一、实验基础信息

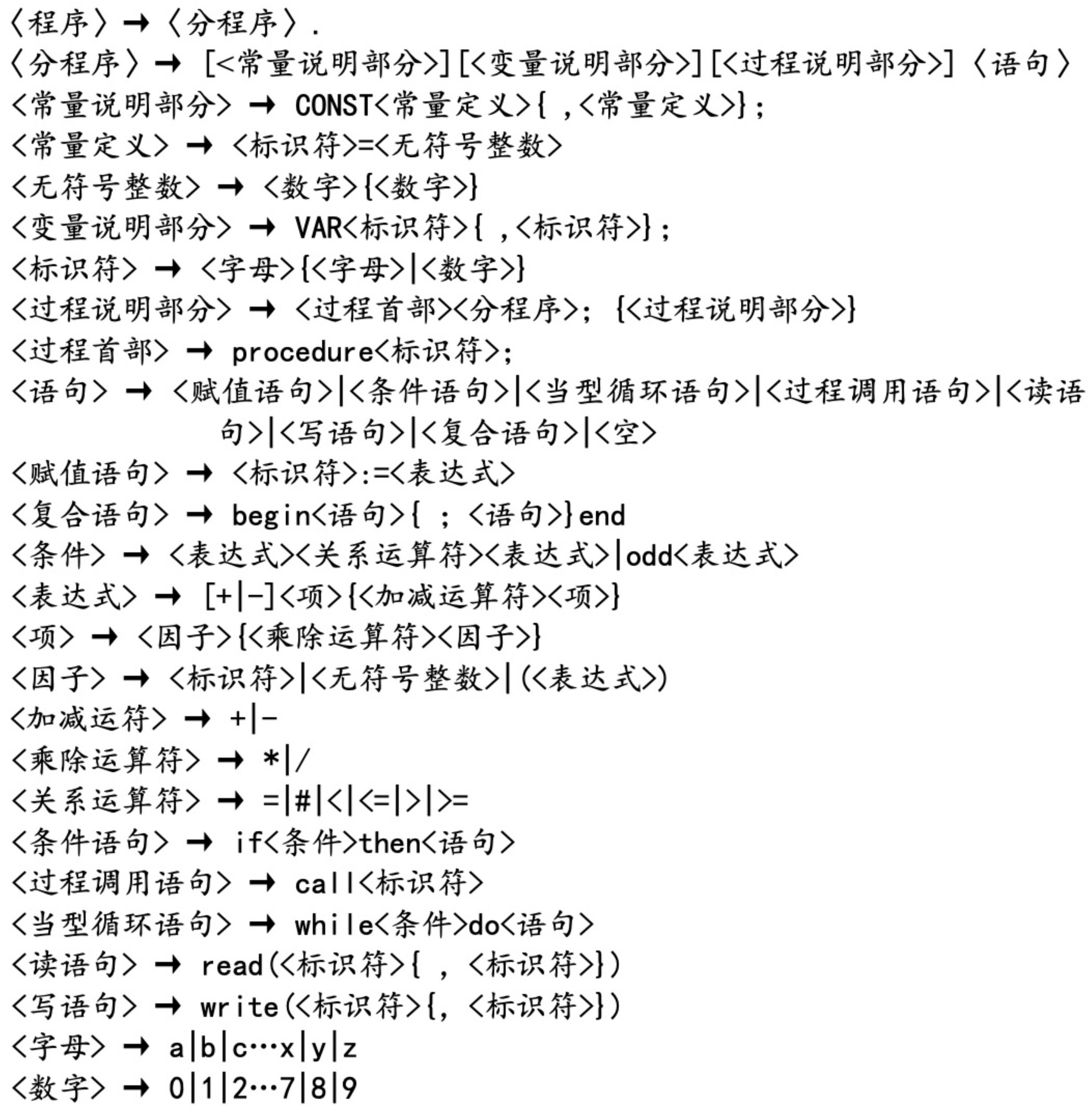
> 个人信息

201700130011 —— 刘建东 —— 17级菁英班

> 实验信息

日期：2020.5.22 题目：PL/0 语言目标指令生成 + 虚拟机运行

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、生成目标指令

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 + 错误类型识别
3. 目标指令生成

3.1 PL/0 目标指令格式

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

3.2 错误类型识别

对于一个编译器来说，识别出具体的错误类型、错误位置是很重要的，因为只有明确指出错误点，编程者才能及时发现错误进行纠正。

在本项目中，错误识别主要由以下代码指出：

```
void Error(std::string error) {
    std::cout << "error: expected " << error << " at the position " << ll2string(p1) << "." << std::endl;
    std::cout << "Syntax Analyzer Error!" << std::endl;
    exit(0);
}
```

其中 $p1$ 为语法分析的位置，$error$ 是错误类型。举下述几个例子来具体介绍错误类型识别的效果。

```
const a=10
var b,c;
procedure p;
...
end.
```

可以发现 `const a=10` 缺少 `;`，程序运行时输出下述结果：

```
Lexical Analyzer Succeed!
error: expected ';' in const declaration at the position 4.
Syntax Analyzer Error!
```

再举一个例子，PL/0 代码如下所示：

```
const a=10;
var b,b;
procedure p;
...
end.
```

可以发现 b 被连续定义了两次，因此程序运行时输出下述结果：

```
Lexical Analyzer Succeed!
error: expected different 'variable name' in variable / const declaration at the
position 8.
Syntax Analyzer Error!
```

3.3 目标指令生成

上周确定了翻译模式，因此这周只需要根据翻译模式进行模拟即可，代码部分不详述了，直接看效果。

示例代码如下所示：

```
var x, squ;

procedure square;
begin
   squ:= x * x
end;

begin
   x := 1;
   while x <= 10 do
   begin
      call square;
      write(squ);
      x := x + 1
   end
end.
```

对应的目标指令如下所示：

```
1 jmp 0 8
2 int 0 3
3 lod 1 3
4 lod 1 3
5 opr 0 4
6 sto 1 4
7 opr 0 0
8 int 0 5
9 lit 0 1
10 sto 0 3
```

```
11 lod 0 3
12 lit 0 10
13 opr 0 11
14 jpc 0 24
15 cal 0 2
16 lod 0 4
17 opr 0 14
18 opr 0 15
19 lod 0 3
20 lit 0 1
21 opr 0 2
22 sto 0 3
23 jmp 0 11
24 opr 0 0
```

可以发现目标指令对应正确，再看下一份示例代码，如下所示：

```
const max = 100;
var arg, ret;

procedure isprime;
var i;
begin
  ret := 1;
  i := 2;
  while i < arg do
  begin
    if arg / i * i == arg then
    begin
      ret := 0;
      i := arg
    end;
    i := i + 1
  end
end;

procedure primes;
begin
  arg := 2;
  while arg < max do
  begin
    call isprime;
    if ret == 1 then write(arg);
    arg := arg + 1
  end
end;

call primes
.
```

即可得到下述目标指令：

```
1 jmp 0 50
2 int 0 4
3 lit 0 1
4 sto 1 4
5 lit 0 2
6 sto 0 3
7 lod 0 3
8 lod 1 3
9 opr 0 10
10 jpc 0 28
11 lod 1 3
12 lod 0 3
13 opr 0 5
14 lod 0 3
15 opr 0 4
16 lod 1 3
17 opr 0 8
18 jpc 0 23
19 lit 0 0
20 sto 1 4
21 lod 1 3
22 sto 0 3
23 lod 0 3
24 lit 0 1
25 opr 0 2
26 sto 0 3
27 jmp 0 7
28 opr 0 0
29 int 0 3
30 lit 0 2
31 sto 1 3
32 lod 1 3
33 lit 0 100
34 opr 0 10
35 jpc 0 49
36 cal 1 2
37 lod 1 4
38 lit 0 1
39 opr 0 8
40 jpc 0 44
41 lod 1 3
42 opr 0 14
43 opr 0 15
44 lod 1 3
45 lit 0 1
46 opr 0 2
```

```
47 sto 1 3
48 jmp 0 32
49 opr 0 0
50 int 0 5
51 cal 0 29
52 opr 0 0
```

四、虚拟机实现

执行目标指令时，需要定义 3 个寄存器变量，分别为：

1. 指令地址寄存器：PC
2. 栈顶地址寄存器：SP
3. 基地址寄存器：BA

除此之外还需注意，每个过程被调用时，需要在栈顶分配三个联系单元，三个单元内容分别为：

1. 静态链：SL，指向定义该过程的直接外过程运行时数据段的基地址
2. 动态链：DL，指向调用该过程前正在运行过程的数据段的基地址
3. 返回地址：RA，记录调用该过程时目标程序的断电，即当时的程序地址寄存器 P 的值

定义完上述内容之后，只需根据上周的翻译模式，针对每一条指令进行具体微操作解释即可，以下述代码为例：

```
var x, y, z, q, r, n, f;

procedure multiply;
var a, b;
begin
  a := x;
  b := y;
  z := x*y
end;

procedure divide;
var w;
begin
  r := x;
  q := 0;
  w := y;
  while w <= r do w := 2 * w;
  while w > y do
  begin
    q := 2 * q;
    w := w / 2;
    if w <= r then
    begin
      r := r - w;
      q := q + 1
    end
  end
end;

procedure gcd;
var  g;
begin
  f := x;
  g := y;
  while f <> g do
  begin
    if f < g then g := g - f;
    if g < f then f := f - g
  end;
  z := f
end;

procedure fact;
begin
  if n > 1 then
  begin
    f := n * f;
    n := n - 1;
    call fact
  end
end;

begin
  read(x);read(y); call multiply;write(z);
  read(x);read(y); call divide; write(q); write(r);
  read(x);read(y); call gcd; write(z);
  read(n); f := 1; call fact; write(f)
end.
```

运行结果如下所示：

```
Lexical Analyzer Succeed!
Syntax Analyzer Succeed!
Choose: '0' for Run, '1' for Debug
0
Please input an integer: 10
Please input an integer: 20
200
Please input an integer: 30
Please input an integer: 40
0
30
Please input an integer: 55
Please input an integer: 7
1
```

```
Please input an integer: 10
3628800
Program Finished Successfully!
```

五、代码

5.1 syntax_analyzer.h

```
#ifndef SYNTAX_ANALYZER_H
#define SYNTAX_ANALYZER_H

#include <string>
#include <vector>
#include <map>

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

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 Instruction {
    std::string f;
    int l, a;
    Instruction() { f = "", l = a = 0; }
    Instruction(std::string f1, int l1, int a1) {
        f = f1; l = l1; a = a1;
    }
};

struct Table {
    int PreTable, PreEntry, VNUM;
    std::vector<TableEntry> table;
    std::map<std::string, std::pair<int, int> > mp1;
    std::map<std::string, int> mp2;
    Table() {
        PreTable = PreEntry = -1;
        VNUM = 0; // 记录变量个数
        table.clear();
        mp1.clear(); // 用于记录常量-(数值,0)，变量-(相对地址,1)
        mp2.clear(); // 用于记录过程-入口地址
    }
    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;
extern std::vector<Instruction> CODE;

void BLOCK();
void TABLE_OUTPUT(char *outdir);
void GRAPH_OUTPUT(char *outdir);
void CODE_OUTPUT(char *outdir);

/*------------------- Procedure Function -------------------*/
void Procedure();
void SubProcedure(int, int);

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

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

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

#endif
```

5.2 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 = "PROCEDURE";
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;
std::vector<Instruction> CODE;

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) << TableNum;
        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[EntryNum].NAME;
    output << "KIND: " << std::left << std::setw(25) << TABLE[TableNum].table[EntryNum].KIND;
    if(TABLE[TableNum].table[EntryNum].KIND == CONSTANT)
        output << "VAL: " << std::left << std::setw(15) << TABLE[TableNum].table[EntryNum].VAL << std::endl;
    else{
        if(TABLE[TableNum].table[EntryNum].KIND == PROCEDURE) {
            output << "LEVEL: " << std::left << std::setw(15) << TABLE[TableNum].table[EntryNum].LEVEL
                  << "ADR: " << " " << std::endl;
        }
        else{
            output << "LEVEL: " << std::left << std::setw(15) << TABLE[TableNum].table[EntryNum].LEVEL
                  << "ADR: " << TABLE[TableNum].table[EntryNum].ADR << std::endl;
        }
    }
    // Recursion & Backtrack Output
    if(TABLE[TableNum].table[EntryNum].KIND == PROCEDURE)
DFS_OUTPUT(TABLE[TableNum].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::endl;
    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();
}

void CODE_OUTPUT(char *outdir) {
    std::ofstream output(outdir);
    int sz = CODE.size();
    for(int i = 0; i < sz; i++)
        output << i+1 << " " << CODE[i].f << " " << CODE[i].l << " " << CODE[i].a <<
std::endl;
    output.close();
}

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

```
}

void Error(std::string error) {
    std::cout << "error: expected " << error << " at the position " << ll2string(p1) << "." << std::endl;
    std::cout << "Syntax Analyzer Error!" << std::endl;
    exit(0);
}

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;
}

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

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

    // the first code: jump to main procedure
    CODE.push_back(Instruction("jmp",0,0));
    SubProcedure(id, 1);

    if(SYM[p1] == 0 && p1 == sz){
        Graph_init(keywords[0], id);
        CODE.push_back(Instruction("opr", 0, 0));
    }
    else Error("'.' in the end of main procedure");
}

void SubProcedure(int fa, int op = 0) {
    int id = Graph_init("分程序", fa);

    if(SYM[p1] == 1) ConstSpecification(id);
    if(SYM[p1] == 5) VariableSpecification(id);
    if(SYM[p1] == 6) ProcedureSpecification(id);
    if(op){
        // main procedure add INT code and backfill JMP addr
        CODE.push_back(Instruction("int", 0, TABLE[0].VNUM+3));
        CODE[0].a = CODE.size();
    }
    Statement(id);
}

/*-------------------- Specification Function --------------------*/
void ConstDefinition(int fa) {
    int id = Graph_init("常量定义", fa);
    TableEntry entry;
    if(SYM[p1] == tagtype) {
        // std::cout << p1 << "," << ID[p2] << "," << SYM[p1] << std::endl;
        entry.NAME = ID[p2++];
        entry.KIND = CONSTANT;
        p1++;
        Graph_init(entry.NAME, id);
    }
    else Error("'tagtype' in const definition");

    Judge(4, "'=' in const definition");
    Graph_init(keywords[4], id);

    if(SYM[p1] == numtype) {
        entry.VAL = NUM[p3++];
        entry.LEVEL = LEV;
        // redifinition const error
        if(TABLE[TX].mp1.find(entry.NAME) != TABLE[TX].mp1.end()) Error("undeclared 'variable name' in const definition");
        // assign value to the map of TABLE[TX], const's second dimensionality is 0
        TABLE[TX].mp1[entry.NAME] = std::make_pair(entry.VAL, 0);
        TABLE[TX].table.push_back(entry);
        p1++;
        Graph_init(ll2string(entry.VAL), id);
    }
    else Error("'numtype' in const definition");
}

void ConstSpecification(int fa) {
    int id = Graph_init("常量说明部分", fa);
    Judge(1, "'const' in const declaration");
    Graph_init(keywords[1], id);

    ConstDefinition(id);

    while(SYM[p1] == 2){
        p1++;
        Graph_init(keywords[2], id);
        ConstDefinition(id);
    }
    Judge(3, "';' in const declaration");
    Graph_init(keywords[3], id);
}

void RecognizeFlag(std::string flag, int fa, int op = 0) {
    TableEntry entry;
    entry.NAME = ID[p2++];
    entry.KIND = flag;
    entry.LEVEL = LEV;
    entry.ADR = ADR++;
    if(op){
        // redifinition procedure error
        if(TABLE[TX].mp2.find(entry.NAME) != TABLE[TX].mp2.end()) Error("different
'procedure name' in procedure declaration");
        // assign value to the map of TABLE[TX]
        TABLE[TX].mp2[entry.NAME] = CODE.size()+1;
    }
    else{
        // redifinition variable error
        if(TABLE[TX].mp1.find(entry.NAME) != TABLE[TX].mp1.end()) return
Error("different 'variable name' in variable / const declaration");
        // assign value to the map of TABLE[TX], variable's second dimensionality is 1
        TABLE[TX].mp1[entry.NAME] = std::make_pair(entry.ADR, 1);
        // the number of variable in TABLE[TX] ++
        TABLE[TX].VNUM++;
    }
    TABLE[TX].table.push_back(entry);
    p1++;
    Graph_init(entry.NAME, fa);
}

void VariableSpecification(int fa) {
    int id = Graph_init("变量说明部分", fa);
    Judge(5, "'var' in variable declaration");
    Graph_init(keywords[5], id);
    if(SYM[p1] != tagtype) Error("'tagtype' in variable declaration");
    RecognizeFlag(VARIABLE, id);

    while(SYM[p1] == 2) {
        p1++;
        Graph_init(keywords[2], id);
        if(SYM[p1] != tagtype) Error("'tagtype' in variable declaration");
        RecognizeFlag(VARIABLE, id);
    }
    Judge(3, "';' in variable declaration");
    Graph_init(keywords[3], id);
}
```

```
void ProcedureSpecification(int fa) {
    int id = Graph_init("过程说明部分", fa);
    Judge(6, "'procedure' in procedure declaration");
    Graph_init(keywords[6], id);
    if(SYM[p1] == tagtype){
        RecognizeFlag(PROCEDURE, id, 1);
        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("'tagtype' in procedure declaration");

    Judge(3, "';' in procedure declaration");
    Graph_init(keywords[3], id);

    // add the first INT code of procedure
    int index = CODE.size();
    CODE.push_back(Instruction("int",0,0));
    SubProcedure(id);
    // backfill the size of INT code
    CODE[index].a = TABLE[TX].VNUM + 3;

    Judge(3, "';' in procedure declaration");
    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++;
    CODE.push_back(Instruction("opr", 0, 0));

    while(SYM[p1] == 6) ProcedureSpecification(id);
}

/*------------------- Statement Function -------------------*/
std::pair<int,int> SearchPosition(std::string NAME, int op = 0) {
    /*
        constant: (-1, val), variable: (l, a), procedure: (l, addr)
        variable name should differ from const name
        procedure name could be same with variable or const name
    */
    int now = TX, cnt = 0;
    while(now != -1) {
        if(!op && TABLE[now].mp1.find(NAME) != TABLE[now].mp1.end()){
            if(TABLE[now].mp1[NAME].second == 0) return std::make_pair(-1,
TABLE[now].mp1[NAME].first);
```

```
            else return std::make_pair(cnt, TABLE[now].mp1[NAME].first);
        }
        // search procedure
        if(op && TABLE[now].mp2.find(NAME) != TABLE[now].mp2.end())
            return std::make_pair(cnt, TABLE[now].mp2[NAME]);
        now = TABLE[now].PreTable, cnt++;
    }
    return std::make_pair(-3, -1);
}

void Statement(int fa) {
    if(SYM[p1] == tagtype) AssignStatement(Graph_init("语句", fa));
    else if(SYM[p1] == 21) ConditionStatement(Graph_init("语句", fa));
    else if(SYM[p1] == 24) LoopStatement(Graph_init("语句", fa));
    else if(SYM[p1] == 23) CallStatement(Graph_init("语句", fa));
    else if(SYM[p1] == 26) ReadStatement(Graph_init("语句", fa));
    else if(SYM[p1] == 27) WriteStatement(Graph_init("语句", fa));
    else if(SYM[p1] == 8) CompoundStatement(Graph_init("语句", fa));
}

void 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("'tagtype' in assign statement");

    // acquire the position of variable
    std::pair<int, int> pos = SearchPosition(ID[p2-1]);
    // pos must be variable
    if(pos.first < 0) return Error("declared 'variable name' in assign statement");

    Judge(7, "':=' in assign statement");
    Graph_init(keywords[7], id);
    Expression(id);
    // assign code
    CODE.push_back(Instruction("sto", pos.first, pos.second));
}

void ConditionStatement(int fa) {
    int id = Graph_init("条件语句", fa);
    Judge(21, "'if' in condition statement");
    Graph_init(keywords[21], id);
    Condition(id);
    CODE.push_back(Instruction("jpc", 0, 0));
    int index = CODE.size();

    Judge(22, "'then' in condition statement");
    Graph_init(keywords[22], id);
    Statement(id);
    CODE[index-1].a = CODE.size()+1;
```

```
}

void LoopStatement(int fa) {
    int id = Graph_init("当型循环语句", fa);
    Judge(24, "'while' in loop statement");
    Graph_init(keywords[24], id);
    int index1 = CODE.size();
    Condition(id);
    CODE.push_back(Instruction("jpc", 0, 0));
    int index2 = CODE.size();

    Judge(25, "'do' in loop statement");
    Graph_init(keywords[25], id);
    Statement(id);
    CODE.push_back(Instruction("jmp", 0, index1+1));
    CODE[index2-1].a = CODE.size()+1;
}

void CallStatement(int fa) {
    int id = Graph_init("过程调用语句", fa);
    Judge(23, "'call' in call procedure statement");
    Graph_init(keywords[23], id);
    if(SYM[p1] == tagtype) {
        // acquire the position of precedure
        std::pair<int, int> pos = SearchPosition(ID[p2], 1);
        // pos must be precedure
        if(pos.first >= 0) CODE.push_back(Instruction("cal", pos.first, pos.second));
        else Error("declared 'procedure name' in call procedure statement");

        Graph_init(ID[p2++], id), p1++;
    }
    else return Error("declared 'procedure name' in call procedure statement");
}

void ReadStatement(int fa) {
    int id = Graph_init("读语句", fa);
    Judge(26, "'read' in read statement");
    Graph_init(keywords[26], id);
    Judge(28, "'(' in read statement");
    Graph_init(keywords[28], id);
    if(SYM[p1] == tagtype) {
        CODE.push_back(Instruction("opr", 0, 16));
        // acquire the position of variable
        std::pair<int, int> pos = SearchPosition(ID[p2]);
        // pos must be variable
        if(pos.first >= 0) CODE.push_back(Instruction("sto", pos.first, pos.second));
        else Error("declared 'variable name' in read statement");

        Graph_init(ID[p2++], id), p1++;
    }
    else Error("declared 'variable name' in read statement");

    while(SYM[p1] == 2){
        p1++;
        Graph_init(keywords[2], id);
        if(SYM[p1] == tagtype) {
            CODE.push_back(Instruction("opr", 0, 16));
            // acquire the position of variable
            std::pair<int, int> pos = SearchPosition(ID[p2]);
            // pos must be variable
            if(pos.first >= 0) CODE.push_back(Instruction("sto", pos.first,
pos.second));
            else Error("declared 'variable name' in read statement");

            Graph_init(ID[p2++], id), p1++;
        }
        else Error("declared 'variable name' in read statement");
    }

    Judge(29, "')' in read statement");
    Graph_init(keywords[29], id);
}

void WriteStatement(int fa) {
    int id = Graph_init("写语句", fa);
    Judge(27, "'write' in write statement");
    Graph_init(keywords[27], id);
    Judge(28, "'(' in write statement");
    Graph_init(keywords[28], id);
    Expression(id);
    CODE.push_back(Instruction("opr", 0, 14));
    CODE.push_back(Instruction("opr", 0, 15));

    while(SYM[p1] == 2){
        p1++;
        Graph_init(keywords[2], id);
        Expression(id);
        CODE.push_back(Instruction("opr", 0, 14));
        CODE.push_back(Instruction("opr", 0, 15));
    }

    Judge(29, "')' in write statement");
    Graph_init(keywords[29], id);
}

void CompoundStatement(int fa) {
    int id = Graph_init("复合语句", fa);
    Judge(8, "'begin' in compound statement");
    Graph_init(keywords[8], id);
    Statement(id);
    while(SYM[p1] == 3){
        p1++;
        Graph_init(keywords[3], id);
        Statement(id);
    }
    Judge(9, "'end' in compound statement");
    Graph_init(keywords[9], id);
}

/*-------------------- Other Functions --------------------*/
void Item(int fa) {
    int id = Graph_init("项", fa), op = 0;
    Factor(id);
    while(SYM[p1] == 13 || SYM[p1] == 14){
        if(SYM[p1] == 13) op = 1;
        else op = -1;
        Graph_init(keywords[SYM[p1++]], id);
        Factor(id);
        // solve *、/
        if(op == 1) CODE.push_back(Instruction("opr", 0, 4));
        else CODE.push_back(Instruction("opr", 0, 5));
    }
}

void Factor(int fa) {
    int id = Graph_init("因子", fa);
    if(SYM[p1] == tagtype) {
        // acquire the position of variable / constant
        std::pair<int, int> pos = SearchPosition(ID[p2]);
        // pos must be variable
        if(pos.first >= 0) CODE.push_back(Instruction("lod", pos.first, pos.second));
        else if(pos.first == -1) CODE.push_back(Instruction("lit", 0, pos.second));
        else Error("declared 'variable name' as a factor");

        Graph_init(ID[p2++], id), p1++;
        return;
    }
    if(SYM[p1] == numtype) {
        CODE.push_back(Instruction("lit", 0, NUM[p3]));
        Graph_init(ll2string(NUM[p3++]), id), p1++;
        return;
    }
    if(SYM[p1] == 28) {
        p1++;
        Graph_init(keywords[28], id);
        Expression(id);
        Judge(29, "')' in the end of expression statement as a factor");
        Graph_init(keywords[29], id);
        return;
    }
    return Error("'tagtype' / 'numtype' / 'expression statement' as a factor");
}

void Condition(int fa) {
    int id = Graph_init("条件", fa);
    if(SYM[p1] == 10){
        p1++;
        Graph_init(keywords[10], id);
        Expression(id);
        CODE.push_back(Instruction("opr", 0, 6));
    }
    else{
        Expression(id);
        if(SYM[p1] >= 15 && SYM[p1] <= 20) {
            Graph_init(keywords[SYM[p1++]], id);
            int op = SYM[p1-1];
            Expression(id);
            CODE.push_back(Instruction("opr", 0, op-7));
        }
        else Error("'operator specifier' between expression statements");
    }
}

void Expression(int fa) {
    int id = Graph_init("表达式", fa), op = 0;
    if(SYM[p1] == 11 || SYM[p1] == 12) {
        if(SYM[p1] == 12) op = -1;
        Graph_init(keywords[SYM[p1++]], id);
    }
    Item(id);
    // solve uminus
    if(op == -1) {
        CODE.push_back(Instruction("lit", 0, -1));
        CODE.push_back(Instruction("opr", 0, 4));
    }
    while(SYM[p1] == 11 || SYM[p1] == 12) {
        if(SYM[p1] == 11) op = 1;
        else op = -1;
        Graph_init(keywords[SYM[p1++]], id);
        Item(id);
        // solve +、-
        if(op == 1) CODE.push_back(Instruction("opr", 0, 2));
        else CODE.push_back(Instruction("opr", 0, 3));
    }
}
```

5.3 code_interpreter.h

```
#ifndef CODE_INTERPRETER_H
#define CODE_INTERPRETER_H

#include <string>
#include <vector>
#include <iostream>

extern int CAPACITY, BASE, PC, SP, BA;
extern std::vector<int> S;

void Run();

#endif
```

5.4 code_interpreter.cpp

```
#include "syntax_analyzer.h"
#include "code_interpreter.h"

int CAPACITY = 100, BASE = 100, PC, SP, BA;
std::vector<int> S;

void RuntimeError(std::string error) {
    std::cout << "error: " << error << " at the CODE[" << PC << "], "
              << CODE[PC-1].f << " " << CODE[PC-1].l << " " << CODE[PC-1].a << "." <<
std::endl;
    std::cout << "Code Interpreter Error!" << std::endl;
    exit(0);
}

void Push(int v) {
    if(SP+2 > S.size()) CAPACITY += BASE, S.resize(CAPACITY);
    S[++SP] = v;
}

bool OneInstruction() {
    std::string op = CODE[PC-1].f;
    int l = CODE[PC-1].l, a = CODE[PC-1].a;
    if(op == "lit") {
        Push(a);
    }
    else if(op == "lod") {
        int tb = BA;
        while(l--) tb = S[tb];
        Push(S[tb+a]);
    }
```

```
        else if(op == "sto") {
            int tb = BA;
            while(l--) tb = S[tb];
            if(tb+a >= S.size()) RuntimeError("access the undefined stack space");
            S[tb+a] = S[SP];
            SP--;
        }
        else if(op == "cal") {
            int tb = BA;
            while(l--) tb = S[tb];
            Push(tb);
            Push(BA);
            Push(PC);
            SP -= 3;
            PC = a-1;
            BA = SP+1;
        }
        else if(op == "int") {
            SP += a;
        }
        else if(op == "jmp") {
            PC = a-1;
        }
        else if(op == "jpc") {
            if(!S[SP]) PC = a-1;
            SP--;
        }
        else if(op == "opr") {
            switch (a)
            {
                case 0:
                    SP = BA-1;
                    PC = S[BA+2];
                    BA = S[BA+1];
                    break;
                case 1:
                    S[SP] = -S[SP];
                    break;
                case 2:
                    SP--;
                    S[SP] = S[SP] + S[SP+1];
                    break;
                case 3:
                    SP--;
                    S[SP] = S[SP] - S[SP+1];
                    break;
                case 4:
                    SP--;
                    S[SP] = S[SP] * S[SP+1];
                    break;
                case 5:
                    SP--;
                    if(S[SP+1] == 0) RuntimeError("divisor is 0");
                    S[SP] = S[SP] / S[SP+1];
                    break;
                case 6:
                    S[SP] = S[SP] % 2;
                    break;
                case 7:
                    SP--;
                    S[SP] = S[SP] % S[SP+1];
                    break;
                case 8:
                    SP--;
                    S[SP] = (S[SP] == S[SP+1]);
                    break;
                case 9:
                    SP--;
                    S[SP] = (S[SP] != S[SP+1]);
                    break;
                case 10:
                    SP--;
                    S[SP] = (S[SP] < S[SP+1]);
                    break;
                case 11:
                    SP--;
                    S[SP] = (S[SP] <= S[SP+1]);
                    break;
                case 12:
                    SP--;
                    S[SP] = (S[SP] > S[SP+1]);
                    break;
                case 13:
                    SP--;
                    S[SP] = (S[SP] >= S[SP+1]);
                    break;
                case 14:
                    std::cout << S[SP--];
                    break;
                case 15:
                    std::cout << "\n";
                    break;
                case 16:
                    Push(0);
                    std::cout << "Please input an integer: ";
                    std::cin >> S[SP];
                    break;
                default:
                    break;
            }
        }
        return true;
    }

    void Run() {
        S.resize(CAPACITY);
        PC = 1, SP = -1, BA = 0;
        Push(0);
        Push(0);
        Push(-1);
        SP -= 3;

        std::cout << "Choose: '0' for Run, '1' for Debug" << std::endl;
        int op;
        while(std::cin >> op) {
            if(op == 0 || op == 1) break;
            else std::cout << "Choose: '0' for Run, '1' for Debug" << std::endl;
        }
        if(!op) {
            while(PC) {
                OneInstruction();
                PC += 1;
            }
        }
        else {
            while(PC) {
                std::cout << "PC: " << PC << ", CODE[PC]: " << CODE[PC-1].f << " " <<
    CODE[PC-1].l << " " << CODE[PC-1].a << std::endl;
                OneInstruction();
                std::cout << "One Instruction Executed, BA: " << BA << ", SP: " << SP <<
    ", STACK[0~10]: [";
                for(int i = 0; i <= 10; i++)
                    std::cout << S[i] << " ]"[i==10];
                std::cout << std::endl << "Please continue" << std::endl;
                getchar();
                PC += 1;
            }
        }
    }
```