

# 苏州大学实验报告

院、系	计算机科学与技术学院	年级专业	21 计科	姓名	赵鹏	学号	2127405037
课程名称	编译原理实践					成绩	
指导教师	段湘煜	同组实验者	无	实验日期	2023.11.27		

实验名称 语义分析

## 一. 实验题目

基于翻译方案，采用递归下降的分析方法生成抽象机的汇编指令代码。

## 二. 实验原理及流程图

### 1. 中间代码

中间代码在编译过程中充当桥梁，连接高级语言与机器码，具有与机器无关、便于优化和移植的优势。常见中间代码形式包括波兰后缀表示、N-元表示、抽象语法树和抽象机代码，每种都有着不同的特点和适用场景。它们的选择取决于编译器设计的需求和实际情况，有些更适合优化处理，有些更便于转换成目标机器指令。

使用中间代码有如下优点：

1. 简化实现：生成中间代码时无需考虑特定机器特性，简化了编译器的设计和实现。
2. 可移植性：中间代码与特定机器无关，使得编译器更易于移植到不同平台上。
3. 便于优化：提供了更抽象的表示形式，使得编译器在中间代码上进行优化更为便利。

### 2. 栈式抽象机及其汇编指令

目标平台采用的机器是一台抽象的栈式计算机，它用一个栈来保存操作数，并有足够的内存空间。该抽象机的常用汇编指令如下：

LOAD D 将 D 中的内容加载到操作数栈；

LOADI 常量 将常量压入操作数栈；

STO D 将操作数栈栈顶单元内容存入 D,且栈顶单元内容保持不变；

POP 将操作数栈栈顶出栈；

ADD 将次栈顶单元与栈顶单元内容出栈并相加,和置于栈顶；

SUB 将次栈顶单元与栈顶单元内容出栈并相减,差置于栈顶；

MULT 将次栈顶单元与栈顶单元内容出栈并相乘,积置于栈顶；

DIV 将次栈顶单元与栈顶单元内容出栈并相除,商置于栈顶；

BR lab 无条件转移到 lab；

BRF lab 检查栈顶单元逻辑值并出栈,若为假(0)则转移到 lab；

EQ 将栈顶两单元做等于比较并出栈,并将结果真或假(1 或 0)置于栈顶；

NOTEQ 将栈顶两单元做不等于比较并出栈,并将结果真或假(1 或 0)置于栈顶；

GT 次栈顶大于栈顶操作数并出栈,则栈顶置 1,否则置 0；

LES 次栈顶小于栈顶操作数并出栈,则栈顶置 1,否则置 0；

GE 次栈顶大于等于栈顶操作数并出栈,则栈顶置 1,否则置 0；

LE 次栈顶小于等于栈顶操作数并出栈,则栈顶置 1,否则置 0;  
 AND 将栈顶两单元做逻辑与运算并出栈,并将结果真或假(1 或 0)置于栈顶;  
 OR 将栈顶两单元做逻辑或运算并出栈,并将结果真或假(1 或 0)置于栈顶;  
 NOT 将栈顶的逻辑值取反;  
 IN 从标准输入设备(键盘)读入一个整型数据,并入操作数栈;  
 OUT 将栈顶单元内容出栈,并输出到标准输出设备上(显示器);  
 STOP 停止执行;

### 3. 翻译方案

针对各个语法规则制定翻译方案（语法规则中加入花括号括起来的语义动作）。

#### (1) 声明的处理

符号表用来存储变量信息，由于 TEST 语言的变量只有整型，可以按如下建立符号表：

```
Class symbolTable
{
  map<string, int>symbolTable;//从变量名到地址
  map<int, int>addressValue;//从地址到值
}
```

其中, symbolTable 用于保存变量名的地址, addressValue 用于记录地址中存储的值,可以使用 symbolTable.size()获取符号表中已有的变量个数,通过 find 方法判断某个变量是否已经存储在符号表中。

针对声明语法规则，其翻译方案如下：

$\langle declaration\_stat \rangle \rightarrow int\ ID\ \{name - def\ function\}$

其中 name-def function 完成在符号表中插入当前变量：首先查询符号表，从表中最后一个变量开始向前查找直到第一个变量，如果找到了当前这个变量，则报重复定义的错误，否则，将当前变量插到符号表的最后。

#### (2) 表达式语句

其翻译方案为：

$\langle expression\_stat \rangle \rightarrow \langle expression \rangle \{POP;\}$

因为表达式的计算结果会保留在操作数栈的栈顶，因此表达式语句的翻译方案只是在表达式后面加上动作 POP，将栈顶内容弹出栈。

expression，其翻译方案如下：

```
< expression > → ID {LOOKUP(n,d); ASSIGN;} = < bool_expr > {STO d} |
  < bool_expr > < bool_expr > → < additive_expr >
| < additive_expr > > < additive_expr > {GT}
| < additive_expr > < < additive_expr > {LES}
| < additive_expr > >= < additive_expr > {GE}
| < additive_expr > <= < additive_expr > {LE}
| < additive_expr > == < additive_expr > {EQ}
| < additive_expr > != < additive_expr > {NOTEQ}
< additive_expr > → < term > {+ < term > {ADD}} | - < TERM > {SUB}}
< term > → < factor > { * < factor > {MULT}} | / < factor > {DIV}}
< factor > → (< expression >) | ID {LOOKUP(n,d); LOAD d;} | NUM {LOADI I;}
```

翻译方案中的各个动作解释如下：

*LOOKUP*(*n*,*d*): 在符号表中查找当前变量 *n*, 返回地址 *d*; 若没有, 则变量未定义, 报错。

*ASSIGN*: 超前读一个符号, 如果是=, 则表示进入赋值表达式; 如果不是=, 则选择< *bool\_expr* >, 然后将超前读的符号退回。

*STO d*: 输出指令代码 *STO d*;

*LOAD d*: 输出指令代码 *LOAD d*;

*LOADI i*: 输出指令代码 *LOADI i*;

*GT*、*ADD* 等: 输出指令代码 *GT*、*ADD* 等。

### (3) If 语句

其翻译方案为:

< *if\_stat* > → *if*(< *expression* >){*BRF*(*label1*);}

< *statement* > {*BR*(*label2*);*SETLabel*(*label1*);}

[*else* < *statement* >] {*SETLabel*(*label2*);}

翻译方案中的各个动作解释如下:

*BRF*(*label1*): 输出 *BRF label1*

*BR*(*label2*): 输出 *BR label2*

*SETLabel*(*label1*): 设置标号 *label1*

### (4) While 语句

其翻译方案为:

< *while\_stat* > → *while*{*SETLabel*(*label1*);}< *expression* >){*BRF*(*label2*);}

< *statement* > {*BR*(*label1*); *SETLabel*(*label2*);}

### (5) For 循环语句

其翻译方案为:

< *for\_stat* > → *for*(< *expression* > {*POP*;};

{*SETLabel*(*label1*);}< *expression* > {*BRF*(*label2*);*BR*(*label3*);};

{*SETLabel*(*label4*);}< *expression* > {*POP*; *BR*(*label1*);})

{*SETLabel*(*label3*);}< *statement* > {*BR*(*label4*); *SETLabel*(*label2*);}

### (6) Read 语句

其翻译方案如下:

< *read\_stat* > → *read ID* {*LOOKUP*(*n*,*d*); *IN*; *STO*(*d*); *POP*}

其中的某些动作解释如下:

*LOOKUP*(*n*,*d*): 在符号表中查找当前变量 *n*, 返回地址 *d*; 若没有, 则变量未定义, 报错。

*IN*: 输出指令代码 *IN*

*STO*(*d*): 输出指令代码 *STO d*

### (7) write 语句

其翻译方案如下：

`< write_stat > → write < expression > {OUT;}`

其中动作符号解释如下：

*OUT*：输出指令代码 *OUT*。

## 三. 实验步骤

### 1. 实现符号表

为了记录变量相关信息，需要实现符号表的功能，定义`SymbolTable`类用于实现符号表相关功能，具有`map < string, int > symbolTable`，`map < int, int > addressValue`两个成员变量用于实现变量名到地址的转换和记录地址中的值。并实现将变量插入符号表的`insertVariable`函数用于定义新变量，返回插入的地址或-1表示重复定义。实现`LookUpVariable`用于获取变量的地址，返回变量的地址或-1表示变量未定义。随后便可以在添加语义动作时简化实现。

### 2. 实现部分语义动作

代码中主要实现了`Label`和`jump`相关的语义动作，定义全局变量`labelCount`用于为`label`计数，定义`getNewLabel`和`setLabel`函数分别用于获取一个新的 `label` 编号和在当前位置添加一个`label x`:的标记。定义`jumpBR`和`jumpBRF`函数分别用于实现实验原理中的`BR`动作和`BRF`动作，在当前位置添加一个 `BR label`标记或 `BRF label`标记。

其余部分语义动作如`LOAD`, `STO`, `POP`等只需要在处理各个产生式的函数的对应位置将这些动作以字符串形式输出到文件即可，操作非常简单，因此不再单独编写函数处理。

## 四. 实验结果及分析

测试一：

输入：

```
{
    int i;
    int a;
    for (i=1;i<=2;i=i+1)
        a=1;
}
```

输出：

```
LOADI 1
STO 0
POP
LABEL1:
LOAD 0
```

```

LOADI 2
LE
BRF LABEL2
BR LABEL3
LABEL4:
LOAD 0
LOADI 1
ADD
STO 0
POP
BR LABEL1
LABEL3:
LOADI 1
STO 1
POP
BR LABEL4
LABEL2:

```

测试二:

输入:

```

{
    int i;
    int j;
    int n;
    int m;
    int k;
    int c;
    int a;
    int b;
    j=1;
    read n;
    for(i=1; i<=n; i=i+1) {
        for (j=1;j<=m;j=j+1) {
            write i+j;
            for (k=1;1==2;2<=3) {
                c=c+1;
                if (c==1) {
                    a=a+1;
                    b=b-1;
                }else{
                    write i;
                }
            }
        }
    }
}

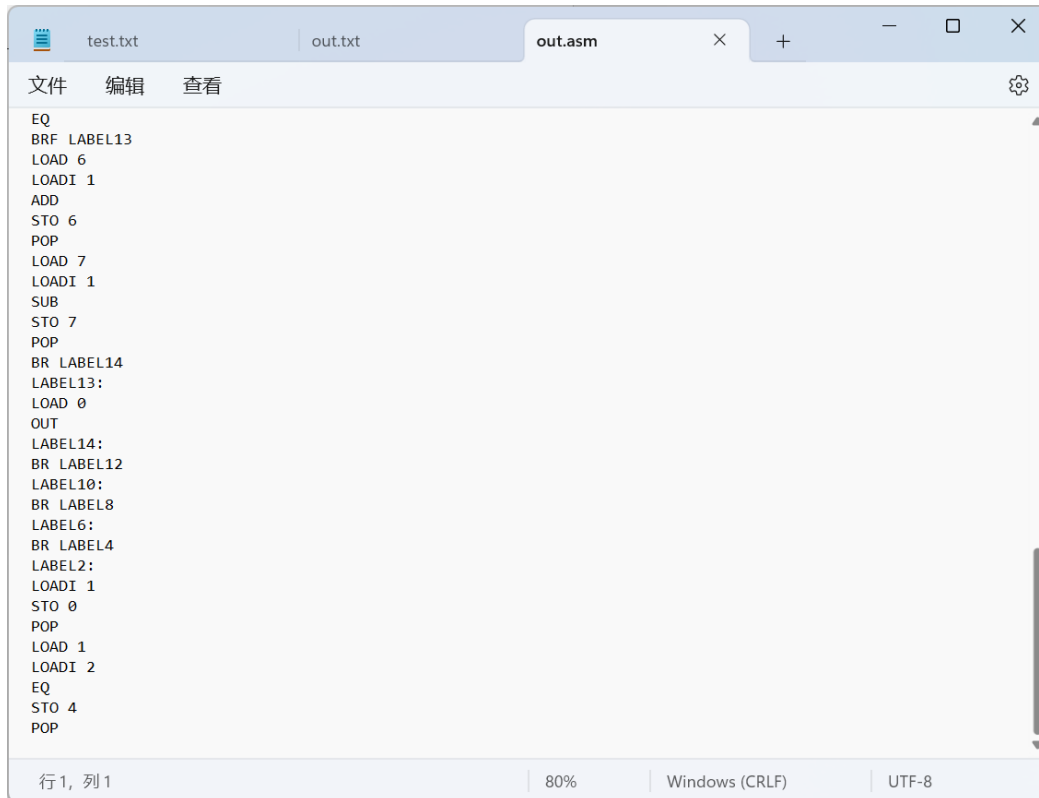
```

```

    }
}
i=1;
k=j==2;
}

```

输出：



```

EQ
BRF LABEL13
LOAD 6
LOADI 1
ADD
STO 6
POP
LOAD 7
LOADI 1
SUB
STO 7
POP
BR LABEL14
LABEL13:
LOAD 0
OUT
LABEL14:
BR LABEL12
LABEL10:
BR LABEL8
LABEL6:
BR LABEL4
LABEL2:
LOADI 1
STO 0
POP
LOAD 1
LOADI 2
EQ
STO 4
POP

```

测试三：

输入：

```

{
    int i;
    int i;
    for (i=1;i<=2;i=i+1)
        write i;
}

```

输出：

```

[ERROR] row:3 col:7 Redeclaration of variable i
== Grammer prase result ==
Redefinition of variable
汇编代码生成失败

```

测试四:

输入:

```
{  
  
    for (i=1;i<=2;i=i+1)  
        write i;  
}
```

输出:

```
[ERROR]row:3 col:11 Undeclared variable i  
== Grammer prase result ==  
Undefinition of variable  
汇编代码生成失败
```

## 五. 实验总结

通过本次实验,我对语义分析的过程有了更加深刻的理解,学习到了在递归下降进行语法分析的过程中同时进行语义分析的方法,尽管本次实验相对简单,但我对符号表管理、添加语义动作、生成汇编代码等过程都有了更加深刻的理解。但使用递归下降进行回溯的方法进行语法分析同时进行语义分析的时间复杂度相对较高,我设想可以在实验五的基础上将语法树构建出来,并在 GrammarItem 类中添加语义动作(以函数指针的方式存储),最后递归下降的遍历语法树也可以实现和基于实验四的语义分析类似的效果。

## 六. 代码

本次实验主要是对实验四中语法分析的代码的修改,Prase代码如下:

```
#include "utility.h"  
//#define DEBUG  
int TESTparse();  
int program();  
int compoundStat();  
int statement();  
int expressionStat();  
int expression();  
int boolExpr();  
int additiveExpr();  
int term();  
int factor();  
int ifStat();  
int whileStat();  
int forStat();  
int writeStat();  
int readStat();
```

```

int declarationStat();
int declarationList();
int statementList();
char Scanin[300] = "out.txt", Scanout[300] = "out.asm";
std::ifstream fin;
std::ofstream fout;
struct Token
{
    std::string tokenType, tokenValue;
    int row, column;
};
std::ifstream& operator>>(std::ifstream& in, Token& token)
{
    fin >> token.tokenType >> token.tokenValue >> token.row >>
token.column;
    // std::cout << "TokenType:" << token.tokenType << "  TokenValue:" <<
token.tokenValue << "  Row:"<< token.row<< "  Column:"<<token.column <<
std::endl;
    return fin;
}
std::iostream& operator<<(std::iostream& out, Token& token)
{
    std::cout << "TokenType:" << token.tokenType << "  TokenValue:" <<
token.tokenValue << std::endl;
    return out;
}
class SymbolTable
{
public:
    map<string, int>symbolTable;//从变量名到地址
    map<int, int>addressValue;//从地址到值
    int insertVariable(string variableName)
    {
        if (symbolTable.find(variableName) != symbolTable.end())
        {
            return -1;
        }
        int siz = symbolTable.size();
        symbolTable[variableName] = siz;
        return siz;
    }

    void LookUpVariable(string variableName, int& addr)
    {

```



```

        if (symbolTable.find(variableName) == symbolTable.end())
        {
            addr = -1;
            return;
        }
        addr = symbolTable[variableName];
        return;
    }

```

```

}variabletable;
extern char Scanout[300];
int lableCount=1;
int getNewLabel()
{
    return lableCount++;
}
void setLable(int label)
{
    fout << "LABEL" << label << ":" << endl;
    cout << "LABEL" << label << ":" << endl;
}
void jumpBR(int label)
{
    fout << "BR" << " LABEL" << label << endl;
    cout << "BR" << " LABEL" << label << endl;
}
void jumpBRF(int label)
{
    fout << "BRF" << " LABEL" << label << endl;
    cout << "BRF" << " LABEL" << label << endl;
}
FILE* fp;
// 语法分析程序
int TESTparse()
{
    int es = 0;
    fin.open(Scanin);
    fout.open(Scanout);
    if (!fin.is_open())
    {
        printf("\n Can not open % S\n", Scanout);
        es = 10;
    }
}

```

```

}
if (es == 0)
    es = program();
printf("== Grammer prase result ==\n");
switch (es)
{
case 0:
    printf("Grammar parse success\n");
    break;
case 10:
    printf("failed to open %s ", Scanout);
    break;
case 1:
    printf("Unclosed Parentheses Error\n");
    break;
case 2:
    printf(" declaration Error\n");
    break;
case 3:
    printf("Missing Semicolon Error\n");
    break;
case 4:
    printf("Redefinition of variable\n");
    break;
case 5:
    printf("Missing Left Parenthesis\n");
    break;
case 6:
    printf("Undefinition of variable\n");
    break;
case 7:
    printf("\n");
    break;

case -1:
    printf("Terminal Symbol error\n");
}
fin.close();
return (es);
}
int program()
{

    Token token;

```

```

int es = 0;
fin >> token;
if (token.tokenType == "{")
{
    int es = declarationList();

    if (es > 0)
    {
#ifdef DEBUG
        std::cout << "Error occur in program!\n";
#endif
        return es;
    }
    es = statementList();
    if (es > 0)
    {
#ifdef DEBUG
        std::cout << "Error occur in program!\n";
#endif
        return es;
    }
    fin >> token;
    if (token.tokenType == "}")

        return es;
#ifdef DEBUG
    std::cout << "Error occur in program!\n";
#endif
    return 1;
}
else
{
#ifdef DEBUG
    std::cout << "Error occur in program! expected {\n";
#endif
    return 1;
}
}

int compoundStat()
{

    auto tempPosition = fin.tellg();
    Token leftBracketToken, rightBracketToken;

```

```

    fin >> leftBracketToken;

    if (leftBracketToken.tokenType != "{")
    {
        fin.seekg(tempPosition);

#ifdef DEBUG
        std::cout << "Error occur in compoundStat!\n";
#endif
        return -1;
    }
    int es = statementList();
    if (es)
    {
#ifdef DEBUG
        std::cout << "Error occur in compoundStat!\n";
#endif
        return es;
    }
    fin >> rightBracketToken;
    if (rightBracketToken.tokenType != "}")
    {
#ifdef DEBUG
        std::cout << "Error occur in compoundStat!\n";
#endif

        return 1;
    }
    return 0;
}
int statement()
{
    Token t;
    auto tmpPosBef = fin.tellg();
    fin >> t;
    fin.seekg(tmpPosBef);
    if (t.tokenType == "if")
        return ifStat();
    if (t.tokenType == "while")
        return whileStat();
    if (t.tokenType == "for")
        return forStat();
    if (t.tokenType == "read")
        return readStat();
}

```

```

    if (t.tokenType == "write")
        return writeStat();
    if (t.tokenType == "{")
        return compoundStat();
    return expressionStat();
}
int expressionStat()
{
    // std::cout << "enter expressionStat\n";
    int es = expression();
    if (es != 0)
    {
        if (es != -1)
        {
#ifdef DEBUG
            std::cout << "Error occur in expressionStat!\n";
#endif
        }
        return es;
    }
    fout << "POP" << endl;
    cout << "POP" << endl;
    Token t;
    fin >> t;
    if (t.tokenType != ";")
    {
#ifdef DEBUG
        std::cout << "Error occur in expressionStat!\n";
#endif
        return 3;
    }
    return 0;
}

int expression()
{
    int es = 0;
    auto begin = fin.tellg();
    Token t1;
    fin >> t1;

    if (t1.tokenType == "ID")
    {
        Token t2;

```

```

    fin >> t2;

    int addr = 0;
    variabletable.LookUpVariable(t1.tokenValue, addr);
    if (addr == -1)
    {
        std::cout << "[ERROR]"<<"row:"<<t1.row<<" col:"<<t1.column <<
" Undeclared variable " << t1.tokenValue << std::endl;
        return 6;
    }
    if (t2.tokenType == "=")
    {
        es = boolExpr();
        if (es > 0)
        {
#ifdef DEBUG
            std::cout << "Error occur in expression!\n";
#endif
            return es;
        }
        fout<<"STO "<<addr<<std::endl;
        cout<<"STO "<<addr<<std::endl;
    }
    else
    {
        fin.seekg(begin);
        es = boolExpr();
        if (es > 0)
        {
#ifdef DEBUG
            std::cout << "Error occur in expression!";
#endif
            return es;
        }
    }
    else
    {
        fin.seekg(begin);
        es = boolExpr();
    }
    return es;
}

```

```

int boolExpr()
{
    int es = additiveExpr();
    if (es != 0)
    {
        if (es != -1)
        {
#ifdef DEBUG
            std::cout << "Error occur in boolExpr!\n";
#endif
        }
        return es;
    }
    auto tmp = fin.tellg();
    Token t;
    fin >> t;

    if (t.tokenType == ">" || t.tokenType == "<" || t.tokenType == ">="
    || t.tokenType == "<=" || t.tokenType == "==" || t.tokenType == "!=")
    {

        int es1= additiveExpr();
        if (t.tokenType == ">")
        {
            fout << "GT" << endl;
            cout << "GT" << endl;
        }
        else if (t.tokenType == "<")
        {
            fout << "LES" << endl;
            cout << "LES" << endl;
        }
        else if (t.tokenType == ">=")
        {
            fout << "GE" << endl;
            cout << "GE" << endl;
        }
        else if (t.tokenType == "<=")
        {
            fout << "LE" << endl;
            cout << "LE" << endl;
        }
        else if (t.tokenType == "==")
    }

```

```

        {
            fout << "EQ" << endl;
            cout << "EQ" << endl;
        }
        else if (t.tokenType == "!=")
        {
            fout << "NOTEQ" << endl;
            cout << "NOTEQ" << endl;
        }
        return es1;
    }
    fin.seekg(tmp);
    return es;
}

int additiveExpr()
{
    int es = term();
    if (es != 0)
    {
        if (es != -1)
        {
#ifdef DEBUG
            std::cout << "Error occur in additiveExpr!\n";
#endif
        }
        return es;
    }
    auto tmp = fin.tellg();
    Token t;
    fin >> t;

    if (t.tokenType == "+" || t.tokenType == "-")
    {
        es = term();
        if (!term) return es;
        if (t.tokenType == "+")
        {
            fout << "ADD" << endl;
            cout << "ADD" << endl;
        }
        else
        {
            fout << "SUB" << endl;
            cout << "SUB" << endl;
        }
    }
}

```



```

    }
    return es;

}
fin.seekg(tmp);
return es;
}
int term()
{
    int es = factor();
    if (es != 0)
    {
        if (es != -1)
        {
#ifdef DEBUG
            std::cout << "Error occur in term!";
#endif
        }
        return es;
    }
    auto tmp = fin.tellg();
    Token t;
    fin >> t;
    if (t.tokenType == "*" || t.tokenType == "/")
    {
        int es=factor();
        if (!factor)return es;
        if (t.tokenType == "*")
        {
            fout << "MULT" << endl;
            cout << "MULT" << endl;
        }
        else
        {
            fout << "DIV" << endl;
            cout << "DIV" << endl;
        }
        return es;
    }

    fin.seekg(tmp);

```

```

        return es;
    }
    int factor()
    {
        auto begin = fin.tellg();
        Token t;
        fin >> t;

        if (t.tokenType == "(")
        {
            int es = expression();
            if (es > 0)
            {
#ifdef DEBUG
                std::cout << "Error occur in factor!" << t.row << " " <<
t.column << " \n";;
#endif
                return es;
            }
            fin >> t;
            if (t.tokenType != ")")
            {
#ifdef DEBUG
                std::cout << "Error occur in factor!" << t.row << " " <<
t.column << " \n";;
#endif
                return 1;
            }
            else return 0;
        }
        else if (t.tokenType == "ID")
        {
            int es = 0;
            variabletable.LookUpVariable(t.tokenValue, es);
            if (es == -1)
            {
                std::cout << "[ERROR]" << "row:" << t.row << " col:" <<
t.column << " Undeclared variable " << t.tokenValue << std::endl;
                return 6;
            }
            else
            {
                fout << "LOAD " << es << endl;
                cout << "LOAD " << es << endl;
            }
        }
    }
}

```

```

    }

    return 0;
}

else if (t.tokenType == "NUM")
{
    fout << "LOADI " << t.tokenValue << endl;
    cout << "LOADI " << t.tokenValue << endl;
    return 0;
}

fin.seekg(begin);

return -1;
}
int ifStat()
{
    auto begin = fin.tellg();
    Token t;
    fin >> t;
    if (t.tokenType != "if")
    {
#ifdef DEBUG
        std::cout << "Error occur in ifStat!" << "row: " << t.row << "
col:" << t.column << " \n";
#endif

        fin.seekg(begin);
        return -1;
    }
    fin >> t;
    if (t.tokenType != "(")
    {
#ifdef DEBUG
        std::cout << "Error occur in ifStat!" << "row: " << t.row << "
col:" << t.column << " \n";
#endif
        return 5;
    }
    int es = expression();
    if (es > 0)
    {
#ifdef DEBUG

```

```

        std::cout << "Error occur in ifStat!" << "row: " << t.row << "
col:" << t.column << " \n";
#endif
        return es;
    }
    fin >> t;
    if (t.tokenType != ")")
    {

#ifdef DEBUG
        std::cout << "Error occur in ifStat!" << "row: " << t.row << "
col:" << t.column << " \n";
#endif

        return 1;
    }
    int lable1 = getNewLabel();
    jumpBRF(lable1);
    es = statement();
    if (es > 0)
    {
#ifdef DEBUG
        std::cout << "Error occur in ifStat!" << "row: " << t.row << "
col:" << t.column << " \n";
#endif
        return es;
    }

    int lable2= getNewLabel();
    jumpBR(lable2);
    setLable(lable1);
    auto tmp = fin.tellg();
    fin >> t;
    if (t.tokenType == "else")
    {
        es = statement();
        if (es > 0)
        {
#ifdef DEBUG
            std::cout << "Error occur in ifStat!" << "row: " << t.row <<
" col:" << t.column << " \n";
#endif
            return es;
        }
    }

```

```

    }
    else
        fin.seekg(tmp);
        setLable(lable2);
        return 0;
}
int whileStat()
{
    auto begin = fin.tellg();
    Token t;
    fin >> t;
    if (t.tokenType != "while")
    {
#ifdef DEBUG
        std::cout << "Error occur in whileStat!" << "row: " << t.row << "
col:" << t.column << " \n";
#endif

        fin.seekg(begin);
        return -1;
    }

    int label1 = getNewLable();
    setLable(label1);

    fin >> t;
    if (t.tokenType != "(")
    {
#ifdef DEBUG
        std::cout << "Error occur in whileStat!" << "row: " << t.row << "
col:" << t.column << " \n";
#endif
        return 5;
    }
    int es = expression();
    if (es > 0)
    {
#ifdef DEBUG
        std::cout << "Error occur in whileStat!" << "row: " << t.row << "
col:" << t.column << " \n";
#endif

        return es;
    }
}

```

```

    }
    fin >> t;
    if (t.tokenType != ")")
    {
#ifdef DEBUG
        std::cout << "Error occur in whileStat!" << "row: " << t.row << "
col:" << t.column << " \n";
#endif

        return 1;
    }

    int lable2= getNewLabel();
    jumpBRF(lable2);

    es = statement();
    if (es > 0)
    {
#ifdef DEBUG
        std::cout << "Error occur in whileStat!" << "row: " << t.row << "
col:" << t.column << " \n";
#endif
        return es;
    }
    jumpBR(label1);
    setLable(lable2);
    return 0;
}
int forStat()
{
    // std::cout << "enter forStat" << " \n";
    auto begin = fin.tellg();
    Token t;
    fin >> t;
    if (t.tokenType != "for")
    {
        fin.seekg(begin);
        return -1;
    }
    fin >> t;
    if (t.tokenType != "(") return 5;

    int es;

```

```

    es = expression();
    if (es > 0) return es;
    fout<<"POP"<<endl;
    cout<<"POP"<<endl;
    fin >> t;
    if (t.tokenType != ";") return 3;

    int lable1= getNewLabel();
setLabel(lable1);

    es = expression();
    if (es > 0) return es;
    int lable2= getNewLabel();
    int lable3= getNewLabel();
    jumpBRF(lable2);
    jumpBR(lable3);
    fin >> t;
    if (t.tokenType != ";") return 3;

    int lable4= getNewLabel();
setLabel(lable4);

    es = expression();
    if (es > 0) return es;
    cout<<"POP"<<endl;
    fout<<"POP"<<endl;
    jumpBR(lable1);
    fin >> t;
    if (t.tokenType != ")") return 1;

    setLabel(lable3);
    es = statement();
    if (es > 0) return es;
    jumpBR(lable4);
    setLabel(lable2);

    return 0;
}
int writeStat()
{
    auto begin = fin.tellg();
    Token t;
    fin >> t;

```

```

    if (t.tokenType != "write")
    {
        fin.seekg(begin);
        return -1;
    }
    int es = expression();
    if (es > 0)
    {
        return es;
    }
    fin >> t;
    if (t.tokenType != ";")
    {
        return 3;
    }
    fout<<"OUT"<<endl;
    cout<<"OUT"<<endl;
    return 0;
}
int readStat()
{
    auto begin = fin.tellg();
    Token t;
    fin >> t;
    string variableName;
    if (t.tokenType != "read")
    {
        fin.seekg(begin);
        return -1;
    }
    fin >> t;
    if (t.tokenType != "ID")
    {
        return -1;
    }
    variableName = t.tokenValue;
    int row=t.row,col=t.column;
    fin >> t;

```



```

    if (t.tokenType != ";")
    {

        return 3;
    }
    int addr = 0;
    variabletable.LookUpVariable(variableName, addr);
    if (addr == -1)
    {
        std::cout << "[ERROR]" << "row:" << row << " col:" << col << "
Undeclared variable " << variableName << std::endl;
        return 6;
    }
    fout << "IN" << endl;
    cout << "IN" << endl;
    fout << "STO " << addr << endl;
    cout << "STO " << addr << endl;
    fout << "POP" << endl;
    cout << "POP" << endl;
    return 0;
}
int declarationStat()
{
    string variableName;
    Token t;
    auto tmpPos = fin.tellg();
    fin >> t;

    if (t.tokenType != "int")
    {

        fin.seekg(tmpPos);
        return -1;
    }
    fin >> t;
    if (t.tokenType != "ID")
    {

        return 2;
    }
    variableName = t.tokenValue;
    int row = t.row, col = t.column;
    fin >> t;
    if (t.tokenType != ";")

```

```

{

    return 2;
}
/* if(variabletable.symbolTable.find(variableName)!=
variabletable.symbolTable.end())
{
    std::cout<<"Redeclaration of variable "<< variableName
<<std::endl;
    return 4;
}
int siz = variabletable.symbolTable.size();
variabletable.symbolTable[variableName] = siz;*/
int es=variabletable.insertVariable(variableName);
if (es == -1)
{
    std::cout<<"[ERROR] "<<"row:"<<row<<" col:"<<col << "
Redeclaration of variable " << variableName << std::endl;
    return 4;
}
return 0;
}
int declarationListRemoveLeftRecursivion()
{
    int es = declarationStat();
    if (es == -1)
        return 0;
    else if (es > 0)
        return es;
    es = declarationListRemoveLeftRecursivion();
    return es;
}
int declarationList()
{
    //<declaration_list>-><declaration_list><declaration_stat> | ε
    //<declaration_list>->εR
    // R-><declaration_stat>R | ε
    int es = declarationListRemoveLeftRecursivion();
    if (es <= 0)
        return 0;
    return es;
}

int statementListRemoveLeftRecursivion()

```

```

{
    int es = statement();
    if (es == -1)
        return 0;
    else if (es > 0)
        return es;
    es = statementListRemoveLeftRecursion();
    return es;
}
int statementList()
{
    //<statement_list>-><statement_list><statement> | ε
    //<statement_list>->εR
    // R-><statement>R | ε
    int es = statementListRemoveLeftRecursion();
    if (es <= 0)
        return 0;
    return es;
}

// statement_list → statement | statement statement_list

```