设计文档2

使用方式

将testfile放置于根目录下，通过设置flag控制不同阶段的输出文件。

整体架构

```
public class Compiler {
    public static void main(String[] args) throws IOException {
        File file = new File("testfile.txt");
        if (!file.exists()) {
            System.out.println("Filename error");
            System.exit(0);
        }
        String content = TurnToFile.readFile(file);
        //词法
        LexicalAnalyzer lexicalAnalyzer = new LexicalAnalyzer(content);
        ArrayList<LexicalAnalyzerForm> lexicalAnalyzerForms =
lexicalAnalyzer.LexicalAnalyze();
        TurnToFile.LexicalToFile(false, lexicalAnalyzerForms, "output.txt");
        ETB.addAll(lexicalAnalyzer.getErrorTables());

        //语法
        GrammerAnalyzer grammerAnalyzer = new
GrammerAnalyzer(lexicalAnalyzerForms);

TurnToFile.GrammerToFile(false,grammerAnalyzer.grammerAnalyze(),"output.txt" );
        ETB.addAll(grammerAnalyzer.getErrorTables());

        //错误处理
        ETBSorter.ETBSort(ETB);//整合词法分析语法分析阶段建立的两个错误表
        TurnToFile.ErrorToFile(false,ETB,"error.txt");

        //代码生成
        Interpreter interpreter = new Interpreter(grammerAnalyzer.getCodelist());
        TurnToFile.PcodeToFile(true, interpreter.interpret(),"pcoderesult.txt");
    }
}

```

设计思路

1.IO工具

1.1错误表整合器ETBSorter

设计初本来不想添加这个整合器，将所有错误代码生成全放到语法分析阶段完成，但此前的语法分析阶段无法优雅地处理a类错误，故只能在词法分析阶段进行处理，由于词法分析和语法分析阶段都存在对错误表的写入，所以为了格式化输出，定义ETBSorter对所有错误表进行整合，ErrorTable的定义见2.2错误表建立部分。

```
public class ETBSorter {
    public static void ETBSort(ArrayList<ErrorTable> ETB){
        for (int i=0;i<ETB.size()-1;i++){
            for (int j=0;j<ETB.size()-1-i;j++){
                if (ETB.get(j).getLineNum()>ETB.get(j+1).getLineNum()){
                    ErrorTable tmp = new ErrorTable(0,"");
                    tmp.changeTable(ETB.get(j));
                    ETB.get(j).changeTable(ETB.get(j+1));
                    ETB.get(j+1).changeTable(tmp);
                }
            }
        }
    }
}
```

1.2输出工具TurnToFile

新增了错误表输出类ErrorToFile和最终代码输出类PcodeToFile

```
    public static void ErrorToFile(boolean flag, ArrayList<ErrorTable> res, String outFileName) throws IOException {
        if (flag) {
            StringBuilder buf = new StringBuilder();
            for (int i = 0; i < res.size(); i++) {
                if (i != res.size() - 1) {
                    buf.append(res.get(i).turnToFileFormat()).append("\n");
                } else {
                    buf.append(res.get(i).turnToFileFormat());
                }
            }
            File file = new File(outFileName);
            FileWriter fileWritter = new FileWriter(file.getName(), false);
            fileWritter.write(buf.toString());
            fileWritter.close();
        }
    }

    public static void PcodeToFile(boolean flag, ArrayList<String> res, String outFileName) throws IOException {
        if (flag) {
            StringBuilder buf = new StringBuilder();
```

```
            for (int i = 0; i < res.size(); i++) {
                    buf.append(res.get(i));
            }
            File file = new File(outFileName);
            FileWriter fileWritter = new FileWriter(file.getName(), false);
            fileWritter.write(buf.toString());
            fileWritter.close();
        }
    }
```

1.3最终代码解释器Interpreter

解释并生成最终代码的结果 Pcode指令及其具体定义详见第三部分

```
public class Interpreter {
    private int[] dstack = new int[100000];//定义运行栈
    private int BAddr = 0;//基地址
    private int at = 0;//当前Pcode指令指针
    private int sp = -1;//栈顶指针
    private ArrayList<Code> codelist = new ArrayList<>();//Pcode表

    public ArrayList<String> interpret(){
        int addr;
        ArrayList<String> res = new ArrayList<>();
        Scanner scanner = new Scanner(System.in);
        while (at < codelist.size()) {
            Code curCode = codelist.get(at);
            switch (curCode.getName()) {
                case "INT":
                case "DOWN":
                case "LOD":
                case "LODS":
                case "LDA":
                case "LDC":
                case "STOS":
                case "ADD"://+
                case "SUB"://-
                case "MUL"://*
                case "DIV":///
                case "MOD"://%
                case "MINU"://-1
                case "GET"://getint
                case "PRF"://print
                case "JTM"://jump to main
                case "CAL"://func call
                case "RET"://return
                case "RET_TO_END"://eof
                case "INT_L"://int label
                case "BGT"://>
```

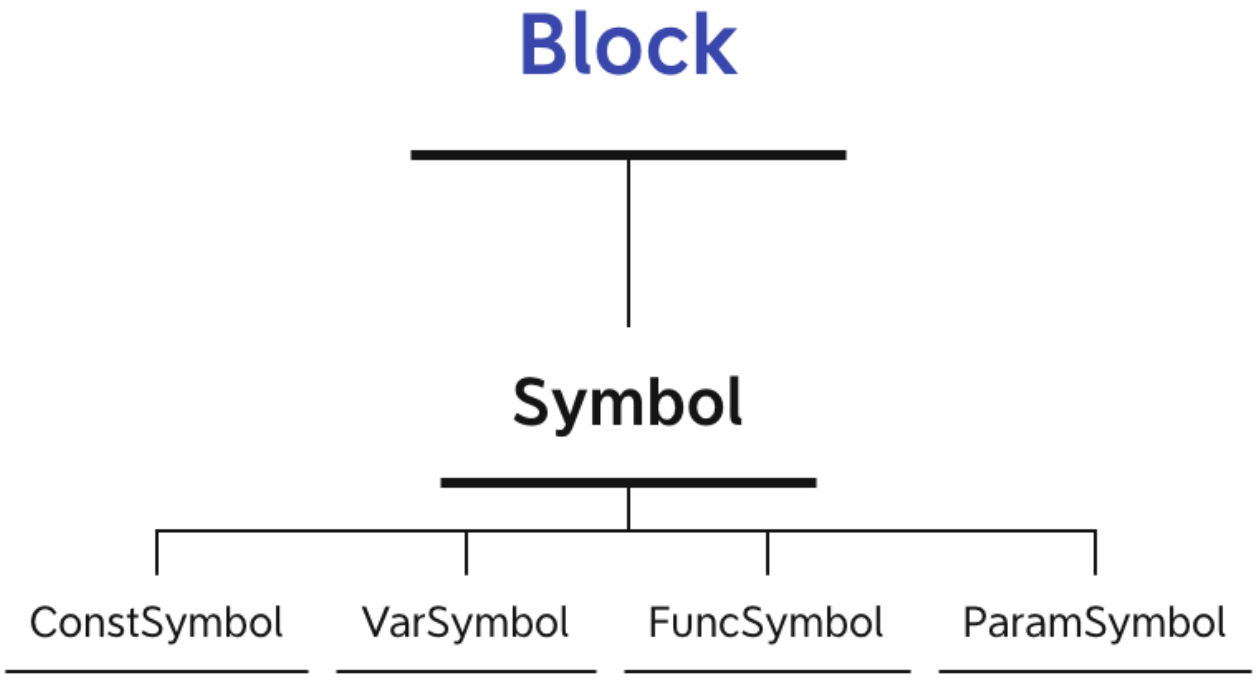
```
                case "BGE": //>=
                case "BLT"://<
                case "BLE"://<=
                case "BEQ"://==
                case "BNE"://!=
                case "BZT"://if 0 jump
                case "J":// jump
                case "JP0"://jump when 0
                case "JP1"://jump when 1
                case "NOT"://!a
                default:
            }
        }
        return res;
    }
}

```

2.错误处理

2.1符号表建立

为了方便将读到的Block和各个需要用到的常量变量存入符号表中，本项目符号表生成共定义了六种情况，并在语法分析执行过程中按顺序进行填充，他们的依存关系见下：



Block

```
public class Block {
    private String type;//负责区分全局块，函数块，普通块
    private ArrayList<Block> CBlock;//子块
    private Block FBlock;//父块
    private ArrayList<Symbol> SymbolTable;//块内符号表
    private int level;//块深度 全局为1 每读到一个新的Block就+1 退块就-1
    private boolean returnTk;//判断是否有返回值 为了处理return相关错误类型用
}
```

Symbol

```
public class Symbol {
    private String name;//名称
    private int dim;//void函数为-1，int函数和一般表达式为0，一维数组为1，二维数组为2
    private int dim1 = 0;//第一维大小或者是二维数组的第二维大小
    private int dim2 = 0;//二维数组的第一维大小
    private int address = 0;//地址
    private boolean isConst = false;//判断是否是常数
    private boolean isGlobal = false;//判断是否是全局定义
}
```

ConstSymbol

```
public class Const_symbol extends Symbol{
    private String type = "const";
    private List<Integer> values = new ArrayList<>();
    }
```

VarSymbol

```
public class Var_symbol extends Symbol{
    private String type = "var";
    }
```

FuncSymbol

```
public class Func_symbol extends Symbol{
    private String type = "func";
    private List<Param_symbol> params = new ArrayList<>();
    private int startCode;//记录函数初始位置 方便代码生成的函数跳转
    }
```

ParamSymbol

```
public class Param_symbol extends Symbol{
    private String type = "param";
    }
```

2.2错误表建立

```
public class ErrorTable {
    private String type;//错误类型
    private int lineNum;//发生行号
}
```

2.3新增语法分析返回值类型RecordDim

在处理函数调用的相关错误时，必须要对比实参表和形参表的所有参数类型和参数数量是否对应，所以在传递语法分析阶段的输出内容时，在进行FuncRparams的分析时还需要传递各个形参的dim（dim定义见2.1Symbol）。

```
public class RecordDim {
    private ArrayList<String> res;//语法分析的输出
    private int retDim;//实参维度
}
```

2.4新增语法分析返回值类型Record

对于常数表达式而言，等号右边的数值不需要存入符号表，需要直接计算出来并进行传递，将其存到等号左边的名称对应的符号表项中,为了方便值的传输，定义了新的返回值类型。

```
public class Record {
    private ArrayList<String> res;//语法分析的输出
    private int retValue;//计算出的数值
    }
```

2.5新增语法分析返回值类型RecordValue

原理同上，不过对于数组而言，要传递的值就不只一个了，所以再次定义了新的返回值类型。同时在实参的传递过程中，在讲最后的一整组实参传入UnaryExp中时也能用到这个类。

```
public class RecordValue {
    private List<String> res;//语法分析的输出
    private List<Integer> values;//一组计算出的数值
    }
```

3.代码生成

代码生成依然在语法分析阶段完成，这里先介绍方便代码生成的辅助类

3.1Pcode代码表CodeTable

```
public class Code{
    private String name;//Pcode指令名
    private int level;//所处层次
    private int addr;//对应地址
    private String print;//print中Strcon内容
    private Label label;//跳转目标
    private int type = 0;//指令类型
}
```

```
public class Label {
    private int point = 0;//跳转目标
}
```

3.2Pcode指令定义

指令名	指令描述
INT x	栈顶指针上移x
DOWN x	栈顶指针下移x
LOD x y	从相对位置为y处查询内容并存入栈顶 x为0或1 0表示绝对地址中进行查询 1表示相对地址中进行查询
LODS	从栈顶指针的地址中查询内容并存入栈顶
LDA x y	从相对位置为y处的地址存入栈顶 x为0或1 0表示绝对地址中进行查询 1表示相对地址中进行查询
LDC	将值存入栈顶
STOS	从栈顶指针的地址中查询内容并存入次栈顶的地址中 退两次栈
ADD	栈顶和次栈顶相加并存入栈顶
SUB	栈顶和次栈顶相减并存入栈顶
MUL	栈顶和次栈顶相乘并存入栈顶
DIV	栈顶和次栈顶相除并存入栈顶
MOD	栈顶和次栈顶相模并存入栈顶
MINU	栈顶取负

GET	从终端获得输入
PRF	输出至终端
JTM x	跳转到main函数 x为main函数的前一条指令的序列地址
CAL x	函数调用 x为调用函数的前一条指令的序列地址
RET	函数返回
RET_TO_END	主函数返回
INT_L x	栈指针上移Lable个位置 Lable由函数块决定 如其中包含两个变量 Lable就是5(3+2) 对每个函数块都会预留三个位置分别存返回值 返回值基地址 返回值指令序列号
BGT	退一次栈 栈顶和退栈前的栈顶比较 若存在大于关系(>)则当前栈顶存入1 否则存入0
BGE	退一次栈 栈顶和退栈前的栈顶比较 若存在大于等于关系(>=)则当前栈顶存入1 否则存入0
BLT	退一次栈 栈顶和退栈前的栈顶比较 若存在小于关系(<)则当前栈顶存入1 否则存入0
BLE	退一次栈 栈顶和退栈前的栈顶比较 若存在小于等于关系(<=)则当前栈顶存入1 否则存入0
BEQ	退一次栈 栈顶和退栈前的栈顶比较 若存在等于等于关系(==)则当前栈顶存入1 否则存入0
BNE	退一次栈 栈顶和退栈前的栈顶比较 若存在不等关系(!=)则当前栈顶存入1 否则存入0
BZT x	如果栈顶为0 则跳转到指令序列为x处 退一次栈(IFTK WHILETK)
J x	无条件跳转到x处
JP0 x	如果栈顶为0 则跳转到指令序列为x处 不退栈(LAND TK)
JP1 x	如果栈顶为1 则跳转到指令序列为x处 不退栈(LOR TK)
NOT	栈顶进行非运算

3.3代码生成实例

3.3.1常量变量定义

```
const int a = 10;
int b = 10;
int f;
const int c[1] = {1};
const int d[2][2] = {{1,2},{3,4}};

0  INT  1
1  LDA  0  0
2  LDC  10
3  STOS//const int a = 10;
4  INT  1
5  LDA  0  1
```

```
6  LDC  10
7  STOS//int b = 10;
8  INT  1
9  LDA  0  2
10  LDC  0
11  STOS//int f;
12  INT  1
13  LDA  0  3
14  LDC  1
15  STOS//const int c[1] = {1};
16  INT  1
17  LDA  0  4
18  LDC  1
19  STOS//const int d[2][2] = {{1,2},{3,4}}; 1
20  INT  1
21  LDA  0  5
22  LDC  2
23  STOS//const int d[2][2] = {{1,2},{3,4}}; 2
24  INT  1
25  LDA  0  6
26  LDC  3
27  STOS//const int d[2][2] = {{1,2},{3,4}}; 3
28  INT  1
29  LDA  0  7
30  LDC  4
31  STOS//const int d[2][2] = {{1,2},{3,4}}; 4
```

栈顶指针初始值设置为-1，所以每次先将栈顶指针上移1(INT 1)，将相对地址存到栈顶(LDA 0 0)，再将赋值内容存到栈顶(LDC)，将栈顶内容存到次栈顶所示地址中然后退两次栈(STOS)即可。对于没有进行赋值操作的语句，默认将其赋值为0即可。

对于数组 只需要进行顺序储值即可，其基本原理同常量变量的相关定义。

3.3.2运算操作

```
1 + 2;

1 LDC 1
2 LDC 2
3 ADD
```

对于加减乘除模这一类二院的运算操作都只需要三步，将第一个操作数入栈（LDC 1），将第二个操作数入栈（LDC 2），进行相关运算（ADD）。

3.3.3读写操作

```
int a ;
a = getint();
printf("a is %d",a);

1  INT  1
2  LDA  0  0
3  LDC  0
4  STOS
5  LDA  0  0
6  GET
7  STOS
8  LDA  0  0
9  LODS
10  PRF  "a is %d"
```

对读操作和写操作，都是先读取地址，然后对这个地址里的值进行读写即可。

3.3.4左值表达式

```
const int a = 10;
int b[2][2] = {{1,2},{3,4}};
int main(){
    int d, c;
    d = a;
    c = b[1][1];
    return 1;
}

//一般传值
30  LDA  0  0//d address
31  LDA  1  0//a address
32  LODS// load a value
33  STOS//store a value
//数组传值
34  LDA  0  1//c address
35  LDC  1//b dim1
36  LDC  2//b dim2
37  MUL// calculate b[1][1]'s address
38  LDC  1
39  LDA  1  1//load b address
40  ADD//calculate b baseAddr+offset
41  ADD//calculate b baseAddr+offset
42  LODS//load from b baseAddr+offset
43  STOS//store value
```

对于左值表达式，一般的传值方式先将等号左边的地址读到栈顶(LDA 0 0)，再将等号右边的地址读到栈顶(LDA 1 0)，取出当前栈顶的值(LODS)，最后将栈顶内容存到次栈顶所示地址中然后退两次栈(STOS)即可。

对于数组传参，需要记录数组头的基地址，并通过数组的维度计算出需要用到的偏移量，最后进行储值，详细流程见上文34-43行Pcode

3.3.5条件语句

```
int main(){
    int a,b,c,d;
    a = 1;
    if(a){
        b = 2;
    }else{
        b = 3;
    }
}

21  LDA  0  0//load a address
22  LODS//load a value
23  BZT  28//if 0 jumpto Pcode28(进入else块)
24  LDA  0  1//load b address
25  LDC  2//load 2
26  STOS// b = 2
27  J  31//jumpto Pcode31(离开条件语句块)
28  LDA  0  1
29  LDC  3
30  STOS//b = 3
```

对于条件语句，先对cond块内的部分进行计算，得出最终的结果存入栈顶，如果栈顶为1就进入if块，如果为0就进入else块(BZT 28)，如果没有else块就直接出条件语句块、

3.3.6循环语句

```
int main(){
    int a,b,c,d;
    a = 3;
    while(a > 0){
        a = a - 1;
    }
}


18  LDA  0  0//load a address
19  LDC  3//load 3
20  STOS//a = 3
21  LDA  0  0//load a address
22  LODS//load a value
23  LDC  0//load 0
24  BGT//if a > 0?1:0
25  BZT  33//jumpto Pcode33
26  LDA  0  0//load a address
```

```
27  LDA  0  0//load a address
28  LODS//load a value
29  LDC  1//load 1
30  SUB//a - 1
31  STOS//a = a-1
32  J  21//jumpto Pcode21
```

对于循环语句，也是对循环语句块内的条件进行判断，如果为真就进入循环快否则跳出，并在每次循环块的结尾再次跳入循环快即可。

3.3.7逻辑表达式

```
int main(){
    int a,b,c;
    a = 1;
    b = 0;
    c = 2;
    if(a && b && c){
        b = 1;
    }
    if(a || b || c){
        b = 2;
    }
}

23  LDA  0  0
24  LODS//load a value
25  JP0  29//短路求值 a为0直接跳转
26  DOWN  1
27  LDA  0  1
28  LODS//load b value
29  JP0  33//短路求值 b为0直接跳转
30  DOWN  1
31  LDA  0  2
32  LODS
33  BZT  37//if块判断
34  LDA  0  1
35  LDC  1
36  STOS
37  LDA  0  0
38  LODS
39  JP1  43//短路求值 a为1直接跳转
40  DOWN  1
41  LDA  0  1
42  LODS
43  JP1  47//短路求值 b为1直接跳转
44  DOWN  1
45  LDA  0  2
46  LODS
```

```
47  BZT  51//if块判断
48  LDA  0  1
49  LDC  2
50  STOS
```

对于逻辑表达式，只需要每读入一个表达值就判断他的值，就能直接进行跳转操作并实现短路求值了。

3.3.8函数调用

```
int add(int a,int b){
    return a+b;
}
int main(){
    int a,b;
    a = 1;
    b = 0;
    b = add(a,b);
    return 1;
}

0  JTM  10
1  INT_L  5//函数块内部 3个基本值(返回值 基地址 返回值指令序列)和2个参数
2  LDA  0  0
3  LDA  0  3
4  LODS
5  LDA  0  4
6  LODS
7  ADD
8  STOS
9  RET
10  INT_L  2
25  LDA  0  1//load b
26  INT  3
27  LDA  0  0//load a address
28  LODS//load a value
29  LDA  0  1//load b address
30  LODS//load b value
31  DOWN  5
32  CAL  1//cal function
33  STOS//sto return
```

对于函数调用，在Pcode定义时要求的每进入函数块就申请三个空间分别存返回值、返回值地址、返回值指令序列就有用了，通过cal跳转到函数中进行计算，最后栈顶一定是返回值大小，直接存即可。