PL0语言的限定



PASCAL语言的子集

- ❖ 数据类型只有整型
- ❖ 标识符的有效长度是10，以字母开始的字母数字串（标识符必须先定义后引用）
- ❖ 数最多为14位
- ❖ 作用域规则（内层可引用包围它的外层定义的标识符）
- ❖ 过程无参，可嵌套定义（最多三层），可递归调用
- ❖ 语句类型：
 - 赋值语句，`if...then...`，`while...do...`，`read`，`write`，`call`，复合语句（`begin... end`），说明语句（`const...`，`var...`，`procedure...`）
- ❖ 13个保留字：`if`，`then`，`while`，`do`，`read`，`write`，`call`，`begin`，`end`，`const`，`var`，`procedure`，`odd`

PL/0程序示例



```
CONST A=10; (* 常量说明部分 *)
VAR   B, C;  (* 变量说明部分 *)
PROCEDURE P;  (* 过程说明部分 *)
    VAR D; (* P的局部变量说明部分 *)
    PROCEDURE Q; (* P的局部过程说明部分 *)
    VAR X;
        BEGIN
            READ(X);
            D:=X;
            WHILE X#0
            DO CALL P;
        END;
    BEGIN
        WRITE(D);
        CALL Q;
    END;
BEGIN
    CALL P;
END.
```

输入圆柱的半径和高，计算一些面积、体积等



❖ var r, h, len, a1, a2, volumn;

❖ begin

❖ read(r);

❖ read(h);



❖ len := 2 * 3 * r;

❖ a1 := 3 * r * r;

❖ a2 := a1 + a1 + len * h;

❖ volumn := a1 * h;



❖ write(len);

❖ write(a1);

❖ write(a2);

❖ write(volumn);

❖ end

计算最大公约数



- var m, n, r, q;
- { 计算m和n的最大公约数 }
- procedure gcd;
- begin
- while r#0 do
- begin
- q := m / n;
- r := m - q * n;
- m := n;
- n := r;
- end
- end;
- begin
- read(m);
- read(n);
- { 为了方便, 规定m >= n }
- if m < n then
- begin
- r := m;
- m := n;
- n := r;
- end;
- begin
- r:=1;
- call gcd;
- write(m);
- end;
- end.

pl/O程序--递归调用



```
var n;  
procedure rec;  
begin  
    if n # 0 then  
        begin  
            write(n);  
            n := n - 1;  
            call rec;  
        end;  
    end;  
begin  
    read(n);  
    call rec;  
end.
```

计算 $\text{sum} = 1! + 2! + \dots + n!$, n 从控制台读入



- var n, m, fact, sum;
- { 递归计算 $\text{fact} = m!$ }
- procedure factorial;
- begin
- if $m > 0$ then
- begin
- $\text{fact} := \text{fact} * m$;
- $m := m - 1$;
- call factorial;
- end;
- end;

- begin
- { 读入 n }
- read(n);
- $\text{sum} := 0$;
- while $n > 0$ do
- begin
- $m := n$;
- $\text{fact} := 1$;
- call factorial;
- $\text{sum} := \text{sum} + \text{fact}$;
- $n := n - 1$;
- end;
- { 输出 $n!$ }
- write(sum);
- end.

PL/0 语言的语法描述图



- ✧ 每个语法单位对应一个语法描述图
- ✧ 一个入口和一个出口的有向图
- ✧ 从入口可到达任何节点
- ✧ 每个节点都可以到达出口
- ✧ 从入口到出口的路径表示该语法单位的一种合法中间形式（短语）
- ✧ 有两种类型的节点



内的文字表示所用到的其他语法单位



或

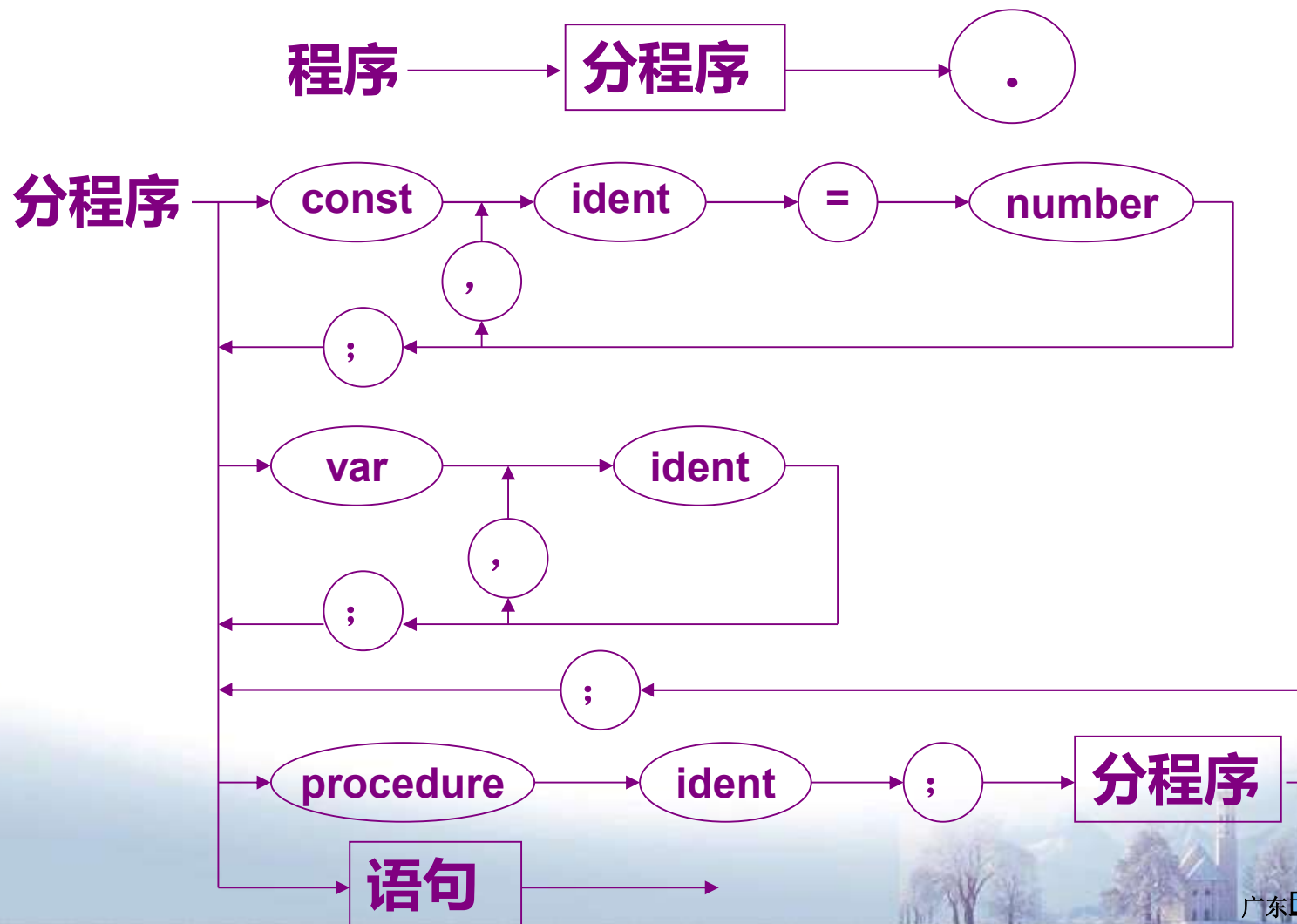


内的文字表示单词符号

PL/0 语言的语法描述图



✧ 例：程序和分程序语法单位的语法描述图



PL/0 语言的EBNF表示



✧ EBNF 的元符号

- ‘< >’ 是用左右尖括号括起来的中文字表示语法构造成分，或称语法单位，为非终结符。
- ‘::=’ 该符号的左部由右部定义，可读作 ‘定义为’
- ‘|’ 表示 ‘或’，即左部可由多个右部定义
- ‘{ }’ 表示花括号内的语法成分可以重复；在不加上下界时可重复0到任意次数，有上下界时为可重复次数的限制
- ‘[]’ 表示方括号内的成分为任选项
- ‘()’ 表示圆括号内的成分优先

PL/0 语言的EBNF表示



✧ 例：PL/0 语言的EBNF表示片断

<程序> ::= <分程序>.

<分程序> ::= [<常量说明部分>] [<变量说明部分>]
[<过程说明部分>] <语句>

<常量说明部分> ::= CONST <常量定义> { , <常量定义> } ;

<常量定义> ::= <标识符> = <无符号整数>

<无符号整数> ::= <数字> { <数字> }

<变量说明部分> ::= VAR <标识符> { , <标识符> } ;

<标识符> ::= <字母> { <字母> | <数字> }

<过程说明部分> ::= <过程首部> <分程序> { ; <过程说明部分> } ;

<过程首部> ::= PROCEDURE <标识符> ;

.....

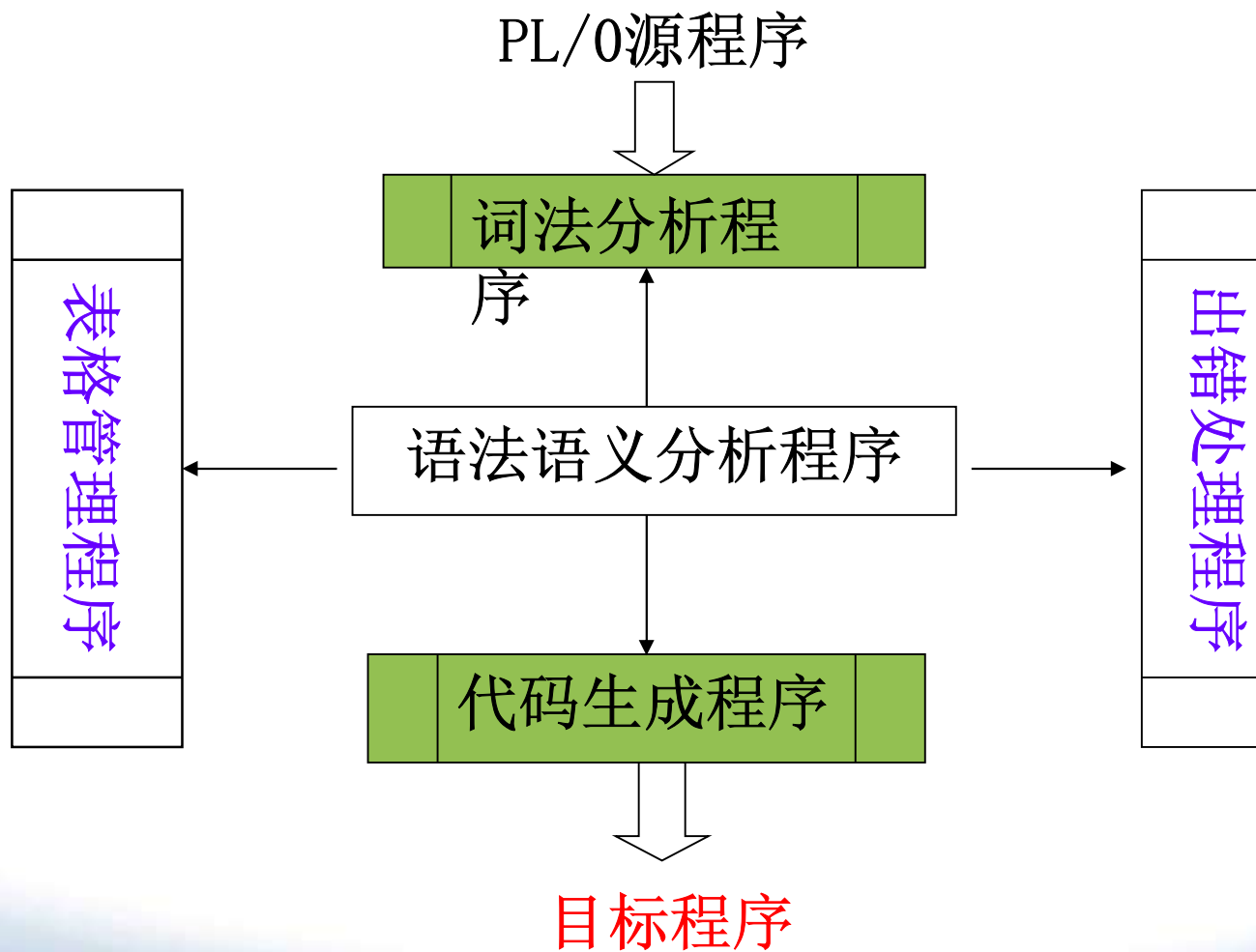
3 PL/0编译程序的结构



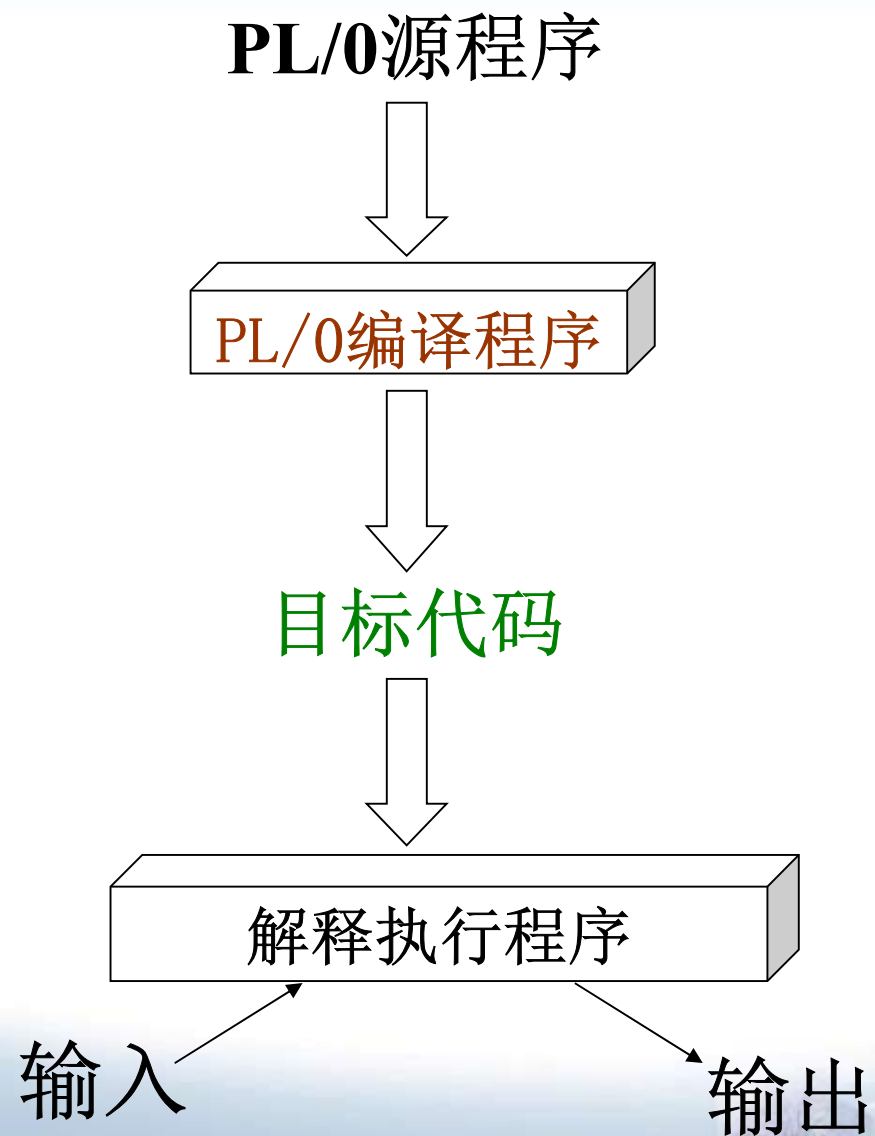
❖ 1、 PL/0编译程序的特点

- (1) 编译程序采用一遍扫描方式，以语法分析程序为核心，词法分析程序和代码生成程序为一个独立过程，被语法分析程序所调用。
- (2) 用表格管理程序建立变量，常量和过程标识符的说明与引用之间的信息联系。
- (3) 用出错处理程序对词法和语法分析遇到的错误给出在源程序中出错的位置和错误性质。
- (4) 当源程序编译正确时， PL/0编译程序自动调用解释执行程序，对目标代码进行解释执行。

3. PL/0编译程序的结构

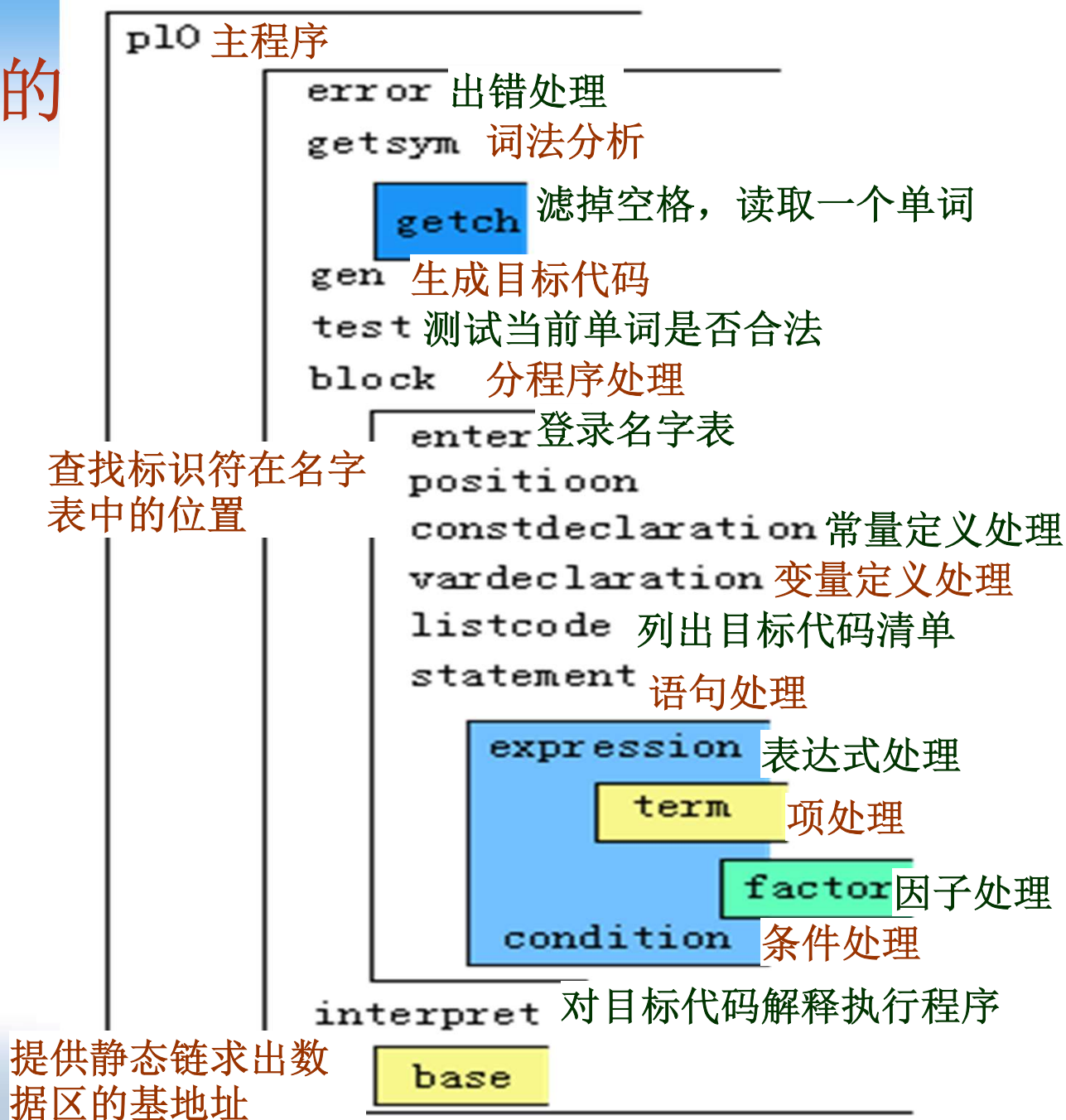


PL/0编译系统的结构框架



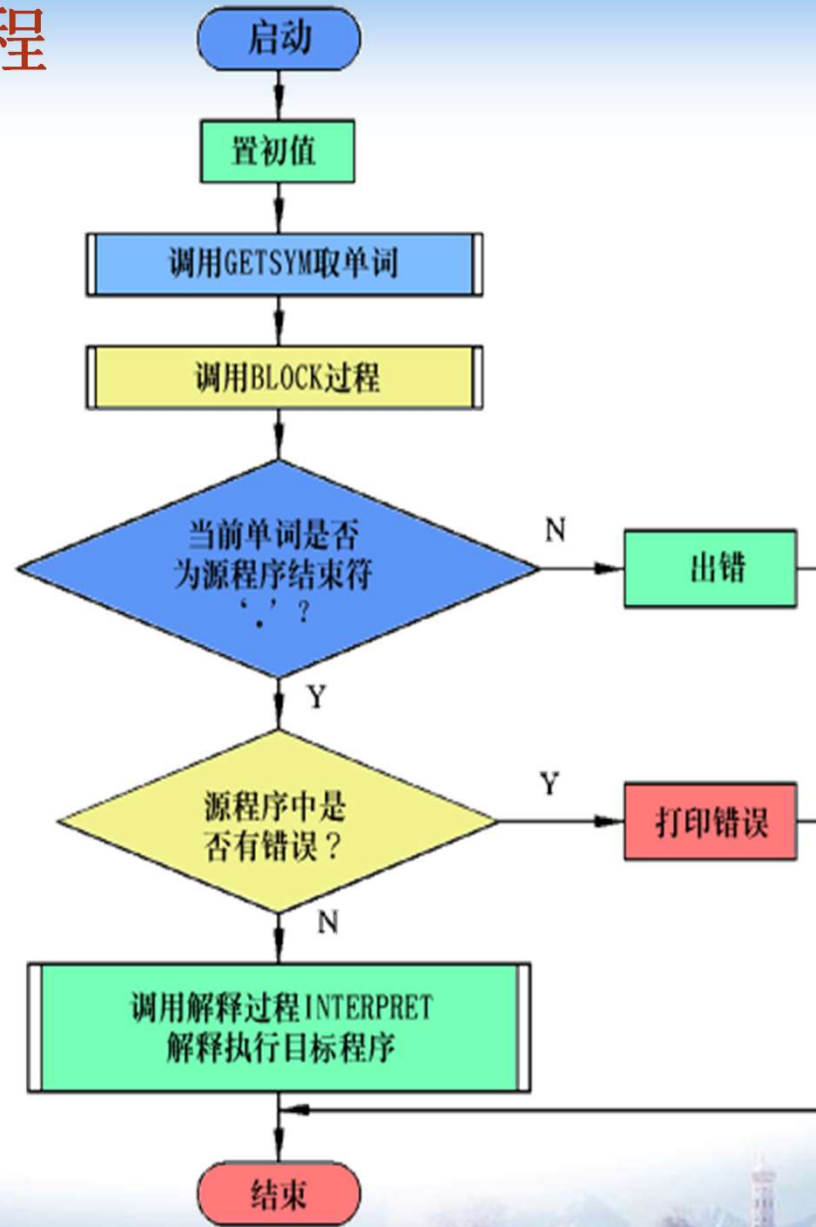
PL/0编译程序的

❖ PL/0编译程序由18个嵌套及并列的过程或函数组成，它们的层次结构如下。



PL/0编译程序总体流程

- ❖ 由于**PL/0**编译程序采用一趟扫描方法，所以语法分析过程**BLOCK**是整个编译过程的核心。
- ❖ **BLOCK**由当前单词根据语法规则再调用其它过程，如说明处理、代码生成或出错处理等过程进行分析。
- ❖ 当分析完一个单词后，**BLOCK**再调用**GETSYM**取下一个单词，一直重复到当前单词为结束符“.”，表明源程序已分析结束。



PL/0编译程序的总体设计



❖ 以语法、语义分析程序为核心

词法分析程序和代码生成程序都作为一个过程，当语法分析需要读单词时就调用词法分析程序，而当语法、语义分析正确，需要生成相应的目标代码时，则调用代码生成程序。

❖ 表格管理程序实现变量，常量和过程标识符的信息的登录与查找。

❖ 出错处理程序，对词法和语法、语义分析遇到的错误给出在源程序中出错的位置和与错误性质有关的编号，并进行错误恢复。

PL/0编译程序的词法分析



❖ 词法分析程序**GETSYM**

❖ 功能

- 识别单词作为语法分析的输入，单词分为关键字、运算符、界符、标识符和常数，其中前三类称为固有单词，后二类称为用户定义单词。

单词的种类

- ❖ 关键字（保留字）：**BEGIN、END、IF、THEN**等
- ❖ 运算符：如**+**、**-**、*****、**/**、**:**、**=**、**#**、**>=**、**<=**等
- ❖ 标识符：用户定义的变量名、常数名、过程名
- ❖ 常数：如**10**、**25**、**100**等整数
- ❖ 界符：如 **'**、**,**、**'**、**:**、**'**、**;**、**'**、**(**、**'**、**)**等

PL/0编译程序的单词种类



- ❖ 全部单词种类由编译程序定义的纯量类型**symbol**给出，也可称为**语法的词汇表**。（书P396）
- ❖ PL/0编译程序中开始对**类型**的定义中给出“**单词定义**”为：

```
enum symbol{  
    nul, ident, number, plus, minus,  
    times, slash, oddsym, eql, neq,  
    lss, leq, gtr, geq, lparen,  
    rparen, comma, semicolon, period, becomes,  
    beginsym, endsym, ifsym, thenym, whilesym,  
    writesym, readsym, dosym, callsym, constsym,  
    varsym, procsym,  
};  
#define symnum 32
```


PL/0编译程序的词法分析



词法分析过程GETSYM所要完成的任务：

- 从源程序读字符 (getch)
- 滤空格
- 识别保留字
- 识别标识符
- 拼数
- 识别单字符单词
- 拼双字符单词
- 输出源程序

重要变量及过程



❖ 符号表

```
type symbol=( nul, ident, number, plus, ...,  
              varsym, procsym );
```

❖ 保留字表:

```
word[1]:='BEGIN';
```

```
word[2]:='CALL';
```

...

```
word[13]:='WRITE';
```

查到时找到相应的内部表示

```
wsym[1]:=beginsym;
```

```
wsym[2]:=callsym;
```

...

```
wsym[13]:=writesym;
```

注意: 按ASCII顺序存储保留字。

❖ 单字符表:

```
ssym['+']:='plus'; ssym['-']:='minus';
```

...

```
ssym[';']:='semicolon';
```



重要变量及过程



❖ 全局变量

1) **SYM**: 存放单词的类别

如: 有程序段落为:

```
begin initial := 60; end
```

对应单词翻译后变为:

begin	beginsym,	initial	ident,
':= '	becomes,	60	number,
'; '	semicolon,	end	endsym .

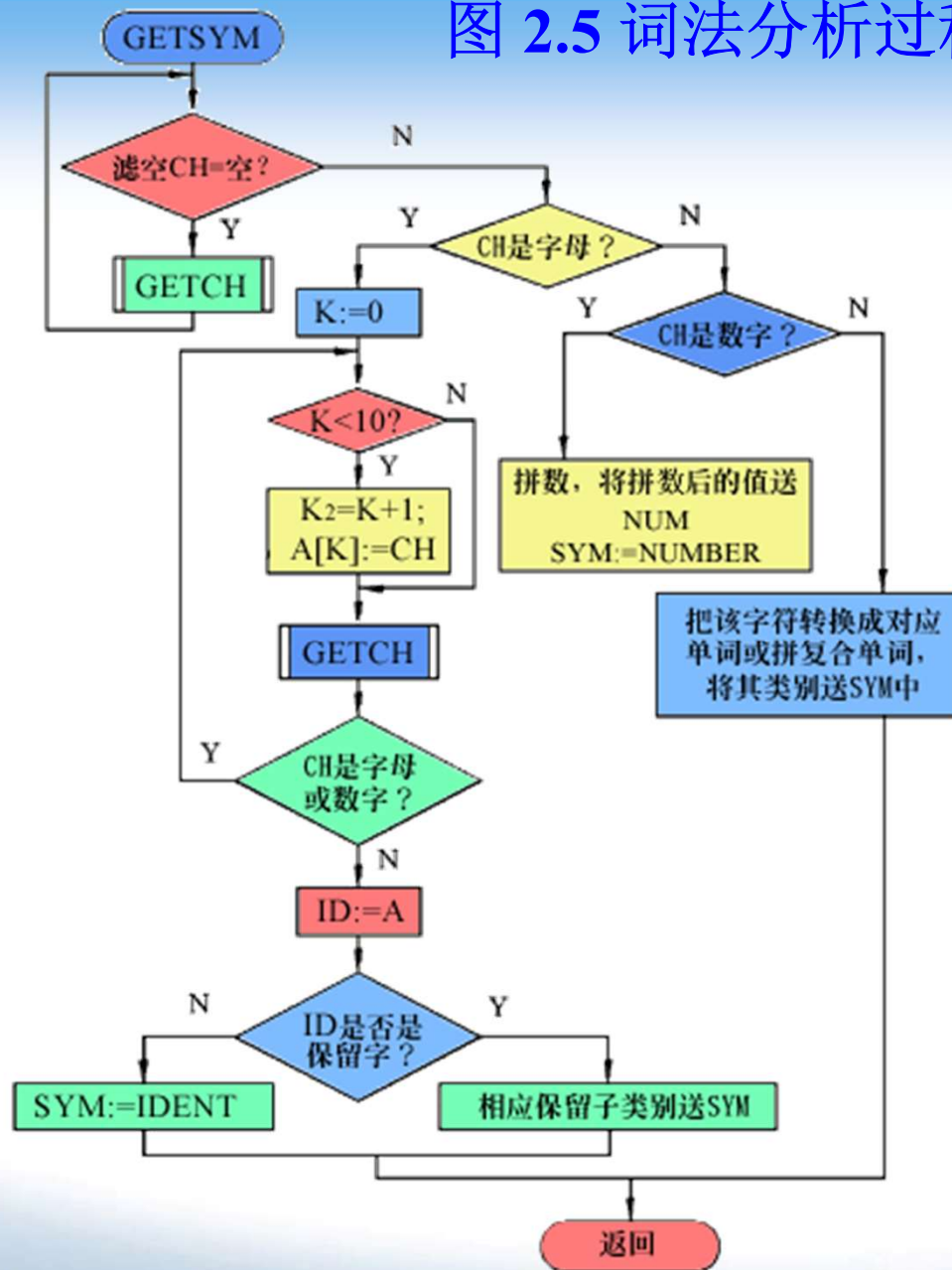
2) **ID**: 存放用户所定义的标识符的值 如: initial
(在**SYM**中放ident, 在**ID**中放initial)

3) **NUM**: 存放用户定义的数 如: 60
(在**SYM**中放在number在**NUM**中放60)

❖ GETSYM框图 (见后面)

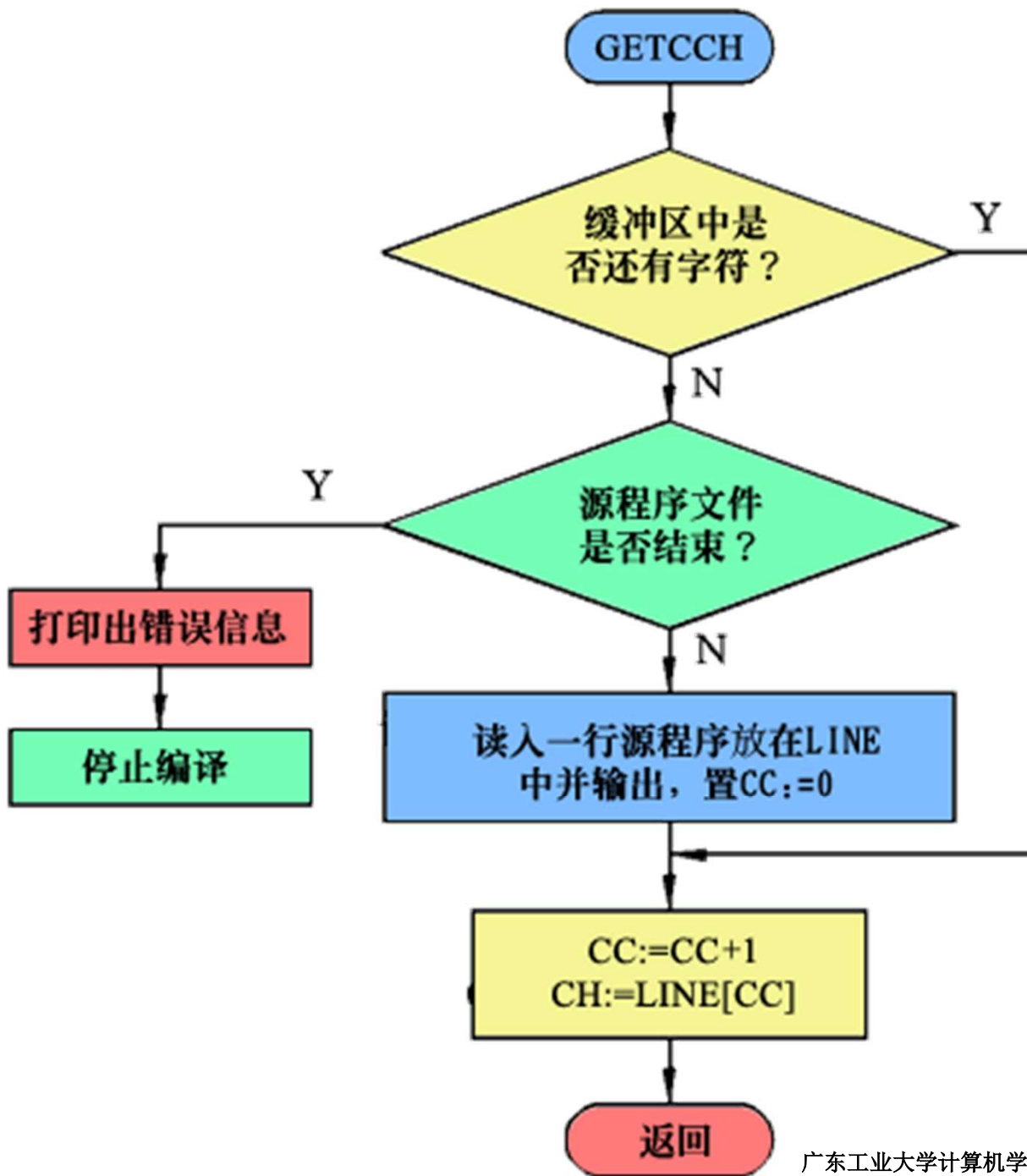


图 2.5 词法分析过程GETSYM



取字符过程

- ❖ **GETCH** 所用单元说明：
- ❖ **CH**：存放当前读取的字符，初值为空。
- ❖ **LINE**：为一维数组，其数组元素是字符，界对为**1:80**。用于读入一行字符的缓冲区。
- ❖ **LL**和**CC**为计数器，初值为0



词法分析总结



❖ 修改**PL/0**的词法分析需要修改的几个地方

一、整体细节的修改

- **symbol**的内容及**symbol**个数（**#define symnum 32**）
- 如果新增的是关键字，则修改**word**和**wsym.** 修改保留字数目**NORW**
- 如果新增的是单符号，则修改**ssym.**

二、**getsym()**函数内部的修改

- 修改**getsym**函数相应的地方，举例修改**++**

任务一：读程序



❖ 内容

■ 读程序GetSym()

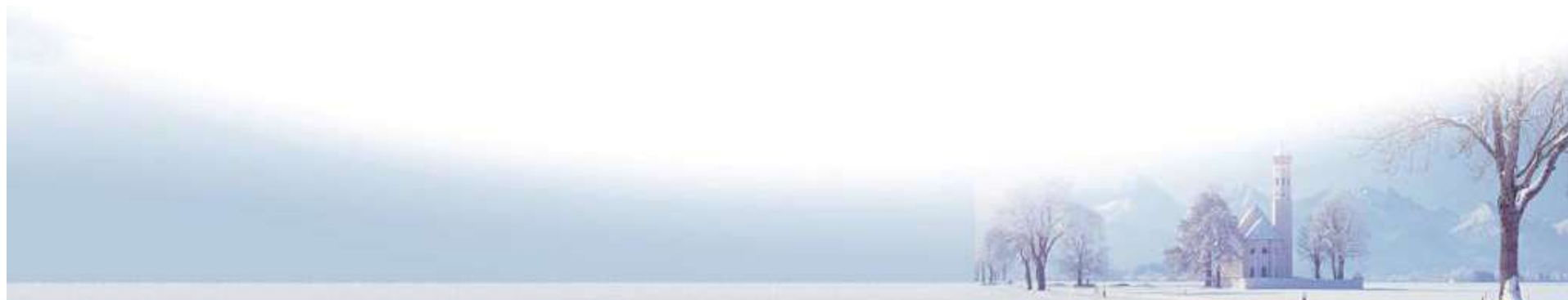
识别保留字

识别标识符

拼数

拼双字符单词

识别单字符单词



***= 、 /= 词法分析**



1.在PL/0中为其定义一个操作符名

```
char *SYMOUT[] = {"NUL", "IDENT", "NUMBER", "PLUS", "MINUS", "TIMES",  
                  "SLASH", "ODDSYM", "EQL", "NEQ", "LSS", "LEQ", "GTR", "GEQ",  
                  "LPAREN", "RPAREN", "COMMA", "SEMICOLON", "PERIOD",  
                  "BECOMES", "BEGINSYM", "ENDSYM", "IFSYM", "THENSYM",  
                  "WHILESYM", "WRITESYM", "READSYM", "DOSYM", "CALLSYM",  
                  "CONSTSYM", "VARSYM", "PROCSYM", "PROGSYM" };
```

2.在getsym中为双操作符拼词

```
else  
    if (CH==' :') {  
        GetCh();  
        if (CH==' =') { SYM=BECOMES; GetCh(); }  
        else SYM=NUL;  
    }
```

3. symbol的内容及symbol个数修改

任务二：扩充单词



❖ 内容

- 增加保留字：FOR、DOWNTO和TO
- 增加双字符单词：*=和/=

❖ 要求

- 设计测试方式，测试单词是否能被识别
- 上交作业到QQ群



5. 语法分析

—递归子程序法



对应**每个非终结符**语法单元，编一个独立的处理过程（或子程序）。

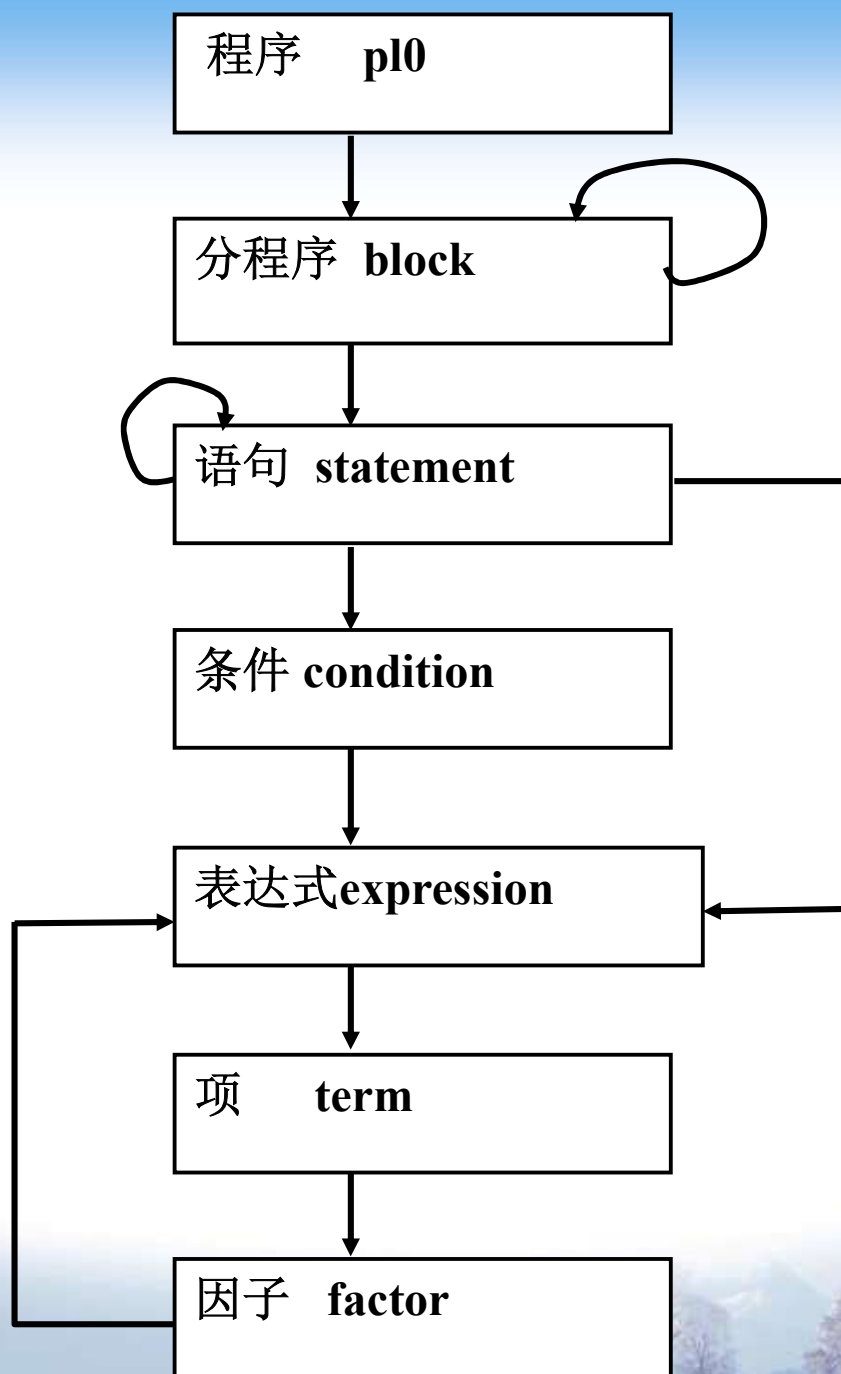
沿语法描述图**箭头**所指出的方向进行分析：

1) 遇到**分支点**时，将当前的单词与分支点上多个终结符**逐个相比较**，若都不匹配时可能是进入下一个非终结符语法单位或是出错。

2) 当遇到**非终结符**时，则**调用**相应的**处理过程**；

3) 当遇到描述图中是**终结符**时，则判断当前读入的单词是否与图中的终结符**相匹配**，若匹配，再读取下一个单词继续分析。

语法调用关系图



PL/O编译程序的语法错误处理



1. PL/O编译程序对语法错误的两种处理方法:

➤对于易于校正的错误，如丢了逗号、分号等，则指出出错位置，并加以校正。校正的方式就是补上逗号或分号

➤对于难于校正的错误，跳过一些后面输入的单词符号，



直到读入一个能使编译程序恢复正常语法分析工作的单词为止。

具体做法是：当语法分析进入或退出一个语法

单元的处理程序时，调用一个测试程序TEST，其流程图

如2.9所示：它的功能是检查当前单词是否属于该语法单

元的开始符号集合或后跟符号集合

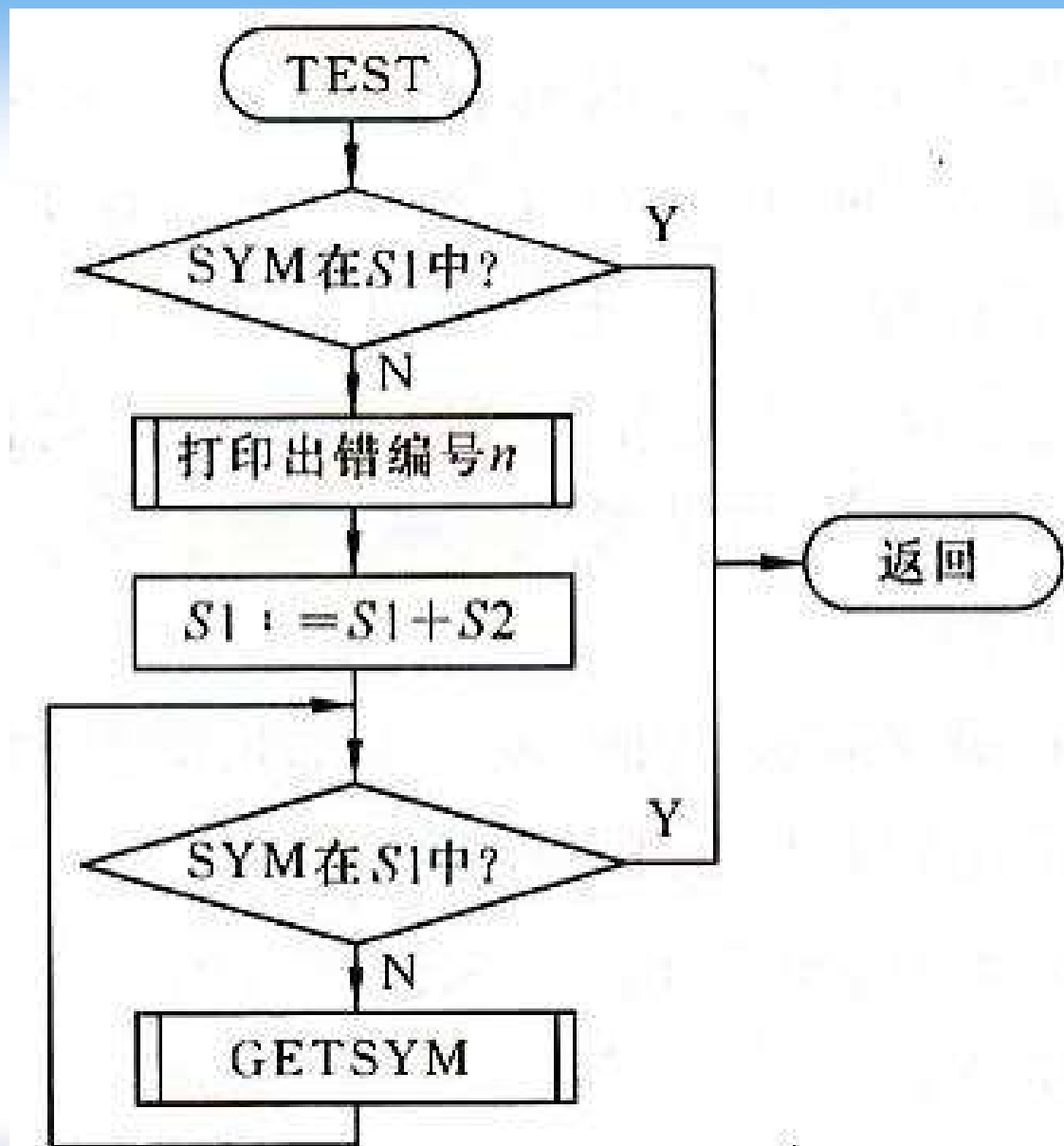


图 2.9 TEST 测试过程流程图

➤表 2.3 PL/0文法非终结符的开始符号与后继符号集合表

非终结符名	开始符号集合	后继符号集合
分程序	const var procedure ident if call begin while read write	. ;
语句	ident call begin if while read write	. ; end
条件	odd + - (ident number	then do
表达式	+ - (ident number	. ;) rop end then do
项	ident number (. ;) rop + - end then do
因子	ident number (. ; + - * / end then do

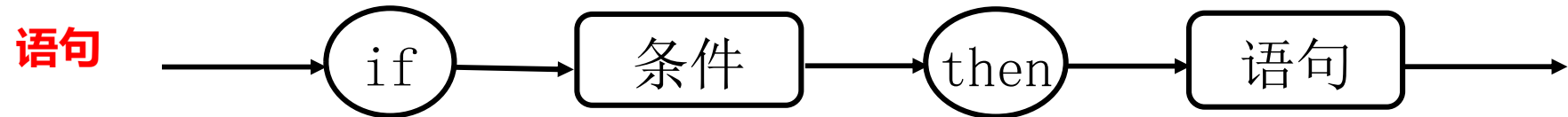
*注： 表2.3中'rop'表示关系运算符集合,如=, #, <, <=, >, >=。



❖TEST测试过程三个参数的含义:

- ①S1 当语法分析进入或退出某一语法单元时当前单词符号应属于的集合，它可能是一个语法单元开始符号的集合或后继符号的集合
- ②S2 在某一出错状态时，可恢复语法分析继续正常工作的补充单词符号集合
- ③n 整型数，出错信息编号

例：条件语句（不带else子句）的 递归子程序



```
statement() {  
    if( SYM == ifsym ) { //处理条件语句  
        getSym();  
        condition();  
        if( SYM == thensym ) getSym();  
        else Error();  
        statement();  
    } else { ..... } //处理其它语句  
}
```

任务三



❖ 扩充条件语句（带else子句）

- 画出语法描述图
- 写出递归子程序



PL/0编译程序的语义分析



- 借助符号表进行上下文相关的静态语义分析
- 确保符号表可以体现作用域规则
- 确保标识符属性与上下文环境一致
- 确保标识符先声明后引用
- 确保标识符的长度、数字的位数、过程嵌套说明的层数符合PL/0语言的约定
- 提示语义错误信息
(其实还包括目标代码的生成!)

PL/0 编译程序的语义分析



✧ 符号表结构

```
Enum object {  
    constant,  
    variable,  
    procedur  
};
```

```
Struct tablestruct {  
    char name[a1];      /*a1-名字最大长度*/  
    enum object kind;  
    int val;            /*constant标识符的数值*/  
    int level;          /*标识符所在的层, constant标识符不用*/  
    int adr;            /*标识符相对地址, constant标识符不用*/  
    int size;          /*需分配的数据区大小, 仅procedur标识符用到*/  
};
```

```
Struct tablestruct table[txmax];  /*txmax-符号表容量*/
```

PL/0 编译程序的语义分析



✧ 符号表管理

- 登录(在符号表中插入一项)

```
void enter(enum object k, int* ptx, int lev, int* pdx)
```

```
/* k: 名字种类 const,var or procedure
```

```
   ptx: 名字表尾指针的指针
```

```
   lev: 名字所在的层次
```

```
   pdx: 当前应分配变量的相对地址，分配后增加1
```

```
*/
```

- 查询

```
int position(char * idt, int tx)
```

```
/* idt: 被查标识符名字串
```

```
   tx: 符号表栈当前栈顶的位置
```

```
返回所查标识符在符号表栈中的位置，没查到则返回0*/
```

PL/0 编译程序的语义分析



✧ 语义处理举例 —— 变量声明语句的处理

<变量说明部分> ::= var <标识符> {, <标识符>;}

```
if (sym == varsym){          /* 收到变量声明符号, 开始处理变量声明 */
    getsymdo;                /*调用 getsym ( )的宏*/
    do {
        vardeclarationdo(&tx, lev, &dx);      /*见下页*/
        while (sym == comma) {
            getsymdo;
            vardeclarationdo(&tx, lev, &dx);
        }
        if (sym == semicolon){
            getsymdo;
        }else error(5);
    } while (sym == ident);
}
```

PL/0 编译程序的语义分析



✧ 语义处理举例 —— 变量声明语句的处理 (接上页)

```
int vardeclaration(int* ptx,int lev,int* pdx)

    /* ptx: 符号表尾位置; lev: 当前层; pdx: 在当前层的偏移量; */

    {
        if (sym == ident){
            enter(variable, ptx, lev, pdx); //填写名字表
            getsymdo;
        } else
        {
            error(4);    /* var 后应是标识符 */
        }
        return 0;
    }
```

PL/0 编译程序的语义分析



✧ 语义处理举例 — 变量引用的处理

- 当遇到标识符的引用时就调用POSITION函数查符号表，看是否有过正确定义，若已有，则从表中取相应的有关信息，供代码的生成使用；若无定义则报错
- 若在符号表有过正确定义，检查引用与说明的属性是否一致，若不一致则报错

PL/0 编译程序的语义分析



☆ 语义处理举例 — 变量引用的处理 (以赋值语句中的变量引用为例)

```
if (sym == ident){ /* 准备按照赋值语句处理 */
    i = position(id, *ptx);
    if (i == 0){
        error(11); /* 变量未找到 */
    }else
    {
        if (table[i].kind != variable) {
            error(12); /* 赋值语句格式错误 */
            i = 0;
        }else
        {
            .....
            gendo(sto, ...); /*生成目标代码*/
            .....
        }
    }
}
```



2.PL/O编译程序对语义错误

如标识符未加说明就引用，或虽经说明，但引用与说明的属性不一致。这时只给出错误信息和出错的位置，编译工作可继续进行



3.PL/O编译程序对运行错

如溢出、越界等，只能在运行时发现，由于PL/O编译程序的功能限制无法指出运行时所发生的错误在源程序的对应位置



4.PL/O语言的出错信息表:

出错编号

出错原因

- 1: 常数说明中的“=”写成“:=”。
- 2: 常数说明中的“=”后应是数字。
- 3: 常数说明中的标识符后应是“=”。
- 4: **const ,var, procedure**后应为标识符。
- 5: 漏掉了‘,’或‘;’。
- 6: 过程说明后的符号不正确(应是语句开始符, 或过程定义符)。
- 7: 应是语句开始符。
- 8: 程序体内语句部分的后跟符不正确。



出错编号

出错原因

- 9: 程序结尾丢了句号 ‘.’。
- 10: 语句之间漏了 ‘; ’。
- 11: 标识符未说明。
- 12: 赋值语句中，赋值号左部标识符属性应是变量。
- 13: 赋值语句左部标识符后应是赋值号 ‘:=’。
- 14: **call**后应为标识符。
- 15: **call**后标识符属性应为过程。
- 16: 条件语句中丢了 ‘**then**’。
- 17: 丢了 ‘**end**’或’ ; ’。
- 18: **while**型循环语句中丢了'**do**'。



出错编号

出错原因

- 19: 语句后的符号不正确。
- 20: 应为关系运算符。
- 21: 表达式内标识符属性不能是过程。
- 22: 表达式中漏掉右括号 ‘)’。
- 23: 因子后的非法符号。
- 24: 表达式的开始符不能是此符号。
- 31: 数越界。
- 32: **read**语句括号中的标识符不是变量。

PL/O编译程序的目标代码结构和代码生成



❖ PL/O编译程序所产生的目标代码是一个假想栈式计算机的汇编语言，可称为类PCODE指令代码，它不依赖任何实际计算机

❖ 指令格式如下：

f	l	a
---	---	---

f 功能码

l 层次差

a 根据不同的指令有所区别

❖ 八条目标指令：（a为其他值均为非法指令）

❖ 类pcode代码是由过程GEN生成的，放在数组CODE中。

pcode指令代码



- ❖ 类**PCODE**指令代码不依赖任何具体计算机，其指令集极为简单，其指令格式如下：



- ❖ **f**：功能码。
- ❖ **l**：层次差，即变量或过程被引用的分程序与说明该变量或过程的分程序之间的层次差。
- ❖ **a**的含意则对不同的指令有所区别。
- ❖ 目标指令有**8**条：
- ❖ ① **LIT**：将常量值取到运行栈顶。**a**域为常数值。
- ❖ ② **LOD**：将变量放到栈顶。**a**域为变量在所说明层中的相对位置，**l**为调用层与说明层的层差值。

pcode指令代码(续)



- ❖ ③ **STO**: 将栈顶的内容送入某变量单元中。**a**和**I**域的含意同**LOD**指令。
- ❖ ④ **CAL**: 调用过程的指令。**a**为被调用过程的目标程序入口地址，**I**为层差。
- ❖ ⑤ **INT**: 为被调用的过程(或主程序)在运行栈中开辟数据区。**a**域为开辟的单元个数。
- ❖ ⑥ **JMP**: 无条件转移指令，**a**为转向地址。
- ❖ ⑦ **JPC**: 条件转移指令，当栈顶的布尔值为非真时，转向**a**域的地址，否则顺序执行。
- ❖ ⑧ **OPR**: 关系运算和算术运算指令。将栈顶和次栈顶的内容进行运算，结果存放在次栈顶。此外还可以是读写等特殊功能的指令，具体操作由**a**域值给出。（**P23**）

指令功能表

LIT 0 a	将常数值取到栈顶，a为常数值
LOD 1 a	将变量值取到栈顶，a为偏移量，1为层差
STO 1 a	将栈顶内容送入某变量单元中，a为偏移量，1为层差
CAL 1 a	调用过程，a为过程地址，1为层差
INT 0 a	在运行栈中为被调用的过程开辟a个单元的数据区
JMP 0 a	无条件跳转至a地址
JPC 0 a	条件跳转，当栈顶布尔值非真则跳转至a地址，否则顺序执行
OPR 0 0	过程调用结束后，返回调用点并退栈
OPR 0 1	栈顶元素取反
OPR 0 2	次栈顶与栈顶相加，退两个栈元素，结果值进栈
OPR 0 3	次栈顶减去栈顶，退两个栈元素，结果值进栈
OPR 0 4	次栈顶乘以栈顶，退两个栈元素，结果值进栈
OPR 0 5	次栈顶除以栈顶，退两个栈元素，结果值进栈
OPR 0 6	栈顶元素的奇偶判断，结果值在栈顶
OPR 0 7	
OPR 0 8	次栈顶与栈顶是否相等，退两个栈元素，结果值进栈
OPR 0 9	次栈顶与栈顶是否不等，退两个栈元素，结果值进栈
OPR 0 10	次栈顶是否小于栈顶，退两个栈元素，结果值进栈
OPR 0 11	次栈顶是否大于等于栈顶，退两个栈元素，结果值进栈
OPR 0 12	次栈顶是否大于栈顶，退两个栈元素，结果值进栈
OPR 0 13	次栈顶是否小于等于栈顶，退两个栈元素，结果值进栈
OPR 0 14	栈顶值输出至屏幕
OPR 0 15	屏幕输出换行
OPR 0 16	从命令行读入一个输入置于栈顶



■编译程序的目标代码生成



- ❖在分析程序体时生成的
- ❖当分析程序体中的每个语句时，当语法正确则调用目标代码生成过程以生成与PL/0语句等价功能的目标代码，直到编译正常结束
- ❖由过程GEND0完成，**GEN**有三个参数，分别代表目标代码的功能码，层差和位移量
 - ❖例如：`gen(opr,0,16)` 从命令行读入一个输入到栈顶
- ❖生成的代码顺序放在数组**CODE**中
 - ❖**CODE**为一维数组，数组元素为记录型数据，每一个记录就是一条目标指令
 - ❖**CX**为指令的指针，由0开始顺序增加

PL/0 编译程序的目标代码生成



✧ 目标代码生成函数

```
#define gendo(a, b, c) if (-1 == gen(a, b, c)) return -1
```

```
.....
```

```
int gen(enum fct x, int y, int z )
```

```
{
```

```
    if (cx >= cxmax)
```

```
    {
```

```
        printf("Program too long");
```

```
        return -1;
```

```
    }
```

```
    code[cx].f = x;
```

```
    code[cx].l = y;
```

```
    code[cx].a = z;
```

```
    cx++;
```

```
    return 0;
```

```
}
```

PL/0 编译程序的目标代码生成



✧ 例

```
int statement(bool* fsys, int* ptx, int lev)
{
    ...
    if (sym == ident)    /* 准备按照赋值语句处理 */
    {
        i = position(id, *ptx);
        ...
        gendo(sto, lev-table[i].level, table[i].adr);
    }
    ...
}
```

PL/0 编译程序的目标代码生成



✧ 跳转指令与地址返填

```
if (sym == ifsym)          /* 准备按照 if c then s 语句处理 */
{
    getsymdo;
    conditiondo(...);      /* 调用条件处理（逻辑运算）函数 */
    if (sym == thensym)
        getsymdo;
    else
        error(16);         /* 缺少 then */
    cx1 = cx;               /* 保存当前指令地址 */
    gendo(jpc, 0, 0);       /* 生成条件跳转指令，跳转地址未知，暂时写0 */
    statementdo(fsys, ptx, lev); /* 处理 then 后的语句 */
    code[cx1].a = cx;       /* 地址返填 */
}
```

PL/0 编译程序的目标代码生成



(0) jmp 0 8	转向主程序入口
(1) jmp 0 2	转向过程p入口
(2) int 0 3	过程p入口,为过程p开辟空间
(3) lod 1 3	取变量b的值到栈顶
(4) lit 0 10	取常数10到栈顶
(5) opr 0 2	次栈顶与栈顶相加
(6) sto 1 4	栈顶值送变量c中
(7) opr 0 0	退栈并返回调用点(16)
(8) int 0 5	主程序入口开辟5个栈空间
(9) opr 0 16	从命令行读入值置于栈顶
(10) sto 0 3	将栈顶值存入变量b中
(11) lod 0 3	将变量b的值取至栈顶
(12) lit 0 0	将常数值0进栈
(13) opr 0 9	次栈顶与栈顶是否不等
(14) jpc 0 24	相等时转(24) (条件不满足转)
(15) cal 0 2	调用过程p
(16) lit 0 2	常数值2进栈
(17) lod 0 4	将变量c的值取至栈顶
(18) opr 0 4	次栈顶与栈顶相乘(2*c)
(19) opr 0 14	栈顶值输出至屏幕
(20) opr 0 15	换行
(21) opr 0 16	从命令行读取值到栈顶
(22) sto 0 3	栈顶值送变量b中
(23) jmp 0 11	无条件转到循环入口(11)
(24) opr 0 0	结束退栈

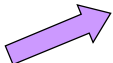


◇ 例

```
const a=10;
var  b,c;
procedure p;
begin
    c:=b+a;
end;
begin
    read(b);
    while b#0 do
    begin
        call p;
        write(2*c);
        read(b);
    end
end.
```

条件语句的语义分析和处理



〈条件语句〉 ::= if 〈条件〉 then 〈语句〉

```
                                (a > b)                                write(a)
if( SYM == ident ) { //处理条件语句
    getSym();                                lod 0 a
    condition();  lod 0 b
    if(SYM == thensym)  opr 0 12 getSym();
    else Error();                                lod 0 a
    statement();  opr 0 14
} else { ..... } //处理其它语句                                opr 0 15
```

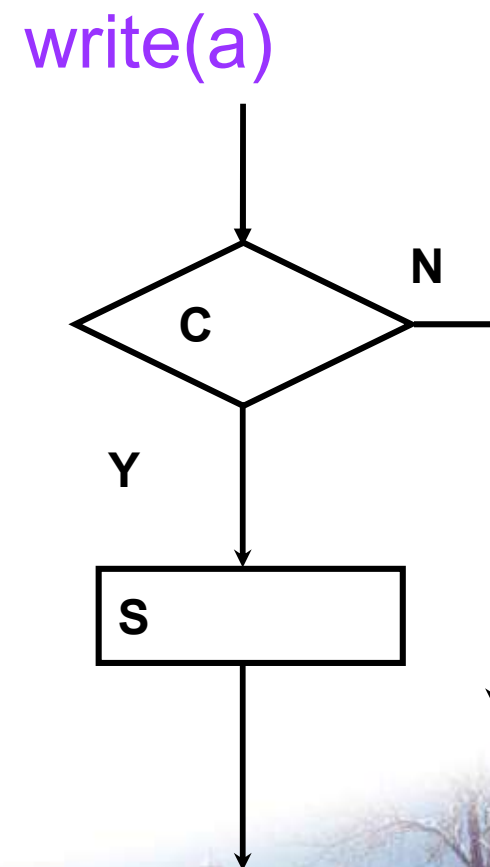

条件语句的语义分析和处理



〈条件语句〉 ::= if 〈条件〉 then 〈语句〉

(a > b)

```
if( SYM == ident ) { //处理条件语句
    getSym();          lod 0 a      lod 0 a
    condition();        lod 0 b      lod 0 b
                        opr 0 12
    if(SYM == thensym)  getSym();    jpc 0 0
    else Error();      lod 0 a      lod 0 a
    statement();        opr 0 14     opr 0 14
} else {.....} //处理其它语句    opr 0 15
```



条件语句的语义分析和处理

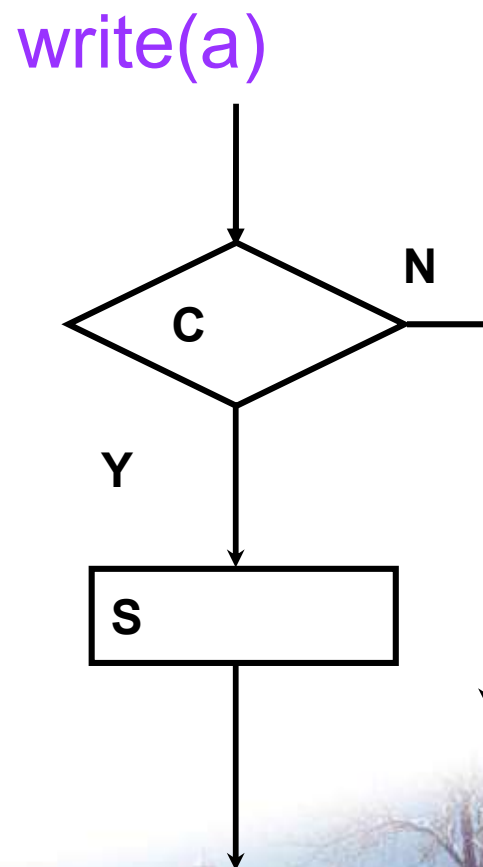


〈条件语句〉 ::= if 〈条件〉 then 〈语句〉

(a > b)

```
if( SYM == ident ) { //处理条件语句
    getSym();
    condition();      GEN(jpc,0,0)
    if(SYM == thensym) getSym();
    else Error();
    statement();
} else {.....} //处理其它语句
```

lod 0 a
lod 0 b
opr 0 12
jpc 0 0
lod 0 a
opr 0 14
opr 0 15



条件语句的语义分析和处理



〈条件语句〉 ::= if 〈条件〉 then 〈语句〉

(a > b)

```
if( SYM == ident ) { //处理条件语句
```

```
    getSym();
```

```
    condition(); cx1 = cx; GEN(jpc, 0, 0)
```

```
    if( SYM == thensym ) getSym();
```

```
    else Error();
```

```
    statement(); code[cx1].a = cx
```

```
} else { ..... } //处理其它语句
```

write(a)

lod 0 a

lod 0 b

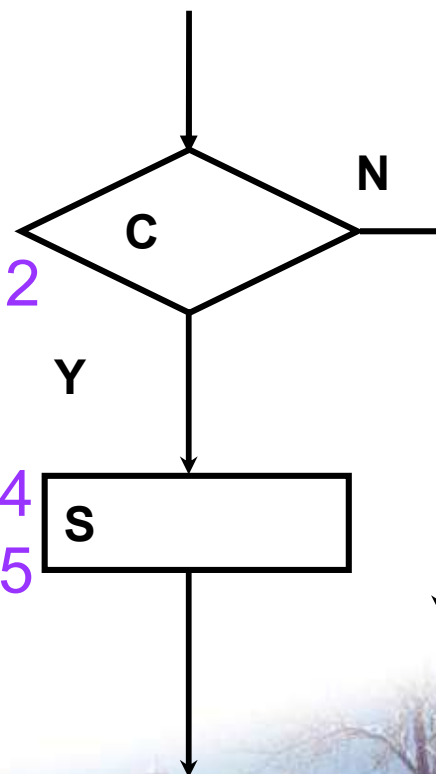
opr 0 12

jpc 0 0

lod 0 a

opr 0 14

opr 0 15



PL/0 编译程序的语法分析修改举例



❖ if....then...else

❖ 1.修改相应的词法分析部分

❖ 2.修改语法分析部分，即**int statement(bool * fsys,int *ptx,int lev)**函数部分。

- 修改的部分为**if(sym==ifsym) //如果是if语句**
- 修改思路按第8章，主要问题就是指令跳转地址的填写

任务四



❖ 扩充条件语句（带else子句）

- 设计出以下条件语句的pcode代码

if $a > b$

then write(a);

else write(b);

(假设变量a和b的偏移地址是3和4)

- 写出条件语句的语义分析程序



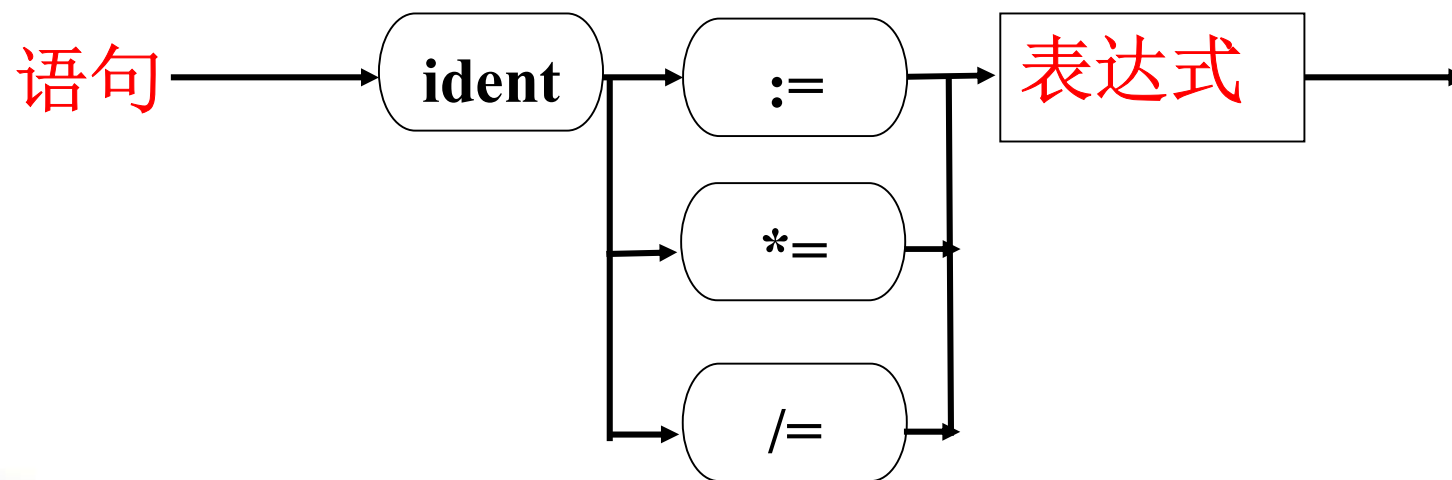


❖ 接下来简单介绍课程设计部分！

CP课程设计之 $\ast=$ 、 $/=$ 语法分析



1. 画出语法描述图



CP课程设计之 *= 、 /= 语法分析



2.在unit1.cpp中添加代码

❖ 添加位置: **block**→**statement**→**'case IDENT'**

```
void STATEMENT(SYMSET FSYS,int LEV,int &TX) { /*STATEMENT*/
    int i,CX1,CX2;
    switch (SYM) {
        case IDENT:
            i=POSITION(ID,TX);
            if (i==0) Error(11);
            else
                if (TABLE[i].KIND!=VARIABLE) { /*ASSIGNMENT TO NON-VARIABLE*/
                    Error(12); i=0;
                }
            GetSym();
            if (SYM==BECOMES) GetSym();
            else Error(13);
            EXPRESSION(FSYS,LEV,TX);
            if (i!=0) GEN(STO,LEV-TABLE[i].vp.LEVEL, TABLE[i].vp.ADR);
            break;
    }
```

替换成*=、/=

CP课程设计之 *= 、 /=语义分析



位置: **block**→**statement**→**'case IDENT'**

1.检查**ident**是否已正确定义

2.生成目标代码

```
void STATEMENT(SYMSET FSYS,int LEV,int &TX) { /*STATEMENT*/
    int i,CX1,CX2;
    switch (SYM) {
        case IDENT:
            i=POSITION(ID,TX);
            if (i==0) Error(11);
            else
                if (TABLE[i].KIND!=VARIABLE) { /*ASSIGNMENT TO NON-VARIABLE*/
                    Error(12); i=0;
                }
            GetSym();
            if (SYM==BECOMES) GetSym();
            else Error(13);
            EXPRESSION(FSYS,LEV,TX);
            if (i!=0) GEN(STO,LEV-TABLE[i].vp.LEVEL, TABLE[i].vp.ADR);
            break;
    }
}
```

替换成*=、/=运算对应的目标代码

CP课程设计之 for语句-词法分析



⑩ 在PL/0中为其定义一个forsym关键字

```
const NORW = 14; /* # OF RESERVED WORDS */

typedef enum { NUL, IDENT, NUMBER, PLUS, MINUS, TIMES,
               SLASH, ODDSYM, EQL, NEQ, LSS, LEQ, GTR, GEQ,
               LPAREN, RPAREN, COMMA, SEMICOLON, PERIOD,
               BECOMES, BEGINSYM, ENDSYM, IFSYM, THENSYM,
               WHILESYM, WRITESYM, READSYM, DOSYM, CALLSYM,
               CONSTSYM, VARSYM, PROCSYM, PROGSYM
             } SYMBOL;

char *SYMOUT[] = {"NUL", "IDENT", "NUMBER", "PLUS",
                  "SLASH", "ODDSYM", "EQL", "NEQ", "LSS",
                  "LPAREN", "RPAREN", "COMMA", "SEMICOLON",
                  "BECOMES", "BEGINSYM", "ENDSYM", "IFSYM",
                  "WHILESYM", "WRITESYM", "READSYM", "DOS",
                  "CONSTSYM", "VARSYM", "PROCSYM", "PROGS"};

strcpy(KWORD[1], "BEGIN");      strcpy(KWORD[2], "CALL");
strcpy(KWORD[3], "CONST");      strcpy(KWORD[4], "DO");
strcpy(KWORD[5], "END");        strcpy(KWORD[6], "IF");
strcpy(KWORD[7], "ODD");        strcpy(KWORD[8], "PROCEDURE");
strcpy(KWORD[9], "PROGRAM");    strcpy(KWORD[10], "READ");
strcpy(KWORD[11], "THEN");      strcpy(KWORD[12], "VAR");
strcpy(KWORD[13], "WHILE");     strcpy(KWORD[14], "WRITE");

WSYM[1]=BEGINSYM;  WSYM[2]=CALLSYM;
WSYM[3]=CONSTSYM;  WSYM[4]=DOSYM;
WSYM[5]=ENDSYM;    WSYM[6]=IFSYM;
WSYM[7]=ODDSYM;    WSYM[8]=PROCSYM;
WSYM[9]=PROGSYM;   WSYM[10]=READSYM;
WSYM[11]=THENSYM;  WSYM[12]=VARSYM;
WSYM[13]=WHILESYM; WSYM[14]=WRITESYM;
SSYM['+']=PLUS;    SSYM['-']=MINUS;
SSYM['*']=TIMES;    SSYM['/']=SLASH;
SSYM['(']=LPAREN;   SSYM[')']=RPAREN;
SSYM['=']=EQL;       SSYM[',']=COMMA;
SSYM['.']=PERIOD;    SSYM['#']=NEQ;
SSYM[';']=SEMICOLON;
```

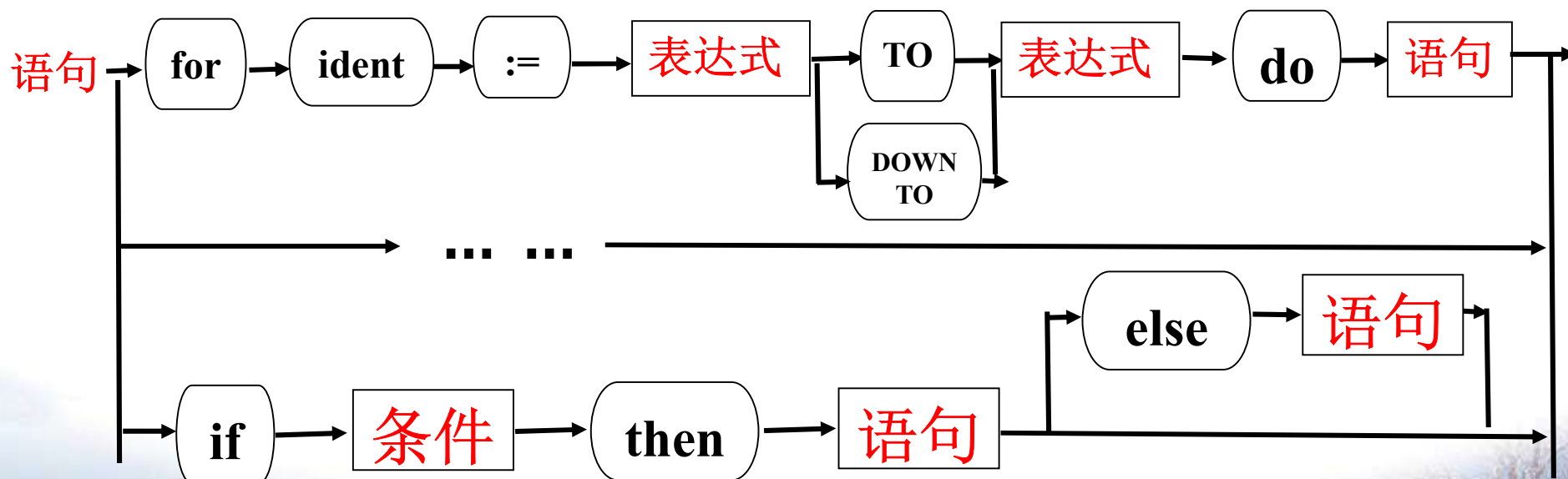
CP课程设计之 for语句-语法分析



1.画出语法描述图

①FOR <变量>:=<表达式> TO <表达式> DO <语句>

②FOR <变量>:=<表达式> DOWNTO <表达式> DO <语句>



2.在unit1.cpp中添加代码 (C语言)

❖ 添加位置: **block**→**statement**→**else if(sym==forsym)**

CP课程设计之 **for**语句-语义分析



位置: **block**→**statement**->**else if(sym==forsym)**

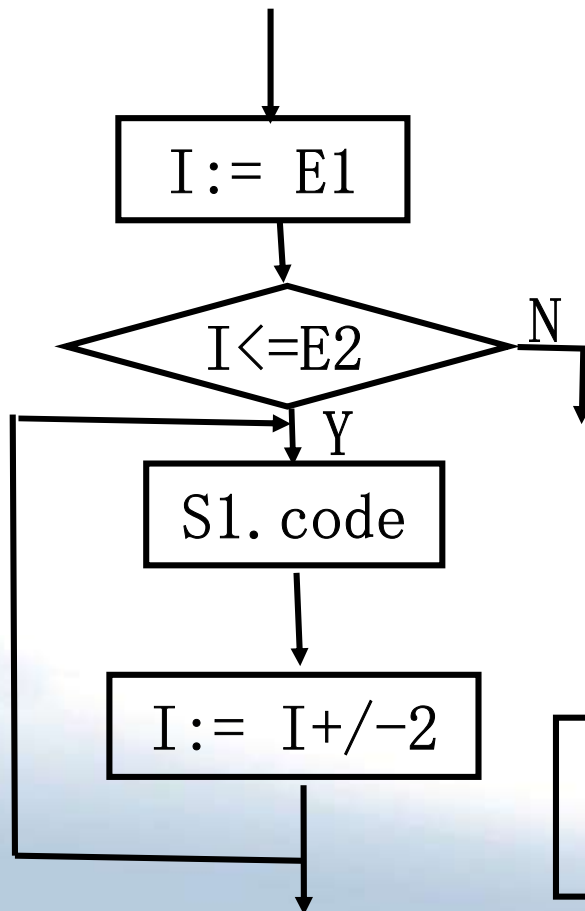
1.检查**ident**是否已正确定义

2.生成相应目标代码

CP课程设计之 **for**语句-语义分析



$S \rightarrow \text{FOR } I := E1 \text{ TO } E2 \text{ DO } S1$



代码片段示例:

```
READ(x);  
FOR i:=x to (x+10) DO  
  BEGIN  
    READ(y);  
    WRITE(y*i)  
  END
```

注意：E1和E2可以是数字、常量、变量或表达式！

CP课程设计之 ++ 、 -- 词法分析



1.在PL/0中为++、--定义一个操作符名

```
char *SYMOUT[] = {"NUL", "IDENT", "NUMBER", "PLUS", "MINUS", "TIMES",  
                  "SLASH", "ODDSYM", "EQL", "NEQ", "LSS", "LEQ", "GTR", "GEQ",  
                  "LPAREN", "RPAREN", "COMMA", "SEMICOLON", "PERIOD",  
                  "BECOMES", "BEGINSYM", "ENDSYM", "IFSYM", "THENSYM",  
                  "WHILESYM", "WRITESYM", "READSYM", "DOSYM", "CALLSYM",  
                  "CONSTSYM", "VARSYM", "PROCSYM", "PROGSYM"};
```

```
else  
    if (CH==':') {  
        GetCh();  
        if (CH=='=') { SYM=BECOMES; GetCh(); }  
        else SYM=NUL;  
    }
```




CP课程设计之 ++ 、 --语法分析

两种情况

- 单独作为一条语句出现

...

`a++;`

...

- 在表达式中出现

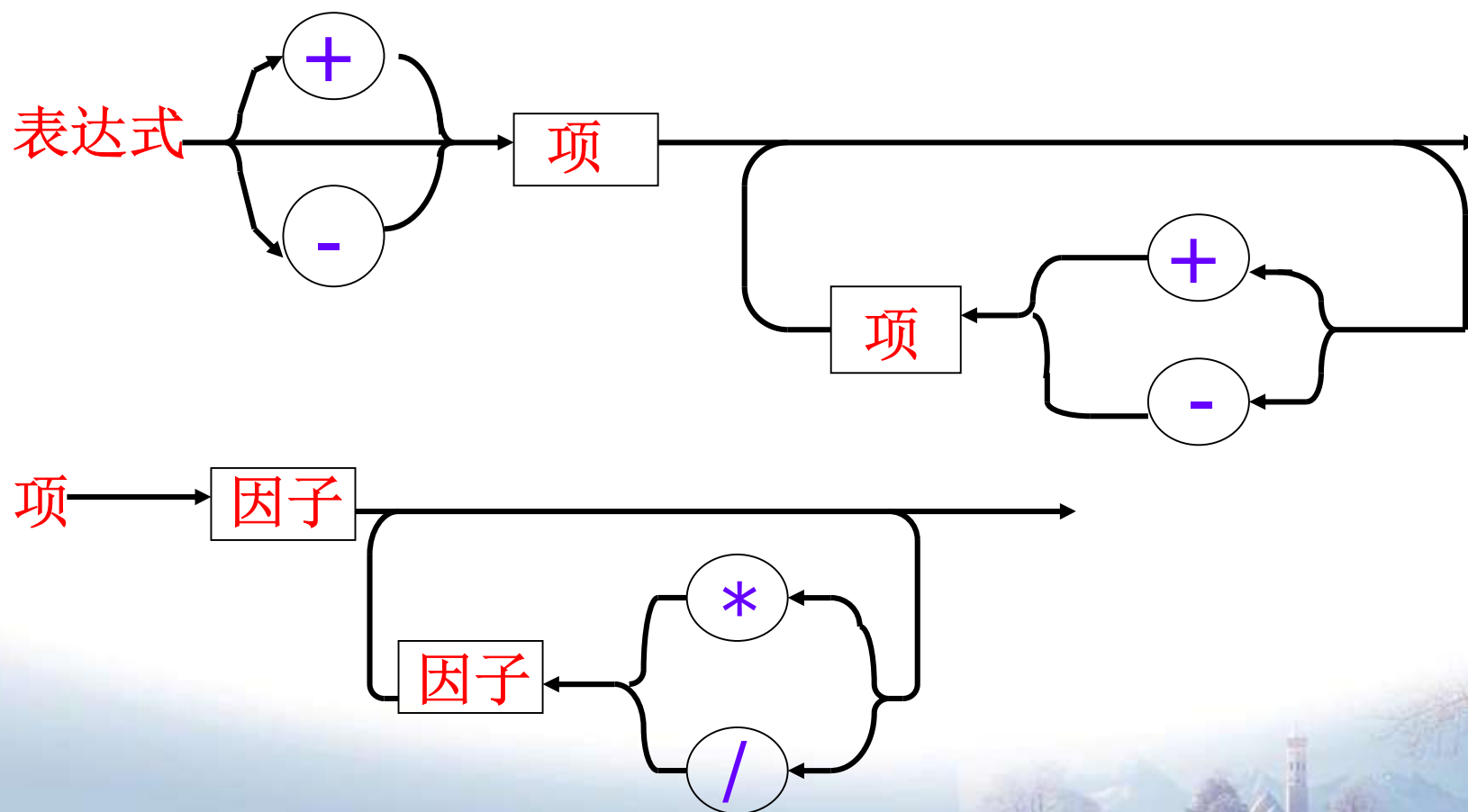
`s:=++a+b++;`

如何实现？

CP课程设计之 ++、-- 语法分析



语法描述图

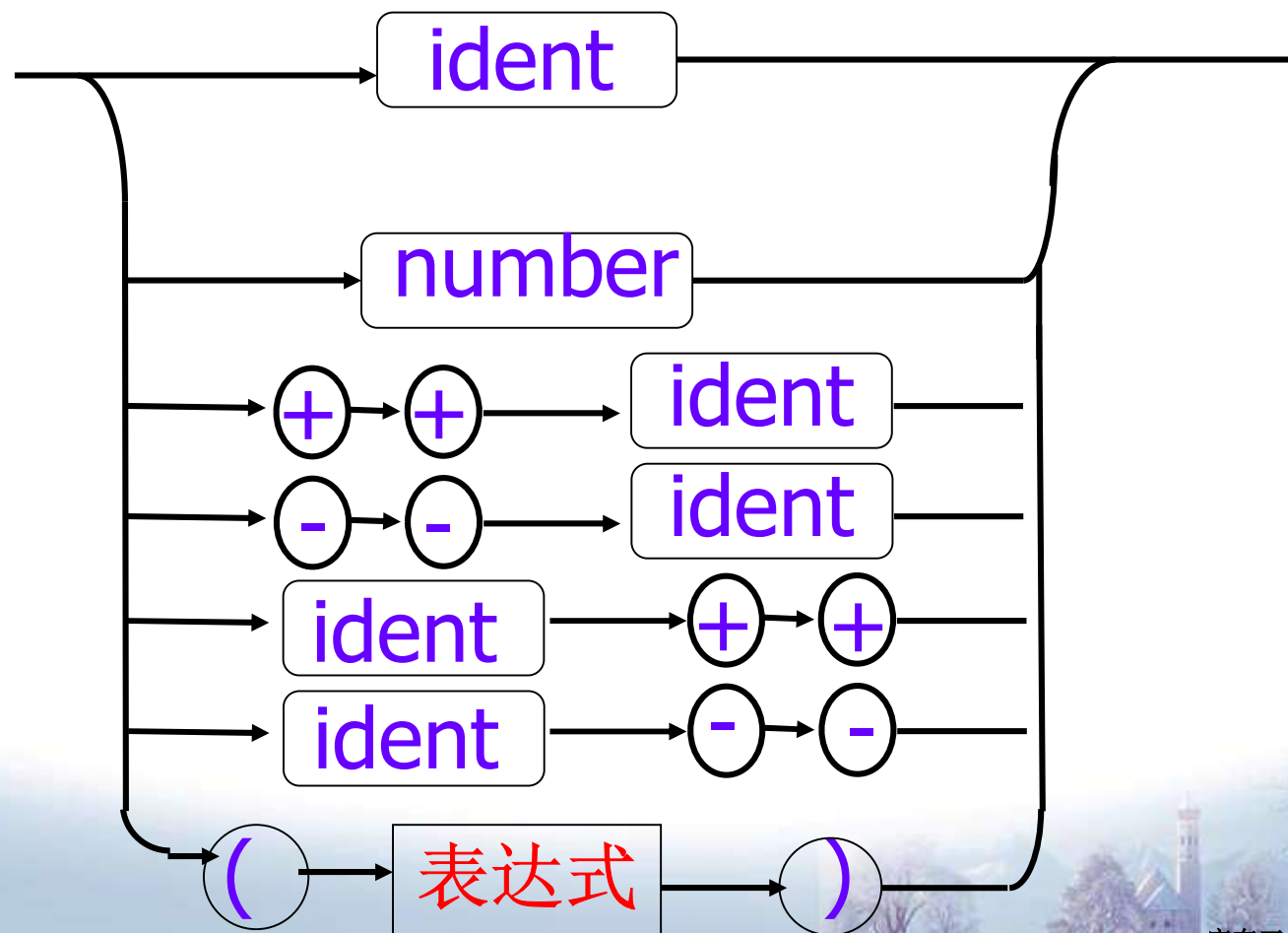


CP课程设计之 ++、--语法分析



语法描述图

因子



CP课程设计之 ++ 、 --语法分析



2.在unit1.cpp中添加代码

❖ 添加位置: **void FACTOR()**

```
//  
void FACTOR(SYMSET FSYS, int LEV, int &TX) {  
    int i;  
    TEST(FACBEGSYS, FSYS, 24);  
    while (SymIn(SYM, FACBEGSYS)) {  
        if (SYM==IDENT) {  
            i=POSITION(ID, TX);  
            if (i==0) Error(11);  
        }  
        else  
            switch (TABLE[i].KIND) {  
                case CONSTANT: GEN(LIT, 0, TABLE[i].VAL); break;  
                case VARIABLE: GEN(LOD, LEV-TABLE[i].vp.LEVEL, TABLE[i].vp.ADR); break;  
                case PROCEDUR: Error(21); break;  
            }  
        GetSym();  
    }  
}
```

CP课程设计之 ++ 、 --语义分析



位置: **void FACTOR**

- 1.检查**ident**是否已正确定义
- 2.生成**++**、**--**对应的目标代码

测试用例:

$s := ++a + b++;$