设计文档

温舒盈　　71066002

代码仓库：https://github.com/suin14/Compiler-2023

编译器介绍

这是一篇基于2023-秋学年编译技术实验部分编译器作业所编写的实验报告。实现的编译器选择了生成 **PCODE**，代码部分选择 **Java** 语言进行编写。编译器的总体设计可以划分为 **词法分析**、**语法分析**、**错误处理**和**代码生成** 四个阶段。接下来的报告中将会通过大量的代码来辅助解释对编译器的设计。

先展示一下，此编译器的各个接口：

```
public class Analyser {

    public Analyser() throws IOException {
        //File inputFile = new File("input.txt"); Scanner input = new
Scanner(inputFile);
        Scanner input = new Scanner(System.in);
        LexicalAnalyser lexicalAnalyser = new LexicalAnalyser();
        GrammaticalAnalyser grammaticalAnalyser = new
GrammaticalAnalyser(lexicalAnalyser.getTokens());

        handleErrors(grammaticalAnalyser); // 错误处理

        if(!grammaticalAnalyser.hasErrors()) {
            executePCode(grammaticalAnalyser, input); // 生成 PCode
        }
    }

    private void handleErrors(GrammaticalAnalyser grammaticalAnalyser) throws
IOException {
        grammaticalAnalyser.printError(new
FileProcessor("testfile.txt","error.txt").getWriter());
    }

    private void executePCode(GrammaticalAnalyser grammaticalAnalyser, Scanner
input) throws IOException {
        Executor pCodeExecutor = new Executor(grammaticalAnalyser.getCodes(),
input);
        pCodeExecutor.run();
        pCodeExecutor.print();
    }
}
```

词法分析

作业要求：设计并实现词法分析程序，从源程序中识别出单词，记录其单词类别和单词值。输入的被编译源文件统一命名为 `testfile.txt`；输出的结果文件统一命名为 `output.txt`，结果文件中每行按如下方式组织：

> 单词类别码 单词的字符/字符串形式(中间仅用一个空格间隔)

词法分析中，单词的类别码统一定义如下：

单词名称	类别码	单词名称	类别码	单词名称	类别码	单词名称	类别码
Ident	IDENFR	!	NOT	*	MULT	=	ASSIGN
IntConst	INTCON	&&	AND	/	DIV	;	SEMICN
FormatString	STRCON	||	OR	%	MOD	,	COMMA
main	MAINTK	for	FORTK	<	LSS	(	LPARENT
const	CONSTTK	getint	GETINTTK	<=	LEQ	)	RPARENT
int	INTTK	printf	PRINTFTK	>	GRE	[	LBRACK
break	BREAKTK	return	RETURNTK	>=	GEQ	]	RBRACK
continue	CONTINUETK	+	PLUS	==	EQL	{	LBRACE
if	IFTK	-	MINU	!=	NEQ	}	RBRACE
else	ELSETK	void	VOIDTK				

FileProcessor

首先，需要实现对输入输出文件的管理。我主要通过 **transferToCode** 来实现读取 `testfile.txt` 中的内容，并将内容转换为一个包含换行符 `\n` 的长字符串。在这段代码中，我使用了 `FileInputStream` 和 `Scanner` 来逐行读取文件内容，然后使用 `StringJoiner` 将每一行连接成一个长字符串，并用换行符分隔。

```
package Lexer;

import java.io.*;
import java.util.Scanner;
import java.util.StringJoiner;

public class FileProcessor {
    private final String code;
    private final FileWriter writer;

    public FileProcessor(String inputFile, String outputFile) throws IOException
{
        code = transferToCode(inputFile);
        writer = new FileWriter(outputFile);
    }

    public static String transferToCode (String filename) throws IOException {
        InputStream stream = new FileInputStream(filename);
        //读取一个文件，并按行转化为带有'\n'的长字符串
        Scanner scanner = new Scanner(stream);
        StringJoiner stringJoiner = new StringJoiner("\n");
        //如果scanner的位置不是被读取文件的最后一行
        while (scanner.hasNextLine()) {
```

```
            stringJoiner.add(scanner.nextLine()); //读取下一行
        }
        scanner.close();
        stream.close();
        return stringJoiner.toString();
    }

    ......
}
```

LexicalAnalyser

接着，通过 **getChar** 不断地读入字符，并对读入的字符做词法分析，直至读到null为止。

```
    private Character getChar() {   // 读取一个字符
        if (index < code.length()) {
            char current = code.charAt(index);
            if (current == '\n') {
                line++;
            }
            index++;
            return current;
        } else {
            return null;
        }
    }

    ......

    private void analyse() {
        Character current;
        while ((current = getChar()) != null) {
            switch (current) {
                case ' ', '\r', '\t' -> { //过滤不必要的空格
                }
                case '+', '-', '*', '%', '(', ')', '[', ']', '{', '}', ',', ';'
-> token.add(new Token(current, line));
                case '/' -> analyseSlash();
                case '=', '<', '>', '!' -> analyseRelation(current);
                case '"' -> analyseQuot();
                case '&', '|' -> analyseLogic(current);
                default -> {
                    if (Character.isDigit(current)) {
                        analyseDigit(current);
                    } else if (Character.isLetter(current) || current == '_') {
                        analyseLetter(current);
                    }
                }
            }
        }
    }

    ......
```

开始分析读入的内容。以读入 `/` 为例，

首先需要判断读入的 `/` 是输入单行注释 `//`、多行注释 `/**/` 还是单纯只是一个 `/` 字符。

于是我们进入 **analyseSlash** 分析，

如果 `/` 后接的下一个读入符号也是 `/` ，那代表接下来读入的内容是属于单行注释 `//`，多行注释`/**/` 同理。

如果不属于上述两种情况，则代表读入了一个 `/` 字符，在 **token**表（token将用作下一阶段语法分析的输入）中添加 `/` 字符及其所在行数（方便在之后处理报错），同时因为在 **analyseSlash** 的第一行读取了下一个字符来做判定，所以我们要回退一步 **backup** , 避免漏读这个字符。

```
    private void analyseSlash() {
        Character current = getChar();

        if (current == '/') {
            // 单行注释
            while (current != null && current != '\n') {
                current = getChar();
            }
        } else if (current == '*') {
            // 多行注释
            while (true) {
                while (current != null && current != '*') {
                    current = getChar();
                }
                if (current == null) {
                    return;
                }
                current = getChar();
                if (current == '/') {
                    return;
                }
            }
        } else {
            token.add(new Token("/", line)); // 字符'/'
            backup(); // 回退
        }
    }

    ......

    private void backup() {  // 回退，避免漏读
        index--; // 当前位置的标识，类比于指针
        char current = code.charAt(index);
        if (current == '\n') {
            line--;
        }
    }
```

其他字符的分析类似于此，比较特别的有 **analyseQuot**（如何读取 `\n` 换行符）及 **analyseLetter**（如何识别关键字）。其中，我将关键字及其对应的类别码种类（从 **nodeMap** 获取）、内容（如果有）和行数保存在 名为**token** 的哈希表中。

```
        private void analyseQuot() {
        Character current;
        int flag = 0;
        StringBuilder buffer = new StringBuilder(); //保存读取到的内容
        while ((current = getChar()) != null) {
            if (current == '"') { // 寻找到结束的‘"’
                if(flag == 1) {
                    buffer.append("\\");
                }
                token.add(new Token(String.valueOf(Word.STRCON), "\"" +
String.valueOf(buffer) + "\"", line));
                return;
            } else {
                if (current == '\\') {
                    flag = 1;
                } else {
                    if (flag == 1 && current == 'n') {
                        buffer.append("\n");
                    } else if (flag == 1 && current != 'n') {
                        buffer.append("\\");
                        buffer.append(current);
                    } else {
                        buffer.append(current);
                    }
                    flag = 0;
                }
            }
        }
    }

    ......

    private void analyseLetter(char pre) {
        StringBuilder builder = new StringBuilder(String.valueOf(pre));
        Character current;
        while ((current = getChar()) != null) {
            if (Character.isLetter(current) || current == '_' ||
Character.isDigit(current)) {
                builder.append(current);
            } else {
                backup();
                if (new NodeMap().isNode(builder.toString())) {   // 先比对是不是关
键字
                    token.add(new Token(builder.toString(),line));
                } else {
                    token.add(new Token((String.valueOf(Word.IDENFR)),
builder.toString(),line));  // 变量名
                }
                return;
            }
        }
    }
```

Token

对于 **Token** 表，主要保存关键字、内容（如果有）、种类、行数四种信息，其中种类 `type` 从 **NodeMap** 表中获取。

```
public class Token {
    private String identification;
    private final String content;
    private final String type;
    private final int line;

    public Token(String identification, int line) {    // 标识符
        this.identification = identification;
        this.type = new NodeMap().getType(this.identification);  // type从NodeMap
中获取
        this.content = this.identification; // 读取到的内容
        this.line = line;
    }

    ......
}
```

NodeMap

NodeMap 用哈希表实现，映射 `token` 的字符串及其对应的类型：

```
public class NodeMap {
    private final HashMap<String, String> nodeMap;

    public NodeMap() {
        nodeMap = new HashMap<>();
        nodeMap.put("main", String.valueOf(Word.MAINTK));
        nodeMap.put("const", String.valueOf(Word.CONSTTK));
        nodeMap.put("int", String.valueOf(Word.INTTK));
        nodeMap.put("break", String.valueOf(Word.BREAKTK));
        ......
```

语法分析

作业要求：设计并实现语法分析程序，从源程序中识别出相应的语法成分，输入输出及处理要求如下：（1）需按文法规则，用递归子程序法对文法中定义的语法成分进行分析；（2）输出信息包含如下两种信息：

> 1）按词法分析识别单词的顺序，按行输出每个单词的信息（要求同词法分析作业，对于预读的情况不能输出）。
> 形如： 单词类别码 单词的字符/字符串形式(中间仅用一个空格间隔)
> 2）在文法中出现（除了<BlockItem>, <Decl>, <BType> 之外）的语法分析成分分析结束前，另起一行输出当前语法成分的名字，形如“<Stmt>”（注：未要求输出的语法成分仍需要进行分析，但无需输出）

终结符 `token` 需要被输出，但是是输出其词法成分，语法分析比起词法分析需要额外输出的是 **若干个词法的 token 组成的非终结符成分，并通过 `<非终结符>` 的形式给出。**

递归下降语法分析

这部分主要根据语法规则，判断从词法分析器获得的一个个`token`属于谁，然后进行对应的分析，其中，我用一个变量 (Token) `current` 来标识当前正在分析的`token`是哪一个。

举例来说，

对于语法规则：

```
ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';'
```

根据提供的逻辑建立相应的分析程序，代码中的 **getToken** 将会读入 `current` 的下一个`token`，而 **getNext** 代表着 `current` 后的下一个 `token` , **typeIs** 是 **Token** 类提供的一个公共方法，用于判断当前 `token` 的类型，**Word** 是一个 `enum` 类，包含了作业要求的关键字。

假如 `const` 后跟着的 `token` 属于 `INTTK` 类型，则进入对 `BType` 的分析中，否则则报错，代表读入的 `token` 串不符合语法规则，最后，用名为 `grammar` 的哈希表保存语法分析器识别出的语法成分（用于做后面代码生成阶段的输入）。

```
    // ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';'
    private void analyseConstDecl() {
        getToken(); // const
        if (getNext().typeIs(String.valueOf(Word.INTTK))){
            analyseBType(); // Btype
        } else {
            error();
        }
        analyseConstDef(); // ConstDef
        Token nextToken = getNext();
        while (nextToken.typeIs(String.valueOf(Word.COMMA))) {
            getToken(); // ,
            analyseConstDef(); // ConstDef
            nextToken = getNext();
        }
        checkSemicn(); // 错误处理：检查有没有;
        grammar.add("<ConstDecl>");
    }
```

在语法分析部分中，对于扩充BNF的[A] (A出现一次或0次)及{A}（A出现多次或0次）的处理有点类似于理论课上的正规式转NFA的思路，

用 `if` 来处理[A]，如果读到了A，则必须把A的所有式子读完，才可以读下一个，例如下面代码处理 `'if' '(' Cond ')' Stmt [ 'else' Stmt ]` ,假如在 `stmt` 后读到了 `ELSETK` , 则必须读完 `else Stmt` 才算是完成这条语法规则的分析。

```
......
    // 'if' '(' Cond ')' Stmt [ 'else' Stmt ]
    else if (nextToken.typeIs(String.valueOf(Word.IFTK))) {
            ......
            getToken(); // if
            getToken(); // (
```

```
            analyseCond(String.valueOf(Word.IFTK)); // Cond
            if (!getNext().typeIs(String.valueOf(Word.RPARENT))) {
                error("j"); //缺少右小括号')'
            } else {
                getToken(); // )
            }
            ......
            analyseStmt(); // Stmt
            nextToken = getNext();
            ......
            if (nextToken.typeIs(String.valueOf(Word.ELSETK))) {
                getToken(); // else
                analyseStmt(); // Stmt
            }
            ......
        }
```

而{A}则通过`while`处理，原理和上一个相似，直到不再读到A才进入下一个式子的分析。

```
    // Block → '{' { BlockItem } '}'
    private boolean analyseBlock(boolean fromFunc) {
        getToken(); // {
        if (!fromFunc) {
            addArea();
        }
        Token nextToken = getNext();
        boolean isReturn = false;
        while (nextToken.typeIs(String.valueOf(Word.CONSTTK)) ||
nextToken.typeIs(String.valueOf(Word.INTTK)) || nextToken.typeSymbolizeStmt()) {
            if (nextToken.typeIs(String.valueOf(Word.CONSTTK)) ||
nextToken.typeIs(String.valueOf(Word.INTTK))) {
                isReturn = analyseBlockItem(); // BlockItem
            } else {
                isReturn = analyseStmt();
            }
            nextToken = getNext();
        }
        getToken(); // }
        if (!fromFunc) {
            removeArea();
        }
        grammar.add("<Block>"); // 语句块
        return isReturn;
    }
```

有一条特殊的语法规则

```
AddExp → MulExp | AddExp ('+' | '−') MulExp
```

它具有左递归，于是根据理论课所学的知识（扩充的BNF文法），可以将其改写来消除左递归：

```
AddExp → MulExp { ('+' | '−') MulExp }
```

最后代码如下，

```
    private int analyseAddExp(ArrayList<Token> exp) {
        int intType = 0;
        Exps exps = divideExp(exp, new ArrayList<>
(Arrays.asList(String.valueOf(Word.PLUS), String.valueOf(Word.MINU))));
        int j = 0;
        for (ArrayList<Token> exp1 : exps.getTokens()) {
            intType = analyseMulExp(exp1); // { ('+' | '-') MulExp }
            if (j > 0) {
                if (exps.getSymbols().get(j -
1).typeIs(String.valueOf(Word.PLUS))) {
                    addCode(Operator.ADD);
                } else if (exps.getSymbols().get(j -
1).typeIs(String.valueOf(Word.MINU))){
                    addCode(Operator.SUB);
                }
            }
            grammar.add("<AddExp>");
            if (j < exps.getSymbols().size()) {
                grammar.add(exps.getSymbols().get(j++).toString()); // MulExp
            }
        }
        return intType;
    }
```

其中，**divideExp** 是一个方法， 它初始化了一些变量，包括 `exps` 用于存储划分后的表达式列表，`exp1` 用于暂时存储单个表达式，`symboltable` 用于存储特定符号。还有一些标志变量，比如 `unaryFlag` 和 `flag1`、`flag2`。

通过循环遍历 `exp` 中的 `Token`，根据特定的条件进行划分：

- 根据括号和方括号的数量进行匹配，并在合适的位置切分表达式。
- 对特定的符号进行处理，如加号、减号、非运算符等，将它们作为单独的表达式或添加到 `symboltable` 中。
- 根据 `unaryFlag` 的状态，判断是否需要将特定的运算符添加到当前的表达式中。

最后，将最后一个表达式添加到 `exps` 中，并将 `exps` 和 `symboltable` 作为参数构造了一个 `Exps` 对象，然后返回这个对象。

翻译成人话就是：这是一个生成表达式的方法。

```
private Exps divideExp(ArrayList<Token> exp, ArrayList<String> symbol) {
        ArrayList<ArrayList<Token>> exps = new ArrayList<>();
        ArrayList<Token> exp1 = new ArrayList<>();
        ArrayList<Token> symboltable = new ArrayList<>();
        boolean unaryFlag = false;
        int flag1 = 0;
        int flag2 = 0;
        for (Token nextToken : exp) {
            if (nextToken.typeIs(String.valueOf(Word.LPARENT))) {
                flag1++;
            }
            if (nextToken.typeIs(String.valueOf(Word.RPARENT))) {
                flag1--;
```

```
            }
            if (nextToken.typeIs(String.valueOf(Word.LBRACK))) {
                flag2++;
            }
            if (nextToken.typeIs(String.valueOf(Word.RBRACK))) {
                flag2--;
            }
            if (symbol.contains(nextToken.getType()) && flag1 == 0 && flag2 ==
0) {
                // UnaryOp
                if (nextToken.typeIs(String.valueOf(Word.PLUS)) ||
nextToken.typeIs(String.valueOf(Word.MINU)) ||
nextToken.typeIs(String.valueOf(Word.NOT))) {
                    if (!unaryFlag) {
                        exp1.add(nextToken);
                        continue;
                    }
                }
                exps.add(exp1);
                symboltable.add(nextToken);
                exp1 = new ArrayList<>();
            } else {
                exp1.add(nextToken);
            }
            unaryFlag = nextToken.typeIs(String.valueOf(Word.IDENFR)) ||
nextToken.typeIs(String.valueOf(Word.RPARENT)) ||
nextToken.typeIs(String.valueOf(Word.INTCON))
                    || nextToken.typeIs(String.valueOf(Word.RBRACK));
        }
        exps.add(exp1);
        return new Exps(exps, symboltable);
    }
```

Exps

构造函数 `Exps()` 接受两个 `ArrayList` 对象作为参数，并将它们分别赋值给 `tokens` 和 `symboltable` 成员变量。并提供两个公共方法：

- `getTokens()` 返回 `tokens` 成员变量，它包含了一个 `ArrayList` 的 `ArrayList`，即二维的 `Token` 列表。
- `getSymbols()` 返回 `symboltable` 成员变量，它包含了一个 `ArrayList`，存储了 `Token` 对象。

`Exps` 类用于在语法分析器中存储和处理 `tokens` 和 `symboltable` 相关的信息。`tokens` 存储了多个 token 的列表，而 `symboltable` 则存储了符号表的信息。

```
package Parser;
import Lexer.Token;

import java.util.ArrayList;

public class Exps {
    private final ArrayList<ArrayList<Token>> tokens;
    private final ArrayList<Token> symboltable;
```

```
    public Exps(ArrayList<ArrayList<Token>> tokens, ArrayList<Token>
symboltable) {
        this.tokens = tokens;
        this.symboltable = symboltable;
    }

    public ArrayList<ArrayList<Token>> getTokens() {
        return tokens;
    }

    public ArrayList<Token> getSymbols() {
        return symboltable;
    }
}
```

错误处理

作业要求：根据给定的文法设计并实现错误处理程序，能诊察出常见的语法和语义错误，进行错误局部化处理，并输出错误信息。输出的错误信息要求如下：（1）包含如下两种信息：错误所在的行号 错误的类别码（行号与类别码之间只有一个空格，类别码严格按照表格中的小写英文字母）

其中错误类别码按下表中的定义输出，行号从1开始计数，**从小到大的顺序输出至** `error.txt`：

错误类型	错误类别码	解释	对应文法及出错符号 （... 表示省略该条规则后续部分）
非法符号	a	格式字符串中出现非法字符报错行号为 <**FormatString**> 所在行数。	<FormatString> → ""{<Char>}""
名字重定义	b	函数名或者变量名在**当前作用域**下重复定义。注意，变量一定是同一级作用域下才会判定出错，不同级作用域下，内层会覆盖外层定义。报错行号为 <**Ident**> 所在行数。	<ConstDef>→<Ident> ... <VarDef>→<Ident> ... <Ident> ... <FuncDef>→<FuncType><Ident> ... <FuncFParam> → <BType> <Ident> ...
未定义的名字	c	使用了未定义的标识符报错行号为 <**Ident**> 所在行数。	<LVal>→<Ident> ... <UnaryExp>→<Ident> ...
函数参数个数不匹配	d	函数调用语句中，参数个数与函数定义中的参数个数不匹配。报错行号为函数调用语句的**函数名**所在行数。	<UnaryExp>→<Ident>'('[<FuncRParams>]')'
函数参数类型不匹配	e	函数调用语句中，参数类型与函数定义中对应位置的参数类型不匹配。报错行号为函数调用语句的**函数名**所在行数。	<UnaryExp>→<Ident>'('[<FuncRParams>]')'
无返回值的函数存在不匹配的return语句	f	报错行号为 '**return**' 所在行号。	<Stmt>→'return' {['<Exp>']};'
有返回值的函数缺少return语句	g	只需要考虑函数末尾是否存在return语句，**无需考虑数据流**。报错行号为函数**结尾的**'}' 所在行号。	<FuncDef> → <FuncType> <Ident> '(' [<FuncFParams>] ')' <Block> <MainFuncDef> → 'int' 'main' '(' ')' <Block>
不能改变常量的值	h	<LVal>为常量时，不能对其修改。报错行号为 <**LVal**> 所在行号。	<Stmt>→<LVal>'=' <Exp>';' <Stmt>→<LVal>'=' 'getint' '(' ')' ';'
缺少分号	i	报错行号为分号**前一个非终结符**所在行号。	<Stmt>,<ConstDecl>及<VarDecl>中的';'
缺少右小括号')'	j	报错行号为右小括号**前一个非终结符**所在行号。	函数调用(<UnaryExp>)、函数定义(<FuncDef>)及<Stmt>中的')'
缺少右中括号']'	k	报错行号为右中括号**前一个非终结符**所在行号。	数组定义(<ConstDef>,<VarDef>,<FuncFParam>)和使用(<LVal>)中的']'
printf中格式字符与表达式个数不匹配	l	报错行号为 '**printf**' 所在行号。	<Stmt> →'printf''('<FormatString>{,<Exp>}')';'
在非循环块中使用break和continue语句	m	报错行号为 '**break**' 与 '**continue**' 所在行号。	<Stmt>→'break';' <Stmt>→'continue';'

首先，考虑到错误处理的两大类型：**语义不相关**和**语义相关**，选择采用将其耦合入递归下降语法分析的过程中的解决方案。同时，由于其中语义相关的错误需要符号表的支持（如名字重定义等错误），在这一阶段部分地实现符号 `Symbol` 和符号表 `SymbolTable`。

符号表

创建一个 `Symbol` 类：

```
public class Symbol {
    private final String type;
    private final int intType; // 0 -> int, 1 -> int[], 2 -> int[][]
    private final String content;
    private final int areaID;
    ......
}
```

包含以下几种信息：

- `type`：符号的类型
- `intType`：如果值为0，代表符号是个整数`int`，如果是1，代表是个一维数组`int[]`，如果是2，代表是个二维数组`int[][]`
- `content`：符号的内容
- `areaID`：符号所在的作用域，用来处理 b类错误：名字重定义

接着创建一个名为`SymbolTable`的哈希表，用于管理语法分析过程中生成的符号对象，记录各符号的**名字（标识符）+其特征信息**。同时，提供一些公共方法以方便在语法分析阶段诊察出错误。

```
public class SymbolTable {
    private final HashMap<String, Symbol> symbolHashMap;

    public SymbolTable() {
        symbolHashMap = new HashMap<>();
    }

    public void addSymbol(String type, int intType, Token token, int areaID) {
//添加符号对象
        symbolHashMap.put(token.getContent(), new Symbol(type, intType, token,
areaID));
    }

    public boolean findSymbol(Token token) { // 符号表中是否有同样命名的符号
        return symbolHashMap.containsKey(token.getContent());
    }

    public boolean isConst(Token token) { // 符号表中保存的这个符号是否是一个'const'值
        Symbol s = symbolHashMap.get(token.getContent());
        return s != null && s.getType().equals("const");
    }
    ......
}
```

最后在语法分析器中创建 **addArea** 和 **removeArea** 来用于在代码的不同部分（或作用域）中管理符号表。

```
    private void addArea() {
        areaID++;
        area++;
        symboltable.put(area, new SymbolTable());
    }

    private void removeArea() {
        symboltable.remove(area);
        area--;
    }
```

Function

构造一个 `Function` 类，保存语法分析阶段分析到的函数及其内容，

```
public class Function {
    private final String content; // 内容
    private final String returnType; //返回类型
    private ArrayList<Integer> params; // 参数（可能有多个）

    public Function(Token token, String returnType) {
        this.content = token.getContent();
        this.returnType = returnType;
    }
    ......
}
```

主要用于处理 **d、e、f、g** 类函数相关的错误。

错误处理

将错误处理**耦合**入递归下降语法分析的过程中，若诊察出错误则输出：**错误类别码 + 所在行数**

- `a` 类：对每一处的 `FormatString` 做检查，检测非法符号

  具体代码如下：

```
    public boolean checkFormat() {
        int len = content.length();
        for (int i = 0; i < len - 1; i++) {
            char current = content.charAt(i);
            char currentNext = content.charAt(i + 1);
            if (!isValidCharacter(current)) { // 不是有效字符
                if (isPercentD(current, currentNext)) { // 是%d
                    continue;
                }
                return false; // 非法符号
            } else if (isBackslashWithoutNewline(current, currentNext)) { //
不是\n换行符
                return false;
            }
        }
        return true;
    }

    // 检查字符是否为有效字符
    private boolean isValidCharacter(char c) {
        return (c == 32 || c == 33 || (c >= 40 && c <= 126));
    }

    // 检查是否是 "%d"
    private boolean isPercentD(char c1, char c2) {
        return (c1 == '%' && c2 == 'd');
    }

    // 检查反斜杠是否后面不是换行符
```

```
    private boolean isBackslashWithoutNewline(char c1, char c2) {
        return (c1 == '\\' && c2 != 'n');
    }
```

如果检测到非法符号，则输出`a`类错误：

```
if (!strcon.checkFormat()) {
    error("a", strcon.getline()); // 非法符号
}
```

- `b`类和`c`类：建立符号表，
  - 对于`b`类重定义，每次调用标识符的时候检查当前作用域下是否已经定义了相同名字的变量或函数：

```
    private boolean checkSymbolInArea(Token token) {
        return symboltable.get(area).findSymbol(token);
    }
```

  - 对于`c`类未定义，检查表中所有数据，判断是否有未定义的情况：

```
    private boolean checkSymbol(Token token) {
            for (SymbolTable s : symboltable.values()) {
                if (s.findSymbol(token)) {
                    return true;
                }
            }
            return false; //符号表中找不到token
        }
```

- `d`类和`e`类：都属于函数调用错误，
  - 对于`d`类参数个数不匹配的情况，首先在函数声明时在符号表中记录该函数名对应的参数及其类型、个数等属性，在参数调用时对参数个数进行计数，如果不匹配则抛出异常，具体代码如下：

```
    private void checkParamsType(Token ident, ArrayList<Integer> params,
ArrayList<Integer> rparams) {
        if (params.size() != rparams.size()) {
            error("d", ident.getline());
        } else {
            ......
        }
    }

    // 检查函数参数
    private void checkParams(Token ident, ArrayList<Integer> params) {
        if (params != null) {
            if (params.size() != 0) {
                error("d", ident.getline()); // 函数参数个数不匹配
            }
```

```
        }
    }
```

- 对于`e`类参数类型不匹配的情况，对每一个传入参数检查其维度，仅有二维数组、一维数组、整数、void和数组的部分维度情况，因此仅需考参数的维度是否匹配

```
// 检查函数参数类型
    private void checkParamsType(Token ident, ArrayList<Integer> params,
ArrayList<Integer> rparams) {
        if (params.size() != rparams.size()) {
            error("d", ident.getline());
        } else {
            for (int i = 0; i < rparams.size(); i++) {
                if (!params.get(i).equals(rparams.get(i))) { // 函数类型不
匹配
                    error("e", ident.getline());
                }
            }
        }
    }
```

- `f`类和`g`类：都属于函数返回值错误问题，分别具体处理如下：
  - 对于`f`类无返回值函数存在不匹配的`return`语句，即`void`函数不能有返回值。代码中使用了一个全局变量`needReturn`来表示当前函数是否需要返回值，要注意的一点是，`return ;`是合法的，因此不能在`void`函数中遇到`return`就判断有`f`类错误：

```
else if (nextToken.typeIs(String.valueOf(Word.RETURNTK))) { // 'return'
[Exp] ';'
            getToken(); // return
            isReturn = true;
            boolean ret = false;
            if (getNext().typeSymbolizeExp()) {
                if (!needReturn) {
                    error("f"); // 无返回值的函数存在不匹配的return语句
                }
                analyseExp(getExp());
                ret = true;
            }
            checkSemicn(); // ;
            addCode(Operator.RET, (ret? 1 : 0));
}
```

  - 对于`g`类有返回值函数缺少`return`问题，需要检测函数`block`结构体最后一句是否为`return`类型语句，若不是`return`或`return ;`，则产生该错误：

```
private void analyseMainFuncDef() { // MainFuncDef → 'int' 'main' '('
')' Block
        ......
        needReturn = true; // main函数需要返回值
        boolean isReturn = analyseBlock(false); // 是否有返回值
        if (needReturn && !isReturn) {
            error("g"); // 有返回值的函数缺少return语句
        }
        addCode(Operator.EXIT);
        grammar.add("<MainFuncDef>"); // main函数定义
    }
```

- `h`类：每当对`Lval`进行赋值时，检查是否该标识符为`const`常量，若是，则产生该错误：

```
    // 检查是不是const
    private void checkConst(Token token) {
        if (isConst(token)) {
            error("h", token.getline()); // 不能改变常量的值
        }
    }

    ......

    private void analyseForStmt() { // ForStmt → LVal '=' Exp
        ArrayList<Token> exp = getExp();
        int intType = analyseLVal(exp); // LVal
        Token ident = exp.get(0);
        addCode(Operator.ADDRESS, getSymbol(ident).getAreaID() + "_" +
ident.getContent(), intType);
        checkConst(exp.get(0)); // 检查左值是否是常量
        getToken(); // =
        analyseExp(getExp()); // Exp
        addCode(Operator.POP, getSymbol(ident).getAreaID() + "_" +
ident.getContent());
        grammar.add("<ForStmt>");
    }
```

- `i,j,k`类：缺少分号`;`小括号`)`中括号`]`的情况，在语法分析过程中顺便处理即可。
- `l`类：`printf`中格式字符与表达式个数不匹配，类似`a`类，检查逻辑是在分别计数格式字符数量和表达式个数，当二者不相等时产生该错误：

```
else if (nextToken.typeIs(String.valueOf(Word.PRINTFTK))) { // 'printf' '('
FormatString { ',' Exp } ')' ';'
            getToken(); // printf
            Token printftk = current;
            getToken(); // (
            getToken(); // STRCON
            Token strcon = current;
            nextToken = getNext();
            int param = 0;
```

```
            while (nextToken.typeIs(String.valueOf(Word.COMMA))) {
                getToken(); // ,
                analyseExp(getExp()); // Exp
                param++;
                nextToken = getNext();
            }
            ......
            if (param != strcon.cntFormat()) {
                error("l", printftk.getline()); // printf中格式字符与表达式个数
不匹配
            }
            ......
            addCode(Operator.PRINT, strcon.getContent(), param);
        }

    ......

    public int cntFormat() {
        int cnt = 0;
        for (int i = 0; i < content.length(); i++) {
            if (i + 1 < content.length()) {
                char current = content.charAt(i);
                char currentNext = content.charAt(i + 1);
                if (isPercentD(current, currentNext)) {
                    cnt++;
                }
            }
        }
        return cnt;
    }
```

- `m`类：在非循环块中使用`break`和`continue`语句，检查方案为创建一个全局变量`forFlag`标记当前的循环体深度，每进入一个循环体该变量`+1`，退出则`-1`，为`0`表示当前在非循环块中，应当产生该错误：

```
        else if (nextToken.typeIs(String.valueOf(Word.FORTK))) {
        // 'for' '(' [ForStmt] ';' [Cond] ';' [ForStmt] ')' Stmt
            .....
            addCode(Operator.LABEL, forLabels.get(forLabels.size() -
1).get("for"));
            getToken(); // for
            forFlag++;
            getToken(); // (
            nextToken = getNext();
            // ForStmt
            // 初始化循环变量
            if (nextToken.typeIs(String.valueOf(Word.IDENFR))) { // ForStmt
                analyseForStmt();
            }
            getToken(); // ;
            nextToken = getNext();
            // Cond
            // 标记 for_cond 循环体条件判断的位置
```

```
            addCode(Operator.LABEL, forLabels.get(forLabels.size() -
1).get("for_cond"));
            if (nextToken.typeSymbolizeExp()) { // Cond
                analyseCond(String.valueOf(Word.FORTK));
                // 生成条件判断的跳转指令
                // cond不成立 -> for_end
                addCode(Operator.JZ, forLabels.get(forLabels.size() -
1).get("for_end"));
            }
            // cond成立 -> for_block
            addCode(Operator.JMP, forLabels.get(forLabels.size() -
1).get("for_block"));
            getToken(); // ;
            // ForStmt
            // 标记 for_stmt 循环变量更新的位置
            addCode(Operator.LABEL, forLabels.get(forLabels.size() -
1).get("for_stmt"));
            nextToken = getNext();
            if (nextToken.typeIs(String.valueOf(Word.IDENFR))) { // ForStmt
                analyseForStmt();
            }
            // 生成回到 for_cond 的跳转指令
            addCode(Operator.JMP, forLabels.get(forLabels.size() -
1).get("for_cond"));

            // 检查是否存在右小括号
            if (!getNext().typeIs(String.valueOf(Word.RPARENT))) {
                error("j"); // 缺少右小括号')'
            } else {
                getToken(); //)
            }

            // 标记 for 循环主体的起始处
            addCode(Operator.LABEL, forLabels.get(forLabels.size() -
1).get("for_block"));
            analyseStmt(); // Stmt
            forFlag--;
            ......

        } else if (nextToken.typeIs(String.valueOf(Word.BREAKTK))) { //
'break' ';'
            getToken(); // break
            addCode(Operator.JMP, forLabels.get(forLabels.size() -
1).get("for_end"));
            if (forFlag == 0) {
                error("m"); // 在非循环块中使用break语句
            }
            checkSemicn(); //;
        } else if (nextToken.typeIs(String.valueOf(Word.CONTINUETK))) { //
'continue' ';'
            getToken(); // continue
            addCode(Operator.JMP, forLabels.get(forLabels.size() -
1).get("for_stmt"));
            if (forFlag == 0) {
                error("m"); // 在非循环块中使用continue语句
```

```
            }
            checkSemicn(); //;
        }
```

最后，将错误信息输出：

```
    //  使用了 Comparator.comparingInt 和 Lambda 表达式来替代匿名比较器类。
    //  同时使用 try-with-resources 语句管理文件写入，自动确保资源被正确关闭，无需手动调用 writer.close()方法。
    public void printError(FileWriter writer) throws IOException {
        errors.sort(Comparator.comparingInt(Errors::getline));
        try {
            for (Errors error : errors) {
                writer.write(error + "\n");
            }
        } finally {
            if (writer != null) {
                writer.close();
            }
        }
    }
```

同时，提供一个公共方法 **hasError**，表示了在语法分析阶段诊察到了错误，这样在后续阶段就不需要再进行代码生成了。

```
   public boolean hasErrors() {
        return !errors.isEmpty();
    }
```

代码生成

在代码生成阶段，我选择生成 **Pcode。**

我基于**逆波兰表达式堆栈** 和 **符号表** 去设计PCODE。

同时，为了方便进行测试，我设计了虚拟机来执行生成的代码。

Pcode虚拟机是用于运行Pcode指令的虚拟机，它由：**代码区（code）、指令指针（EIP）、堆栈、var_table、func_table 和label_table** 组成。

下面将介绍Pcode是如何执行的，以及编译器是如何产生Pcode代码的。

虚拟机的工作原理

首先，我们需要准备一个（Pcode）`codes` 列表和一个 （int）`stack` ，以及

`eip`：显示当前运行代码的地址。

`varTable`：记住变量在堆栈中的地址。

`funcTable`：记住代码列表中函数的地址。

`labelTable`：记住代码列表中标签的地址。

上面三个表都用哈希表来实现。最后，依次运行代码并管理堆栈。

Pcode

在生成代码之前，我通过唯一的范围编号 `areaID` 来区分不同作用域的代码，例如：`areaID + "_" + curWord.getContent()`。这种情况下，除了递归函数调用外，代码中不会出现多次，可以通过压入 `varTable` 入栈来解决。

首先，为 `PCode` 定义一个类：

```
public class PCode {
    private final Operator type;
    private Object value1;
    private Object value2;
    ......
```

类中包含了以下成员：

- `type` 属性代表指令的类型，是一个 `Operator` 类型的对象。
- `value1` 和 `value2` 属性是指令可能需要的参数或值，可以是任意类型的对象。

另外，类中覆盖了 `toString()` 方法，使用了 Java 12 引入的 switch 表达式。根据不同的指令类型，生成不同格式的字符串表示，方便在程序中输出指令的字符串表示：

- 对于 `LABEL` 类型的指令，格式为 `value1:`。
- 对于 `FUNC` 类型的指令，格式为 `FUNC @value1:`。
- 对于其他类型的指令，格式为 `type value1, value2` 或者只有 `value1`，取决于值是否为 `null`。

Operator

接着，一个名为 `Operator` 的枚举类型，其中包含了执行pcode需要的指令集，具体代码如下：

```
public enum Operator {
    LABEL(OperationCategory.FLOW_CONTROL),     // 标签
    VAR(OperationCategory.FLOW_CONTROL),       // 变量

    PUSH(OperationCategory.ARITHMETIC),        // 入栈
    POP(OperationCategory.ARITHMETIC),        // 出栈

    ADD(OperationCategory.ARITHMETIC),        // 加法
    SUB(OperationCategory.ARITHMETIC),        // 减法
    ......
```

Func

```
public record Func(int index, int args) {
}
```

这段代码中使用了 `Java 16` 引入的新特性之一：**records**。Records 是一种轻量级的数据封装方式，用于创建不可变的数据模型。

在这里，我定义了一个名为 `Func` 的 record 类型，它有两个成员变量 `index` 和 `args`，分别表示函数的索引和参数数量。在 record 类型中，这些成员变量是隐式的，不需要额外的字段去声明。

当创建 `Func` 类型的实例时，会自动为这些成员变量生成构造函数、访问方法和 `toString()` 方法等。这样的记录类型非常适合用于表示简单的数据传输对象（DTO），并且具有不变性。

RetInfo

我定义了一个名为 `RetInfo` 的 record 类型，包含了多个成员变量，其中包括 `eip`、`varTable`、`stackPtr`、`paramNum`、`needArgsNum` 和 `nowArgsNum`。

```
public record RetInfo(int eip, HashMap<String, Var> varTable, int stackPtr, int
paramNum, int needArgsNum,
                      int nowArgsNum) {
    public RetInfo(int eip, HashMap<String, Var> varTable, int stackPtr, int
paramNum, int needArgsNum, int nowArgsNum) {
        this.eip = eip;
        this.varTable = (HashMap<String, Var>) new HashMap<>(varTable);
        this.stackPtr = stackPtr;
        this.paramNum = paramNum;
        this.needArgsNum = needArgsNum;
        this.nowArgsNum = nowArgsNum;
    }

    @Override
    public HashMap<String, Var> varTable() {
        return new HashMap<>(varTable);
    }
}
```

同时，提供了一个构造函数，接受相同的参数并将它们分别赋值给记录的成员变量。在构造函数中，对于 `varTable` 成员，使用了拷贝的方式来避免引用问题。这种方式创建了一个新的 `HashMap`，将原始 `varTable` 中的内容拷贝到新的 `HashMap` 中，以保证记录对象的不可变性。

此外，重写了 `varTable()` 方法，返回了一个新的 `HashMap` 的拷贝，同样是为了保证返回的是副本而不是原始引用。

LabelGenerator

我定义了一个 `LabelGenerator` 类，用于生成标签 `label`。在该类中，使用了 `AtomicInteger` 类型的 `labelCount` 变量来保证在多线程环境下对标签计数的安全性。

```
public class LabelGenerator {
    private final AtomicInteger labelCount = new AtomicInteger(0);

    public String generateLabel(String type) {
        int currentCount = labelCount.incrementAndGet();
        return String.format("label_%s_%d", type, currentCount);
    }
}
```

`generateLabel` 方法接受一个 `type` 参数，它会生成一个新的标签，格式为 `"label_type_数字"`，其中数字是递增的计数值，用于标识不同的标签类型。

在方法中，通过调用 `labelCount.incrementAndGet()` 方法获取当前的计数值，然后将其与 `type` 字符串组合成一个新的标签字符串，并返回生成的标签。

这样的设计适合在编译器或解释器等环境中，需要生成唯一标签的场景，在后续中主要用来处理 `if` 和 `for` 语句的跳转指令。

Var

这段代码定义了一个 `Var` 类，用于表示一个变量，在这个场景中，这些变量具有多维度的特性：

```
public class Var {
    private final int index;
    private int dimension = 0;
    private int[] dimensions = new int[2];
    ......
```

- `index` 属性表示该变量的索引。
- `dimension` 属性表示变量的维度数。
- `dimensions` 数组用于存储不同维度的值。

同时，类中提供一些方法：

```
    public int getIndex() {
        return index;
    }

    /**
     * 获取维度值数组。
     *
     * @return 维度值数组
     */
    public int[] getDimensions() {
        return dimensions;
    }

    /**
     * 获取指定维度的值。
     *
     * @param dimension 维度索引（从 1 开始）
     * @return 指定维度的值
     */
    public int getDimensionValue(int dimension) {
        assertValidDimension(dimension);
        return dimensions[dimension - 1];
    }

    private void assertValidDimension(int dimension) {
        if (dimension <= 0 || dimension > dimensions.length) {
            throw new IllegalArgumentException("Invalid dimension index: " +
dimension);
        }
    }

    /**
     * 设置指定维度的值。
     *
     * @param dimension 维度索引（从 1 开始）
```

```
     * @param value      维度值
     */
    public void setDimensionValue(int dimension, int value) {
        assertValidDimension(dimension);
        dimensions[dimension - 1] = value;
    }

    /**
     * 获取维度数。
     *
     * @return 维度数
     */
    public int getDimension() {
        return dimension;
    }

    /**
     * 设置维度数。
     *
     * @param dimension 维度数
     */
    public void setDimension(int dimension) {
        this.dimension = dimension;
        if (dimension >= 2) { // 确保至少有两个维度
            this.dimensions = new int[dimension]; // 初始化维度数组
        } else {
            //throw new IllegalArgumentException("At least two dimensions are
required.");
            this.dimensions[0] = 0;
            this.dimensions[1] = 0;
        }
    }
```

- `getIndex()` 返回变量的索引。
- `getDimensions()` 返回整个维度值数组。
- `getDimensionValue(int dimension)` 返回指定维度的值。
- `setDimensionValue(int dimension, int value)` 设置指定维度的值。
- `getDimension()` 返回维度数。
- `setDimension(int dimension)` 设置维度数，并根据维度数初始化 `dimensions` 数组。

这个类用于管理变量的维度和值。例如，`setDimension(2)` 将变量定义为一个二维数组，`setDimensionValue(1, 10)` 可以设置第一个维度的值为 10。

Executor

最后，我定义了一个名为 `Executor` 的类，用来模拟执行器，来执行 `PCode` 指令集。

类包含了许多成员变量来执行各种指令：

```
public class Executor {
    private final ArrayList<PCode> codes;
    private final ArrayList<RetInfo> retInfos = new ArrayList<>();
    private final ArrayList<Integer> stack = new ArrayList<>();
    private int eip = 0;
```

```
    private HashMap<String, Var> varTable = new HashMap<>();
    private final HashMap<String, Func> funcTable = new HashMap<>();
    private final HashMap<String, Integer> labelTable = new HashMap<>();

    private int mainAddress;

    private final ArrayList<String> prints = new ArrayList<>();
    private final Scanner scanner;

    public Executor(ArrayList<PCode> codes, Scanner scanner) {
        this.codes = codes;
        this.scanner = scanner;

        for (int i = 0; i < codes.size(); i++) {
            PCode code = codes.get(i);
            Operator codeType = code.getType();
            switch (codeType) {
                case MAIN -> mainAddress = i;
                case LABEL -> labelTable.put((String) code.getValue1(), i);
                case FUNC -> funcTable.put((String) code.getValue1(), new
Func(i, (int) code.getValue2()));
            }
        }
    }
```

- `codes` 存储了一系列的 PCode 指令。
- `retInfos` 存储了返回信息。
- `stack` 代表执行中的堆栈。
- `eip` 是指令指针，用于追踪当前执行的指令。
- `varTable` 存储了变量的映射表。
- `funcTable` 存储了函数的映射表。
- `labelTable` 存储了标签与其在指令中的位置的映射。
- `mainAddress` 记录了主函数的地址。
- `prints` 是用于存储打印输出信息的列表。
- `scanner` 用于读取输入信息。

同时，类中提供了一些公共方法：

`run()` 方法是执行 PCode 指令序列的核心方法。根据不同类型的指令，它会调用相应的处理方法。这些处理方法执行了与指令相关的操作，例如操作堆栈、处理变量、跳转等。

```
public void run() {
        int callArgsNum = 0;
        int nowArgsNum = 0;
        boolean mainFlag = false;
        ArrayList<Integer> rparams = new ArrayList<>();
        for (; eip < codes.size(); eip++) {
            PCode code = codes.get(eip);
            switch (code.getType()) {
```

```
                case LABEL, END_FUNC -> { // 不需要执行操作

                }
                case VAR ->
                        handleVarInstruction(code);
                case PUSH ->
                        handlePushInstruction(code);
                case POP ->
                        handlePopInstruction();
                case ADD, SUB, MUL, DIV, MOD, NEG, POS ->
                        handleArithmeticInstruction(code);
                    ......
```

以及，`print()` 方法用于将打印输出信息写入到名为 "pcoderesult.txt" 的文件中。

```
    public void print() throws IOException {
        try (BufferedWriter writer = new BufferedWriter(new
FileWriter("pcoderesult.txt"))) {
            for (String s : prints) {
                writer.write(s);
            }
        }
    }
```

接下来，我们一个个讲解 `Executor` 如何模拟 `Pcode` 的执行。

```
    private void handleVarInstruction(PCode code) {
        Var var = new Var(stack.size());
        varTable.put((String) code.getValue1(), var);
    }
```

该方法创建一个新的 `Var` 对象，并将其添加到 `varTable` 中，以便在后续的执行中对这个变量进行引用和操作。

具体操作步骤如下：

1. 通过 `stack.size()` 创建一个新的 `Var` 对象，使用堆栈的大小作为变量的索引。
2. 使用 PCode 指令中的值作为键，将新创建的 `Var` 对象放入 `varTable` 中。这个键通常是变量的标识符或名称。

```
    private void push(int i) {
        stack.add(i);
    }

    private int pop() {
        if (stack.isEmpty()) {
            // 处理堆栈为空的情况
            return -1;
        }
        return stack.remove(stack.size() - 1);
    }

    private void handlePushInstruction(PCode code) {
        if (code.getValue1() instanceof Integer) {
```

```
            push((Integer) code.getValue1());
        }
    }

    private void handlePopInstruction() {
        int value = pop();
        int address = pop();
        // 检查address是否在合法范围内
        if (address >= 0 && address < stack.size()) {
            stack.set(address, value);
        } else {
            throw new IndexOutOfBoundsException("Invalid address: " + address);
        }
    }
```

- `handlePushInstruction(PCode code)` 方法：这个方法用于处理推送指令。它检查 PCode 指令中的值是否是整数，如果是，则将其压入堆栈中。
  - 首先，它通过 `code.getValue1() instanceof Integer` 条件判断 PCode 中的值是否是整数类型。
  - 如果条件成立，就调用 `push((Integer) code.getValue1())` 方法，将这个整数值压入堆栈中。
- `handlePopInstruction()` 方法：这个方法用于处理弹出指令。它从堆栈中弹出一个值，并将其放入另一个地址中。
  - 首先，它执行两次 `pop()` 操作，分别得到要写入的值 `value` 和目标地址 `address`。
  - 然后，它检查目标地址是否在合法范围内（即是否在堆栈的有效范围内）。
  - 如果目标地址有效，则使用 `stack.set(address, value)` 将值写入堆栈指定的地址。
  - 如果地址无效（超出堆栈范围），则抛出 `IndexOutOfBoundsException` 异常，指出无效的地址。

```
    private void handleArithmeticInstruction(PCode code) {
        switch (code.getType()) {
            case ADD -> {
                int b = pop();
                int a = pop();
                push(a + b);
            }
            case SUB -> {
                int b = pop();
                int a = pop();
                push(a - b);
            }
            case MUL -> {
                int b = pop();
                int a = pop();
                push(a * b);
            }
            case DIV -> {
                int b = pop();
                int a = pop();
                push(a / b);
            }
            case MOD -> {
                int b = pop();
```

```
                int a = pop();
                push(a % b);
            }
            case NEG -> push(-pop());
            case POS -> push(pop());
        }
    }
```

这段代码是处理算术指令的方法，根据 PCode 中的算术操作类型执行相应的算术运算。

- 对于 `ADD`、`SUB`、`MUL`、`DIV`、`MOD` 操作：
  - 首先，它从堆栈中弹出两个整数值 `a` 和 `b`。
  - 然后根据对应的操作类型进行相应的算术运算。
  - 最后，将运算结果推入堆栈中。
- 对于 `NEG` 操作：
  - 直接将堆栈顶部的值取负数后推入堆栈。
- 对于 `POS` 操作：
  - 直接将堆栈顶部的值取正数后推入堆栈。

```
    private void handleComparisonInstruction(PCode code) {
        int b = pop();
        int a = pop();
        int result = 0;
        switch (code.getType()) {
            case EQ -> result = a == b ? 1 : 0;
            case NE -> result = a != b ? 1 : 0;
            case LT -> result = a < b ? 1 : 0;
            case LTE -> result = a <= b ? 1 : 0;
            case GT -> result = a > b ? 1 : 0;
            case GTE -> result = a >= b ? 1 : 0;
        }
        push(result);
    }
```

这段代码是处理比较指令的方法。它通过对比两个从堆栈中弹出的整数值 `a` 和 `b`，根据比较操作的类型执行相应的比较，然后将结果推入堆栈。

- 首先，它从堆栈中弹出两个整数值 `a` 和 `b`。
- 然后根据指令的类型（`EQ`、`NE`、`LT`、`LTE`、`GT`、`GTE`）执行对应的比较操作，并将结果存储在 `result` 变量中。
- 如果比较成立，将 `result` 设置为 `1`，否则设置为 `0`。
- 最后，将比较的结果（`result`）推入堆栈中。

```
    private void handleLogicalInstruction(PCode code) {
        switch (code.getType()) {
            case AND -> {
                boolean b = pop() != 0;
                boolean a = pop() != 0;
                push((a && b) ? 1 : 0);
            }
            case OR -> {
                boolean b = pop() != 0;
```

```
                boolean a = pop() != 0;
                push((a || b) ? 1 : 0);
            }
            case NOT -> {
                boolean a = pop() != 0;
                push((!a) ? 1 : 0);
            }
        }
    }
```

这段代码处理逻辑运算指令，它根据 `PCode` 中的逻辑操作类型执行相应的逻辑运算，并将结果推入堆栈中。

- 对于 `AND` 操作：
  - 从堆栈中弹出两个整数值，并将它们视为布尔类型。然后执行逻辑与运算（`&&`），并将结果推入堆栈。
- 对于 `OR` 操作：
  - 从堆栈中弹出两个整数值，并将它们视为布尔类型。然后执行逻辑或运算（`||`），并将结果推入堆栈。
- 对于 `NOT` 操作：
  - 从堆栈中弹出一个整数值，并将其视为布尔类型。然后执行逻辑非运算（`!`），并将结果推入堆栈。

```
    private void handleControlInstruction(PCode code, ArrayList<Integer> stack,
HashMap<String, Integer> labelTable) {
        int value = stack.get(stack.size() - 1);
        String label = (String) code.getValue1();
        switch (code.getType()) {
            case JZ -> {
                if (value == 0) {
                    eip = labelTable.get(label);
                }
            }
            case JNZ -> {
                if (value != 0) {
                    eip = labelTable.get(label);
                }
            }
            case JMP -> eip = labelTable.get(label);
        }
    }
```

这段代码处理控制指令，根据不同的指令类型执行跳转逻辑，以便在程序执行期间实现不同的控制流程。

- 对于 `JZ` 指令：
  - 从堆栈中弹出一个整数值，并将其视为条件值。
  - 如果条件值为零，则根据指定的标签进行跳转，将执行指令的指针 `eip` 设置为对应标签的地址。
- 对于 `JNZ` 指令：
  - 从堆栈中弹出一个整数值，并将其视为条件值。

    - 如果条件值不为零，则根据指定的标签进行跳转，将执行指令的指针 `eip` 设置为对应标签的地址。
- 对于 `JMP` 指令：
    - 直接根据指定的标签进行无条件跳转，将执行指令的指针 `eip` 设置为对应标签的地址。

```
    // 使用 Matcher 查找 %d 并逐个替换为参数列表中的值，并将结果添加到 StringBuilder 中。
    // 最后将 StringBuilder 转换为字符串，并将其添加到 prints 列表中。
    private void handlePrintInstruction(PCode code) {
        String s = (String) code.getValue1();
        int n = (int) code.getValue2();
        ArrayList<Integer> params = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            params.add(pop());
        }

        StringBuilder builder = new StringBuilder();
        int index = n - 1;
        Matcher matcher = Pattern.compile("%d").matcher(s); // 创建匹配 "%d" 的
Matcher 对象
        while (matcher.find() && index >= 0) {
            builder.append(s, 0, matcher.start()).append(params.get(index--));
            s = s.substring(matcher.end());
            matcher = Pattern.compile("%d").matcher(s);
        }
        builder.append(s);

        prints.add(builder.substring(1, builder.length() - 1));
    }
```

这段代码是处理打印指令的逻辑。它通过在字符串中查找 `%d`，将其逐个替换为参数列表中的值，并将替换后的结果添加到 `prints` 列表中。

1. 首先，从指令中获取一个格式化的字符串 `s` 和参数数量 `n`。
2. 然后，从堆栈中弹出 `n` 个整数值，并将它们放入参数列表 `params` 中。
3. 接下来，使用正则表达式 `Pattern.compile("%d").matcher(s)` 查找字符串 `s` 中的 `%d` 格式。
4. 循环执行以下操作：
    - 找到 `%d` 的匹配项，将 `%d` 之前的部分和对应的参数值拼接到 `StringBuilder` 对象 `builder` 中。
    - 更新字符串 `s`，删除已匹配的部分。
    - 重置 `matcher` 对象，从新的 `s` 中继续查找 `%d`。
5. 将最后一个未匹配的字符串（如果有）追加到 `builder` 中。
6. 最后，将拼接好的字符串（去除开头和结尾的空白字符）添加到 `prints` 列表中。

```
  private void handleFuncControlInstruction(PCode code) {
        switch (code.getType()) {
            case VALUE -> {
                Var var = getVar((String) code.getValue1());
                int n = (int) code.getValue2();
                int address = 0;
                if (var != null) {
                    address = getAddress(var, n);
```

```
                }
                push(stack.get(address));
            }
            case ADDRESS -> {
                Var var = getVar((String) code.getValue1());
                int n = (int) code.getValue2();
                int address = 0;
                if (var != null) {
                    address = getAddress(var, n);
                }
                push(address);
            }
            case DIMVAR -> {
                Var var = getVar((String) code.getValue1());
                int dim = (int) code.getValue2();
                if (var != null) {
                    var.setDimension(dim);
                }
                // 一维数组
                if (dim == 1) {
                    int i = pop();
                    if (var != null) {
                        var.setDimensionValue(1, i);
                    }
                }
                // 二维数组
                if (dim == 2) {
                    int j = pop(), i = pop();
                    if (var != null) {
                        var.setDimensionValue(1, i);
                    }
                    if (var != null) {
                        var.setDimensionValue(2, j);
                    }
                }
            }
            case PLACEHOLDER -> {
                Var var = getVar((String) code.getValue1());
                int intType = (int) code.getValue2();
                if (intType == 0) {
                    push(0);
                }
                // 将第一维度的零推送到栈中
                if (intType == 1) {
                    if (var != null) {
                        for (int i = 0; i < var.getDimensionValue(1); i++) {
                            push(0);
                        }
                    }
                }
                // 将二维数组的零推送到栈中
                if (intType == 2) {
                    if (var != null) {
                        for (int i = 0; i < var.getDimensionValue(1) *
var.getDimensionValue(2); i++) {
```

```
                        push(0);
                    }
                }
            }
        }
        case EXIT -> {
        }
    }
}
```

这段代码负责处理函数控制指令，它根据指令类型执行不同的操作：

- `VALUE`：根据指定的变量和维度索引，获取变量的地址，并将其对应的值推入堆栈中。
- `ADDRESS`：类似于 `VALUE`，但是推送的是变量的地址。
- `DIMVAR`：设置变量的维度，并根据维度的不同进行相应的操作：
  - 如果是一维数组，从堆栈中弹出一个值，并将其作为第一维的值存储在变量中。
  - 如果是二维数组，从堆栈中弹出两个值，并分别作为第一维和第二维的值存储在变量中。
- `PLACEHOLDER`：根据指定的整型类型 `intType` 进行操作：
  - 如果为 `0`，将 `0` 推入堆栈中。
  - 如果为 `1`，将对应变量的第一维度大小个 `0` 推入堆栈中。
  - 如果为 `2`，将对应变量的二维数组大小个 `0` 推入堆栈中。
- `EXIT`：表示指令结束，不进行任何操作。

这些指令用于在虚拟机执行期间处理不同类型的流程控制、变量设置和堆栈操作，以确保正确的执行和结果。

同时需要一个 `getAddress` 方法，根据给定的 `Var` 对象和维度类型来计算地址。

```
    private int getAddress(Var var, int intType) {
        int dimension = var.getDimension() - intType;
        // 根据不同的维度差值调用相应的方法来计算地址
        return switch (dimension) {
            case 0 -> getAddressForDimensionZero(var);
            case 1 -> getAddressForDimensionOne(var);
            case 2 -> getAddressForDimensionTwo(var);
            default -> throw new IllegalArgumentException("Invalid dimension: "
+ dimension);
        };
    }

    private int getAddressForDimensionZero(Var var) {
        // 返回零维度变量的索引
        return var.getIndex();
    }

    private int getAddressForDimensionOne(Var var) {
        // 获取栈顶的值
        int i = pop();
        if (var.getDimension() == 1) {
            // 如果是一维数组，返回索引+偏移量
            return var.getIndex() + i;
        } else {
            // 如果是二维数组，返回索引+偏移量
```

```
            return var.getIndex() + var.getDimensionValue(2) * i;
        }
    }

    private int getAddressForDimensionTwo(Var var) {
        // 获取栈顶的两个值
        int j = pop();
        int i = pop();
        // 返回索引+偏移量
        return var.getIndex() + var.getDimensionValue(2) * i + j;
    }
```

```
    case MAIN -> {
                    mainFlag = true;
                    retInfos.add(new RetInfo(codes.size(), varTable,
stack.size() - 1, 0, 0, 0));
                    varTable = new HashMap<>();
                }
```

在这段代码中，当遇到 `case MAIN` 时，

它标记了程序的主要部分（main）。它设置了 `mainFlag` 为 `true`，表示程序的主要部分已经开始执行。

然后，它向 `retInfos` 中添加了一个 `RetInfo` 对象，这个对象包含了程序结束的标志位(`codes.size()`)、当前的变量表(`varTable`)、当前栈的大小(`stack.size() - 1`)以及需要参数的数量和现有参数的数量（都设置为 0）。

最后，它将 `varTable` 重新初始化为一个空的 `HashMap<>`，以准备执行主程序部分时使用的新变量表。

```
    case PARAM -> {
                    Var para = new Var(rparams.get(rparams.size() - callArgsNum
+ nowArgsNum));
                    int n = (int) code.getValue2();
                    para.setDimension(n);
                    if (n == 2) {
                        para.setDimensionValue(2,pop());
                    }
                    varTable.put((String) code.getValue1(), para);
                    nowArgsNum++;
                    if (nowArgsNum == callArgsNum) {
                        rparams.subList(rparams.size() - callArgsNum,
rparams.size()).clear();
                    }
                }

    case RPARAM -> {
                    int n = (int) code.getValue1();
                    if (n == 0) {
                        rparams.add(stack.size() - 1);
                    } else {
                        rparams.add(stack.get(stack.size() - 1));
```

```
                    }
                }
```

这段代码处理函数调用和返回时的参数管理，

1. `PARAM`：
    - 这个情况处理函数调用时接收参数的情况。
    - 它根据 `rparams` 列表中相应索引处的参数创建一个名为 `para` 的 `Var` 对象。
    - 根据提供的值（`n`）设置参数的维度。
    - 如果维度为 2，使用 `pop()` 从栈中获取第二个维度的值。
    - 将这个参数 `para` 放入 `varTable` 中。
    - 增加 `nowArgsNum` 以跟踪添加的参数数量。
    - 最后，如果当前参数数量（`nowArgsNum`）等于预期的总参数数量（`callArgsNum`），则从 `rparams` 列表中清除添加的参数。
2. `RPARAM`：
    - 这个情况处理函数返回时的参数。
    - 如果 `n` 是 0，则将栈顶的值（`stack.size() - 1`）加入 `rparams` 列表。
    - 否则，将栈顶的值（`stack.get(stack.size() - 1)`）加入 `rparams` 列表。

```
    case RET -> {
                    int n = (int) code.getValue1();
                    RetInfo info = retInfos.remove(retInfos.size() - 1);
                    eip = info.eip();
                    varTable = info.varTable();
                    callArgsNum = info.needArgsNum();
                    nowArgsNum = info.nowArgsNum();
                    if (n == 1) {
                        stack.subList(info.stackPtr() + 1 - info.paramNum(),
stack.size() - 1).clear();
                    } else {
                        stack.subList(info.stackPtr() + 1 - info.paramNum(),
stack.size()).clear();
                    }
                }
```

这段代码对函数返回的情况进行处理：

1. 当执行到 `RET` 时，它首先获取一个参数 `n`，该参数指示是否要清除栈中的参数。
2. 接着，它从 `retInfos` 列表中移除最后一个 `RetInfo` 对象，并使用这个对象存储的信息来恢复执行：
    - 设置指令指针 `eip` 为 `RetInfo` 对象中存储的值。
    - 恢复函数返回前的变量表状态。
    - 恢复调用函数所需的参数数量 `callArgsNum` 和当前已传递的参数数量 `nowArgsNum`。
3. 最后，根据 `n` 的值，它检查要清除的参数数量并从栈中清除这些参数。

```
    case CALL -> {
                    Func func = funcTable.get((String) code.getValue1());
                    retInfos.add(new RetInfo(eip, varTable, stack.size() - 1,
func.args(), func.args(), nowArgsNum));
                    eip = func.index();
                    varTable = new HashMap<>();
                    callArgsNum = func.args();
                    nowArgsNum = 0;
                }
```

这部分代码用来处理函数调用，并在调用前准备好函数的执行环境：

1. 当遇到 `CALL` 指令时，它首先获取要调用的函数的信息，并从 `funcTable` 中检索到相应的函数对象。
2. 创建一个新的 `RetInfo` 对象，存储当前执行的位置（`eip`）、当前的变量表状态（`varTable`）、栈顶指针位置（`stack.size() - 1`）、调用函数所需的参数数量以及当前已传递的参数数量。
3. 将指令指针 `eip` 设置为函数的入口地址。
4. 为调用新函数准备一个新的变量表。
5. 设置调用函数所需的参数数量 `callArgsNum` 为函数定义中声明的参数数量，并将当前已传递的参数数量 `nowArgsNum` 设置为零。

```
    case GETINT -> {
                    int in = scanner.nextInt();
                    push(in);
                }
```

这段代码用来处理输入指令，

1. 当程序执行到 `GETINT` 指令时，它从用户输入中读取一个整数值。
2. 使用 `scanner.nextInt()` 从控制台获取输入的整数。
3. 将获取到的整数值推入栈顶，通过 `push(in)` 将该整数值压入栈顶，以便后续的指令可以使用这个值。

代码生成

addCode

```
    private void addCode(Operator op) {
        codes.add(new PCode(op));
    }

    private void addCode(Operator op, String content) {
        codes.add(new PCode(op, content));
    }

    private void addCode(Operator op, String content, int intType) {
        codes.add(new PCode(op, content, intType));
    }

    private void addCode(Operator op, int value) {
        codes.add(new PCode(op, value));
    }
```

这段代码用来添加指令到 `codes` 列表中的。每个方法都接受不同类型和数量的参数，并且基于这些参数创建一个 `PCode` 对象，然后将其添加到 `codes` 列表中。可能的参数类型包括操作符（`Operator`）、字符串内容（`String`）和整数（`int`），通过重载实现了不同参数类型的组合。

举例来说，

```
private void analyseNumber(Token token) { // Number → IntConst
    addCode(Operator.PUSH, Integer.parseInt(token.getContent()));
    grammar.add(token.toString());
    grammar.add("<Number>");
}
```

这段代码中，使用了 `addCode` 方法将 `Operator.PUSH` 和该整数值添加到 `codes` 列表中，之后解释器便会将整数值推送到栈中。

其中较为复杂的有 **条件控制语句** 和 **短路求值**。

条件控制语句

对于条件控制语句，处理方式类似于理论课上的逆波兰式，参考了拉链回填的设计思路，过程中我生成了标签作为标记，方便跳转语句进行跳转。

以 **if** 为例，

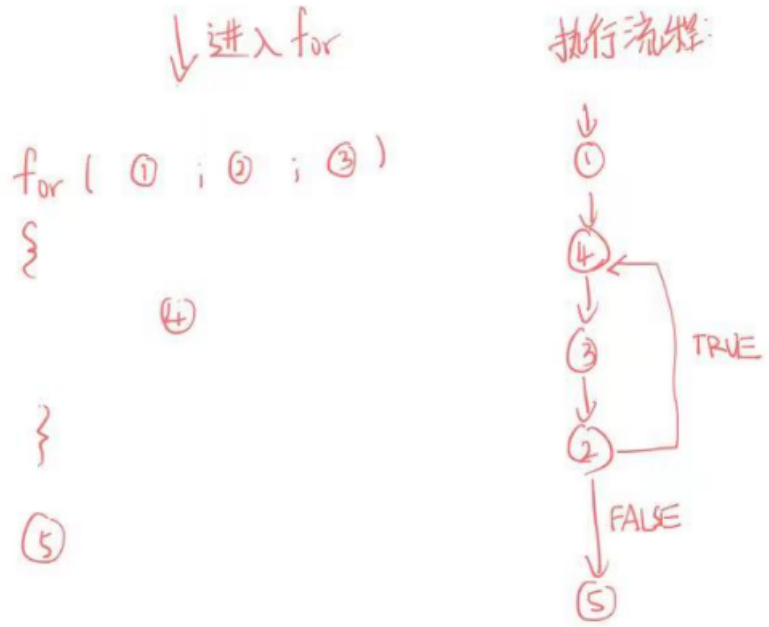
```
    // 'if' '(' Cond ')' Stmt [ 'else' Stmt ]
    else if (nextToken.typeIs(String.valueOf(Word.IFTK))) {
            // 生成 if 标签，然后存储在堆栈类型结构中
            ifLabels.add(new HashMap<>());
            ifLabels.get(ifLabels.size() - 1).put("if",
labelGenerator.generateLabel("if"));
            ifLabels.get(ifLabels.size() - 1).put("else",
labelGenerator.generateLabel("else"));
            ifLabels.get(ifLabels.size() - 1).put("if_end",
labelGenerator.generateLabel("if_end"));
            ifLabels.get(ifLabels.size() - 1).put("if_block",
labelGenerator.generateLabel("if_block"));

            addCode(Operator.LABEL, ifLabels.get(ifLabels.size() -
1).get("if"));// 标记if开始的位置
            getToken(); // if
            getToken(); // (
            analyseCond(String.valueOf(Word.IFTK)); // Cond
            if (!getNext().typeIs(String.valueOf(Word.RPARENT))) {
                error("j"); //缺少右小括号')'
            } else {
                getToken(); //)
            }
            addCode(Operator.JZ, ifLabels.get(ifLabels.size() - 1).get("else"));
// 条件判断为假 → else
            addCode(Operator.LABEL, ifLabels.get(ifLabels.size() -
1).get("if_block")); // 标记if执行的位置
            analyseStmt(); // Stmt
            nextToken = getNext();
```

```
            addCode(Operator.JMP, ifLabels.get(ifLabels.size() -
1).get("if_end")); // 去if结束后下一条语句（if_end）
            addCode(Operator.LABEL, ifLabels.get(ifLabels.size() -
1).get("else")); // 标记else执行的位置
            if (nextToken.typeIs(String.valueOf(Word.ELSETK))) {
                getToken(); // else
                analyseStmt(); // Stmt
            }
            addCode(Operator.LABEL, ifLabels.get(ifLabels.size() -
1).get("if_end")); // 标记整个if模块结束的位置
            ifLabels.remove(ifLabels.size() - 1); // 清空标记
        }
```

用一张图形象地解释下流程，

for 的流程更为复杂，先上图:



再来看代码，

```
    else if (nextToken.typeIs(String.valueOf(Word.FORTK))) {
            // 'for' '(' [ForStmt] ';' [Cond] ';' [ForStmt] ')' Stmt
            // ForStmt 表示初始化和更新部分，Cond 表示条件判断部分，Stmt 表示循环体部分
            forLabels.add(new HashMap<>());
            forLabels.get(forLabels.size() - 1).put("for",
labelGenerator.generateLabel("for"));
            forLabels.get(forLabels.size() - 1).put("for_end",
labelGenerator.generateLabel("for_end"));
            forLabels.get(forLabels.size() - 1).put("for_block",
labelGenerator.generateLabel("for_block"));
            forLabels.get(forLabels.size() - 1).put("for_stmt",
labelGenerator.generateLabel("for_stmt"));
            forLabels.get(forLabels.size() - 1).put("for_cond",
labelGenerator.generateLabel("for_cond"));

            addCode(Operator.LABEL, forLabels.get(forLabels.size() -
1).get("for"));
            getToken(); // for
            forFlag++;
            getToken(); // (
            nextToken = getNext();
            // ForStmt
            // 初始化循环变量
            if (nextToken.typeIs(String.valueOf(Word.IDENFR))) { // ForStmt
                analyseForStmt();
            }
            getToken(); // ;
            nextToken = getNext();
            // Cond
            // 标记 for_cond 循环体条件判断的位置
            addCode(Operator.LABEL, forLabels.get(forLabels.size() -
1).get("for_cond"));
```

```
            if (nextToken.typeSymbolizeExp()) { // Cond
                analyseCond(String.valueOf(Word.FORTK));
                // 生成条件判断的跳转指令
                // cond不成立 -> for_end
                addCode(Operator.JZ, forLabels.get(forLabels.size() -
1).get("for_end"));
            }
            // cond成立 -> for_block
            addCode(Operator.JMP, forLabels.get(forLabels.size() -
1).get("for_block"));
            getToken(); // ;
            // ForStmt
            // 标记 for_stmt 循环变量更新的位置
            addCode(Operator.LABEL, forLabels.get(forLabels.size() -
1).get("for_stmt"));
            nextToken = getNext();
            if (nextToken.typeIs(String.valueOf(Word.IDENFR))) { // ForStmt
                analyseForStmt();
            }
            // 生成回到 for_cond 的跳转指令
            addCode(Operator.JMP, forLabels.get(forLabels.size() -
1).get("for_cond"));

            // 检查是否存在右小括号
            if (!getNext().typeIs(String.valueOf(Word.RPARENT))) {
                error("j"); // 缺少右小括号')'
            } else {
                getToken(); //)
            }

            // 标记 for 循环主体的起始处
            addCode(Operator.LABEL, forLabels.get(forLabels.size() -
1).get("for_block"));
            analyseStmt(); // Stmt
            forFlag--;

            // 生成回到 for_stmt 的跳转指令
            addCode(Operator.JMP, forLabels.get(forLabels.size() -
1).get("for_stmt"));
            // 标记 for 循环结束的位置
            addCode(Operator.LABEL, forLabels.get(forLabels.size() -
1).get("for_end"));
            // 移除该 for 循环的标签信息
            forLabels.remove(forLabels.size() - 1);

        }
```

这块我也解释不清，跳来跳去很复杂，画一张图来辅助自己的帮助很大。有一点需要注意的是，可能存在for语句参数为空的清空，所以生成跳转的前提是在这部分有东西在的情况，否则会进入死循环。

短路求值

有两种情况需要使用短路求值：

```
1. if(a&&b) // a is false
2. if(a||b) // b is true
```

首先，我分析`analyseLOrExp`时，每个`analyseLAndExp`后面都会跟着一个`JNZ`，用于检测如果条件为假。如果是，则跳转到 if 体标签。同时，我生成了 `cond` 标签，为 `analyseLAndExp` 做准备。

```
    private void analyseLOrExp(ArrayList<Token> exp, String from) {
        // LOrExp → LAndExp | LOrExp '||' LAndExp
        Exps exps = divideExp(exp, new ArrayList<>
(List.of(String.valueOf(Word.OR)))); // ||
        int j = 0;
        for (int i = 0; i < exps.getTokens().size(); i++) {
            ArrayList<Token> exp1 = exps.getTokens().get(i);
            String label = labelGenerator.generateLabel("cond_" + i);
            analyseLAndExp(exp1, from, label); // LAndExp
            addCode(Operator.LABEL, label);
            if (j > 0) {
                addCode(Operator.OR);
            }
            if (exps.getTokens().size() > 1 && i != exps.getTokens().size() - 1)
{
                if (from.equals(String.valueOf(Word.IFTK))) {
                    addCode(Operator.JNZ, ifLabels.get(ifLabels.size() -
1).get("if_block"));
                } else if(from.equals(String.valueOf(Word.FORTK))){
                    addCode(Operator.JNZ, forLabels.get(forLabels.size() -
1).get("for_block"));
                }
            }
            grammar.add("<LOrExp>");
            if (j < exps.getSymbols().size()) {
                grammar.add(exps.getSymbols().get(j++).toString());
            }
        }
    }
```

在`analyseLAndExp`中，每个`analyseEqExp`后面都会跟着一个`JZ`，用于检测条件是否满足是真的。如果是的话，跳转到刚才设置的`cond`标签。

```
    private void analyseLAndExp(ArrayList<Token> exp, String from, String label)
{
    // LAndExp → EqExp | LAndExp '&&' EqExp
        Exps exps = divideExp(exp, new ArrayList<>
(List.of(String.valueOf(Word.AND)))); // &&
        int j = 0;
        for (int i = 0; i < exps.getTokens().size(); i++) {
            ArrayList<Token> exp1 = exps.getTokens().get(i);
            analyseEqExp(exp1); // EqExp
            if (j > 0) {
                addCode(Operator.AND);
```

```
            }
            if (exps.getTokens().size() > 1 && i != exps.getTokens().size() - 1)
{
                if (from.equals(String.valueOf(Word.IFTK))) {
                    addCode(Operator.JZ, label);
                } else {
                    addCode(Operator.JZ, label);
                }
            }
            grammar.add("<LAndExp>");
            if (j < exps.getSymbols().size()) {
                grammar.add(exps.getSymbols().get(j++).toString());
            }
        }
    }
```

参考资料

1. “课程资料” ——《2023编译技术实验说明》
2. 参考Github上往届代码 —— https://github.com/acsoto/BUAA-Compiler （Pcode）