# 一、实验基础信息

> 个人信息

201700130011 —— 刘建东 —— 17级菁英班

> 实验信息

日期：2020.5.16
题目：PL/0 语言语法分析 —— PL/0 语言翻译模式

# 二、PL/0 语言文法 BNF

```
〈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>;
<语句> → <赋值语句>|<条件语句>|<当型循环语句>|<过程调用语句>|<读语
          句>|<写语句>|<复合语句>|<空>
<赋值语句> → <标识符>:=<表达式>
<复合语句> → begin<语句>{ ; <语句>}end
<条件> → <表达式><关系运算符><表达式>|odd<表达式>
<表达式> → [+|-]<项>{<加减运算符><项>}
<项> → <因子>{<乘除运算符><因子>}
<因子> → <标识符>|<无符号整数>|(<表达式>)
<加减运符> → +|-
<乘除运算符> → *|/
<关系运算符> → =|#|<|<=|>|>=
<条件语句> → if<条件>then<语句>
<过程调用语句> → call<标识符>
<当型循环语句> → while<条件>do<语句>
<读语句> → read(<标识符>{ , <标识符>})
<写语句> → write(<标识符>{, <标识符>})
<字母> → a|b|c…x|y|z
<数字> → 0|1|2…7|8|9
```

> BNF 规则

| 符号 | 意义 |
| --- | --- |
| | |

| | |
|---|---|
| <> | 必选项 |
| [] | 可选项 |
| {} | 可重复0至无数次的项 |
| ::= | 被定义为 |

> 关键字编号

| 编号 | 关键字 | 编号 | 关键字 | 编号 | 关键字 | 编号 | 关键字 |
|---|---|---|---|---|---|---|---|
| 0 | . | 8 | begin | 16 | <> | 24 | while |
| 1 | const | 9 | end | 17 | < | 25 | do |
| 2 | , | 10 | odd | 18 | <= | 26 | read |
| 3 | ; | 11 | + | 19 | > | 27 | write |
| 4 | = | 12 | - | 20 | >= | 28 | ( |
| 5 | var | 13 | * | 21 | if | 29 | ) |
| 6 | procedure | 14 | / | 22 | then | 30 | 标识符 |
| 7 | := | 15 | = | 23 | call | 31 | 常量 |

# 三、翻译模式

PL/0 语言的语法分析一共由如下三部分组成：

1. 构建符号表
2. 语法分析 + 语法树构建
3. 中间代码生成

前两部分在前几周的实验中已经完成，因此接下来的主要任务是中间代码生成。然而中间代码生成也分为两部分，具体如下所示：

1. PL/0 目标指令格式确定
2. PL/0 语句翻译模式确定
3. PL/0 语句生成中间代码

因此本周主要研究 PL/0 目标指令格式以及 PL/0 语句翻译模式。

## 3.1 PL/0 目标指令格式

由于实验指导书上关于 PL/0 目标指令的具体格式列举不够详细，因此上网查找 PL/0 语言更完善、严谨的目标指令信息。

PL/0 语言的目标指令是一种假想的栈式计算机的汇编语言，即一种栈式机的语言。此类栈式机没有累加器和通用寄存器，但有一个栈式存储器，和四个控制寄存器（分别是指令寄存器 I，指令地址寄存器 P，栈顶寄存器 T 和基址寄存器 B），所有的算术逻辑运算都在栈顶进行。

指令格式如下所示：

| F | L | A |
| --- | --- | --- |
| 操作码 | 层次差（标识符引用层-定义层） | 不同的指令含义不同 |

基于上述指令格式，共有 8 类目标指令。

| 指令 | 具体含义 |
| --- | --- |
| $LIT\ 0,a$ | 将常量 $a$ 放到运行栈栈顶 |
| $OPR\ 0,a$ | 栈顶与次栈顶执行运算，结果存放在次栈顶，$a$ 表示执行的运算类型（$a$ 共有17种取值），$a$ 为 $0$ 时即退出数据区 |
| $LOD\ l,a$ | 将变量放到运行栈栈顶（$a$ 为变量在声明层中的相对地址，$l$ 为调用层与声明层的层差值） |
| $STO\ l,a$ | 将运行栈栈顶内容存入变量（$a$ 为相对地址，$l$ 为层差） |
| $CAL\ l,a$ | 调用过程（$a$ 为被调用过程的目标程序的入口地址，$l$ 为层差） |
| $INT\ 0,a$ | 为被调用的过程（或主程序）在运行栈种开辟数据区，即运行栈栈顶指针增加 $a$ |
| $JMP\ 0,a$ | 无条件转移到指令地址 $a$ |
| $JPC\ 0,a$ | 条件转移到指令地址 $a$，即当栈顶的布尔值为假时，转向地址 $a$，否则顺序执行 |

其中 $OPR\ 0,a$ 指令中 $a$ 共有 17 取值，具体内容如下所示：

- $OPR\ 0\ 0\ (RETURN)$

```
stack[sp+1] <- base(l);
sp <- bp-1;
bp <- stack[sp+2];
pc <- stack[sp+3];
```

- $OPR\ 0\ 1\ (NEG)$

```
stack[sp] <- -stack[sp];
```

- $OPR\ 0\ 2\ (ADD)$

```
sp <- sp - 1;
stack[sp] <- stack[sp] + stack[sp+1];
```

- $OPR\ 0\ 3\ (SUB)$

```
sp <- sp - 1;
stack[sp] <- stack[sp] - stack[sp+1];
```

- $OPR\ 0\ 4\ (MUL)$

```
sp <- sp - 1;
stack[sp] <- stack[sp] * stack[sp+1];
```

- $OPR\ 0\ 5\ (DIV)$

```
sp <- sp - 1;
stack[sp] <- stack[sp] / stack[sp+1];
```

- $OPR\ 0\ 6\ (ODD)$

```
stack[sp] <- stack[sp] % 2;
```

- $OPR\ 0\ 7\ (MOD)$

```
sp <- sp - 1;
stack[sp] <- stack[sp] % stack[sp+1];
```

- $OPR\ 0\ 8\ (EQL)$

```
sp <- sp - 1;
stack[sp] <- stack[sp] == stack[sp+1];
```

- $OPR\ 0\ 9\ (NEQ)$

```
sp <- sp - 1;
stack[sp] <- stack[sp] != stack[sp+1];
```

- $OPR\ 0\ 10\ (LSS)$

```
sp <- sp - 1;
stack[sp] <- stack[sp] < stack[sp+1];
```

- $OPR\ 0\ 11\ (LEQ)$

```
sp <- sp - 1;
stack[sp] <- stack[sp] <= stack[sp+1];
```

- $OPR\ 0\ 12\ (GTR)$

```
sp <- sp - 1;
stack[sp] <- stack[sp] > stack[sp+1];
```

- $OPR\ 0\ 13\ (GEQ)$

```
sp <- sp - 1;
stack[sp] <- stack[sp] >= stack[sp+1];
```

- $OPR\ 0\ 14\ (OUT)$

```
print (stack[sp]);
sp <- sp - 1;
```

- $OPR\ 0\ 15\ (OUL)$

```
print ('\n');
```

- $OPR\ 0\ 16\ (IN)$

```
scan (stack[sp]);
sp <- sp + 1;
```

## 3.2 PL/0 翻译模式

由于 PL/0 语言中只有 `if <condition> then <statement>` 和 `while <condition> do <statement>` 两类语句涉及到了跳转地址回填，因此我们主要针对这两种语句给出相应的翻译模式。

- if then 语句

```
        <condition>
        JPC addr
        <statement>
addr:   ...
```

- while do 语句

```
addr2:  <condition>
        JPC addr1
        <statement>
        JPC addr2
addr1:  ...
```

除此之外，我们再明确一下除 OPR 外其它几条指令的具体执行内容：

- $LIT\ 0\ a$

```
sp <- sp + 1;
stack[sp] <- a;
```

- $LOD\ l\ a$

```
sp <- sp + 1;
stack[sp] <- stack[base(L) + a];
```

- $STO\ l\ a$

```
stack[base(l)+a] <- stack[sp];
sp <- sp - 1;
```

- $CAL\ l\ a$

```
stack[sp+1] <- base(l);
stack[sp+2] <- bp;
stack[sp+3] <- pc;
bp <- sp + 1;
pc <- a;
```

- $INT\ 0\ a$

```
sp <- sp + a;
```

- $JMP\ 0\ a$

```
pc <- a;
```

- $JPC\ 0\ a$

```
if stack[sp] == 0:
    pc <- a;
    sp <- sp - 1;
```

# 四、语法分析代码

## 4.1 syntax_analyzer.h

```
#ifndef SYNTAX_ANALYZER_H
#define SYNTAX_ANALYZER_H

#include <string>
#include <vector>

#define Judge(i) if(SYM[p1] == i) {p1++;} \
                 else {return Error();}

struct TableEntry {
    std::string NAME, KIND;
    int VAL, LEVEL, ADR, NXT;
    TableEntry() { ADR = NXT = -1; }
    TableEntry(std::string t1, int t2, int t3, int t4, int t5, int t6){
        NAME = t1; KIND = t2; VAL = t3; LEVEL = t4; ADR = t5; NXT = t6;
    }
};

struct Table {
    int PreTable, PreEntry;
    std::vector<TableEntry> table;
    Table() {
        PreTable = PreEntry = -1;
        table.clear();
    }
    Table(int t1, int t2) : PreTable(t1), PreEntry(t2) {}
};

extern std::vector<Table> TABLE;
extern std::vector<std::string> label;
extern std::vector<std::vector<int> > G;
```

```
bool BLOCK();
void TABLE_OUTPUT(char *outdir);
void GRAPH_OUTPUT(char *outdir);

/*------------------- Procedure Function -------------------*/
bool Procedure();
bool SubProcedure(int);

/*------------------- Specification Function -------------------*/
bool ConstDefinition(int);
bool ConstSpecification(int);
bool VariableSpecification(int);
bool ProcedureSpecification(int);

/*------------------- Statement Function -------------------*/
bool Statement(int);
bool AssignStatement(int);
bool ConditionStatement(int);
bool LoopStatement(int);
bool CallStatement(int);
bool ReadStatement(int);
bool WriteStatement(int);
bool CompoundStatement(int);

/*------------------- Other Functions -------------------*/
bool Item(int);
bool Factor(int);
bool Condition(int);
bool Expression(int);

#endif
```

## 4.2 syntax_analyzer.cpp

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <algorithm>
#include "lexical_analyzer.h"
#include "syntax_analyzer.h"

const std::string CONSTANT = "CONSTANT", VARIABLE = "VARIABLE", PROCEDURE = "PROC
EDURE";
const int DX = 3;
int TX = 0, LEV = 0, ADR = DX, p1, p2, p3, sz;
```

```
std::vector<Table> TABLE;
std::vector<int> empty_vec;
std::vector<std::string> label;
std::vector<std::vector<int> > G;

void DFS_OUTPUT(int TableNum, int EntryNum, int flag, std::ofstream &output) {
    // Table Header Output
    if(flag == 1){
        output << std::left << std::setfill('-') << std::setw(100) << "-" << std:
:endl;
        output << "TX: " << std::left << std::setfill(' ') << std::setw(10) << Ta
bleNum;
        if(EntryNum >= TABLE[TableNum].table.size()){
            output << std::endl;
            return;
        }
        flag = 0;
    }
    else output << "    " << std::left << std::setfill(' ') << std::setw(10) << "
 ";

    // Table Information Output
    output << "NAME: " << std::left << std::setw(20) << TABLE[TableNum].table[Ent
ryNum].NAME;
    output << "KIND: " << std::left << std::setw(25) << TABLE[TableNum].table[Ent
ryNum].KIND;
    if(TABLE[TableNum].table[EntryNum].KIND == CONSTANT)
        output << "VAL: " << std::left << std::setw(15) << TABLE[TableNum].table[
EntryNum].VAL << std::endl;
    else
        output << "LEVEL: " << std::left << std::setw(15) << TABLE[TableNum].tabl
e[EntryNum].LEVEL
                << "ADR: " << TABLE[TableNum].table[EntryNum].ADR << std::endl;

    // Recursion & Backtrack Output
    if(TABLE[TableNum].table[EntryNum].KIND == PROCEDURE) DFS_OUTPUT(TABLE[TableN
um].table[EntryNum].NXT, 0, 1, output), flag = 1;
    if(EntryNum+1 < TABLE[TableNum].table.size()) DFS_OUTPUT(TableNum, EntryNum+1
, flag, output);
}

void TABLE_OUTPUT(char *outdir) {
    std::ofstream output(outdir);
    DFS_OUTPUT(0, 0, 1, output);
    output << std::left << std::setfill('-') << std::setw(100) << "-" << std::end
l;
    output.close();
}
```

```
void GRAPH_OUTPUT(char *outdir) {
    std::ofstream output(outdir);
    int n = label.size();
    output << n << std::endl;
    for(int i = 0; i < n; i++)
        output << label[i] << " \n"[i==n-1];
    for(int i = 0; i < n; i++){
        int sz = G[i].size();
        if(sz == 0) output << std::endl;
        for(int j = 0; j < sz; j++)
            output << G[i][j] << " \n"[j==sz-1];
    }
    output.close();
}

bool Error() {
    std::cout << "Error! p1: " << p1 << std::endl;
    return false;
}

std::string ll2string(long long x){
    std::string result = "";
    while(x){
        result += '0'+(x%10);
        x /= 10;
    }
    reverse(result.begin(), result.end());
    return result;
}

int Graph_init(std::string name, int fa){
    // Graph Construct
    label.push_back(name);
    G.push_back(empty_vec);
    int id = label.size()-1;
    G[fa].push_back(id);
    return id;
}

bool BLOCK() {
    TABLE.push_back(Table());
    p1 = p2 = p3 = 0; sz = SYM.size()-1;
    label.clear(); G.clear();
    return Procedure();
}

/*------------------ Procedure Function ------------------*/
```

```
bool Procedure() {
    // Graph Construct
    label.push_back("程序");
    G.push_back(empty_vec);
    int id = label.size()-1;

    if(!SubProcedure(id)) return Error();
    if(SYM[p1] == 0 && p1 == sz){
        Graph_init(keywords[0], id);
        return true;
    }
    else return Error();
}

bool SubProcedure(int fa) {
    int id = Graph_init("分程序", fa);

    if(SYM[p1] == 1){
        if(!ConstSpecification(id)) return Error();
    }
    if(SYM[p1] == 5){
        if(!VariableSpecification(id)) return Error();
    }
    if(SYM[p1] == 6){
        if(!ProcedureSpecification(id)) return Error();
    }
    return Statement(id);
}

/*------------------- Specification Function -------------------*/
bool ConstDefinition(int fa) {
    int id = Graph_init("常量定义", fa);
    TableEntry entry;
    if(SYM[p1] == tagtype) {
        entry.NAME = ID[p2++];
        entry.KIND = CONSTANT;
        p1++;
        Graph_init(entry.NAME, id);
    }
    else return Error();

    Judge(4);
    Graph_init(keywords[4], id);

    if(SYM[p1] == numtype) {
        entry.VAL = NUM[p3++];
        entry.LEVEL = LEV;
        TABLE[TX].table.push_back(entry);
```

```
        p1++;
        Graph_init(ll2string(entry.VAL), id);
        return true;
    }
    else return Error();
}

bool ConstSpecification(int fa) {
    int id = Graph_init("常量说明部分", fa);
    Judge(1);
    Graph_init(keywords[1], id);

    if(!ConstDefinition(id)) return Error();

    while(SYM[p1] == 2){
        p1++;
        Graph_init(keywords[2], id);
        if(!ConstDefinition(id)) return Error();
    }
    Judge(3);
    Graph_init(keywords[3], id);
    return true;
}

void RecognizeFlag(std::string flag, int fa) {
    TableEntry entry;
    entry.NAME = ID[p2++];
    entry.KIND = flag;
    entry.LEVEL = LEV;
    entry.ADR = ADR++;
    TABLE[TX].table.push_back(entry);
    p1++;
    Graph_init(entry.NAME, fa);
}

bool VariableSpecification(int fa) {
    int id = Graph_init("变量说明部分", fa);
    Judge(5);
    Graph_init(keywords[5], id);
    if(SYM[p1] == tagtype) RecognizeFlag(VARIABLE, id);
    else return Error();

    while(SYM[p1] == 2) {
        p1++;
        Graph_init(keywords[2], id);
        if(SYM[p1] == tagtype) RecognizeFlag(VARIABLE, id);
        else return Error();
    }
```

```
    Judge(3);
    Graph_init(keywords[3], id);
    return true;
}

bool ProcedureSpecification(int fa) {
    int id = Graph_init("过程说明部分", fa);
    Judge(6);
    Graph_init(keywords[6], id);
    if(SYM[p1] == tagtype){
        RecognizeFlag(PROCEDURE, id);
        TABLE.push_back(Table());
        TABLE[TABLE.size()-1].PreTable = TX;
        TABLE[TABLE.size()-1].PreEntry = TABLE[TX].table.size()-1;
        TABLE[TX].table[TABLE[TX].table.size()-1].NXT = TABLE.size()-1;
        TX = TABLE.size()-1, LEV++, ADR = DX;
    }
    else return Error();

    Judge(3);
    Graph_init(keywords[3], id);
    if(!SubProcedure(id)) return Error();
    Judge(3);
    Graph_init(keywords[3], id);
    LEV--;
    int PreEntry = TABLE[TX].PreEntry;
    TX = TABLE[TX].PreTable;
    ADR = ADR - 1 + TABLE[TX].table[PreEntry].ADR;
    TABLE[TX].table[PreEntry].ADR = ADR++;

    while(SYM[p1] == 6){
        if(!ProcedureSpecification(id)) return Error();
    }
    return true;
}

/*------------------- Statement Function -------------------*/
bool Statement(int fa) {
    if(SYM[p1] == tagtype) return AssignStatement(Graph_init("语句", fa));
    if(SYM[p1] == 21) return ConditionStatement(Graph_init("语句", fa));
    if(SYM[p1] == 24) return LoopStatement(Graph_init("语句", fa));
    if(SYM[p1] == 23) return CallStatement(Graph_init("语句", fa));
    if(SYM[p1] == 26) return ReadStatement(Graph_init("语句", fa));
    if(SYM[p1] == 27) return WriteStatement(Graph_init("语句", fa));
    if(SYM[p1] == 8) return CompoundStatement(Graph_init("语句", fa));
    return true;
}
```

```
bool AssignStatement(int fa) {
    int id = Graph_init("赋值语句", fa);
    if(SYM[p1] == tagtype) Graph_init(ID[p2++], id), p1++;
    else return Error();

    Judge(7);
    Graph_init(keywords[7], id);
    return Expression(id);
}

bool ConditionStatement(int fa) {
    int id = Graph_init("条件语句", fa);
    Judge(21);
    Graph_init(keywords[21], id);
    if(!Condition(id)) return Error();
    Judge(22);
    Graph_init(keywords[22], id);
    return Statement(id);
}

bool LoopStatement(int fa) {
    int id = Graph_init("当型循环语句", fa);
    Judge(24);
    Graph_init(keywords[24], id);
    if(!Condition(id)) return Error();
    Judge(25);
    Graph_init(keywords[25], id);
    return Statement(id);
}

bool CallStatement(int fa) {
    int id = Graph_init("过程调用语句", fa);
    Judge(23);
    Graph_init(keywords[23], id);
    if(SYM[p1] == tagtype) Graph_init(ID[p2++], id), p1++;
    else return Error();
    return true;
}

bool ReadStatement(int fa) {
    int id = Graph_init("读语句", fa);
    Judge(26);
    Graph_init(keywords[26], id);
    Judge(28);
    Graph_init(keywords[28], id);
    if(SYM[p1] == tagtype) Graph_init(ID[p2++], id), p1++;
    else return Error();
```

```
    while(SYM[p1] == 2){
        p1++;
        Graph_init(keywords[2], id);
        if(SYM[p1] == tagtype) Graph_init(ID[p2++], id), p1++;
        else return Error();
    }

    Judge(29);
    Graph_init(keywords[29], id);
    return true;
}

bool WriteStatement(int fa) {
    int id = Graph_init("写语句", fa);
    Judge(27);
    Graph_init(keywords[27], id);
    Judge(28);
    Graph_init(keywords[28], id);
    if(SYM[p1] == tagtype) Graph_init(ID[p2++], id), p1++;
    else return Error();

    while(SYM[p1] == 2){
        p1++;
        Graph_init(keywords[2], id);
        if(SYM[p1] == tagtype) Graph_init(ID[p2++], id), p1++;
        else return Error();
    }

    Judge(29);
    Graph_init(keywords[29], id);
    return true;
}

bool CompoundStatement(int fa) {
    int id = Graph_init("复合语句", fa);
    Judge(8);
    Graph_init(keywords[8], id);
    if(!Statement(id)) return Error();
    while(SYM[p1] == 3){
        p1++;
        Graph_init(keywords[3], id);
        if(!Statement(id)) return Error();
    }
    Judge(9);
    Graph_init(keywords[9], id);
    return true;
}
```

```
/*------------------- Other Functions -------------------*/
bool Item(int fa) {
    int id = Graph_init("项", fa);
    if(!Factor(id)) return Error();
    while(SYM[p1] == 13 || SYM[p1] == 14){
        Graph_init(keywords[SYM[p1++]], id);
        if(!Factor(id)) return Error();
    }
    return true;
}

bool Factor(int fa) {
    int id = Graph_init("因子", fa);
    if(SYM[p1] == tagtype) {
        Graph_init(ID[p2++], id), p1++;
        return true;
    }
    if(SYM[p1] == numtype) {
        Graph_init(ll2string(NUM[p3++]), id), p1++;
        return true;
    }
    if(SYM[p1] == 28) {
        p1++;
        Graph_init(keywords[28], id);
        if(!Expression(id)) return Error();
        Judge(29);
        Graph_init(keywords[29], id);
        return true;
    }
    return Error();
}

bool Condition(int fa) {
    int id = Graph_init("条件", fa);
    if(SYM[p1] == 10){
        p1++;
        Graph_init(keywords[10], id);
        return Expression(id);
    }
    else{
        if(!Expression(id)) return Error();
        if(SYM[p1] >= 15 && SYM[p1] <= 20){
            Graph_init(keywords[SYM[p1++]], id);
            return Expression(id);
        }
        else return Error();
    }
}
```

```
bool Expression(int fa) {
    int id = Graph_init("表达式", fa);
    if(SYM[p1] == 11 || SYM[p1] == 12) Graph_init(keywords[SYM[p1++]], id);
    if(!Item(id)) return Error();
    while(SYM[p1] == 11 || SYM[p1] == 12) {
        Graph_init(keywords[SYM[p1++]], id);
        if(!Item(id)) return Error();
    }
    return true;
}
```