SysY编译器设计文档

1、参考编译器介绍

在整体设计自己的编译器之前，我主要参考、阅读了2020级陶思远学长的编译器，借鉴了该编译器在代码生成之前的总体架构。

1.1 参考编译器总体结构

参考代码的总体结构分为Exceptions、FrontEnd、Ir、utils和Compiler.java这几个部分。其中

- Compiler.java是总的编译器，也是项目的主类。
- Exceptions专用于错误处理，里面包含有各种错误的处理的方式和类型。
- FrontEnd是编译器前端，包含了词法分析Lexer、语法分析Parser、SysY规定的Token还有他们对应的控制器等。
- Lr专门用于代码生成。下属ClangAST、LLVMIR和Generator.java。Generator.java是代码生成的主体部分，所需的基础结构是AST语法树，即ClangAST的产生结构。ClangAST包含了AST、SymbolTable和ASTMaker。AST是AST语法树的组成成分，ASTMaker.java就是生成AST语法树组成成分的主体代码，SymbolTable是源代码的符号表，在搭建AST树的时候生成。LLVMIR是llvm_ir的各个语法成分，包括了各种ir语句和标签系统等。
- utils是通用的工具类，包含了一些编译器可用的工具，比如读写文件工具。

1.2 参考编译器接口设计

```
public static void IrTest() throws ParserError, IrError {
    Lexer lexer = new Lexer();
    LexerController lexerController = new LexerController(lexer);
    Parser parser = new Parser(lexerController);
    ParserNode root = parser.run();
    ASTMaker ast = ASTMaker.getInstance(root);
    Root astRoot = ast.HandleRoot();
    Generator generator = Generator.getInstance(ASTMaker.getRoot(), astRoot);
    Model model = generator.run();
    System.out.println(model.toString());
    System.err.println("finished");
}
```

首先新建一个Lexer实例，作为属性构造LexerController实例，并开始读入源代码进行词法分析。结果保存在LexerController中，将其导入parser实例中，调用`ParserNode root = parser.run();`进行语法分析，构造初级语法树。随后构造AST语法树和构造符号表，将结果导入Generator实例进行最终的llvm_ir代码生成。

1.3 参考编译器文件组织

```
D:.
|  Compiler.java
|
├─Exceptions
|      ErrorType.java
|      ExceptionLog.java
|      IrError.java
|      LexerError.java
|      MyException.java
|      ParserError.java
|
├─FrontEnd
|      Lexer.java
|      LexerController.java
|      Parser.java
|      ParserNode.java
|      Token.java
|      TokenType.java
|
├─Ir
|   |  Generator.java
|   |
|   ├─ClangAST
|   |   |  ASTMaker.java
|   |   |
|   |   ├─AST
|   |   |      ArrDeclStmt.java
|   |   |      AssignStmt.java
|   |   |      BinaryOperator.java
|   |   |      BreakStmt.java
|   |   |      CallExpr.java
|   |   |      CompoundStmt.java
|   |   |      ComputeStmt.java
|   |   |      CondStmt.java
|   |   |      ContinueStmt.java
|   |   |      DeclRefExpr.java
|   |   |      DeclStmt.java
|   |   |      Expr.java
|   |   |      FuncDeclStmt.java
|   |   |      IfStmt.java
|   |   |      IrController.java
|   |   |      LeafStmt.java
|   |   |      MyInteger.java
|   |   |      Op.java
|   |   |      ReturnStmt.java
|   |   |      Root.java
|   |   |      Stmt.java
|   |   |      VarDeclStmt.java
|   |   |      WhileStmt.java
|   |   |
```

```
│   │   └─SymbolTable
│   │           CType.java
│   │           FuncDecl.java
│   │           FuncTable.java
│   │           ParamPT.java
│   │           SymbolTable.java
│   │           VarDecl.java
│   │           VarTable.java
│   │
│   └─LLVMIR
│       ├─Component
│       │       GlobalFunc.java
│       │       GlobalVar.java
│       │       Model.java
│       │
│       ├─IRInstr
│       │       AddIR.java
│       │       AllocaIR.java
│       │       AndIR.java
│       │       ArithIR.java
│       │       BrIR.java
│       │       CallIR.java
│       │       FuncDeclareIR.java
│       │       FuncDeclIR.java
│       │       GetElementPtrIR.java
│       │       GlobalVarIR.java
│       │       IcmpIR.java
│       │       Instr.java
│       │       Label.java
│       │       LoadIR.java
│       │       LogicIR.java
│       │       MulIR.java
│       │       RetIR.java
│       │       SdivIR.java
│       │       SremIR.java
│       │       StoreIR.java
│       │       SubIR.java
│       │       ZextIR.java
│       │
│       └─LabelSystem
│               IfUnit.java
│               LabelStack.java
│               LabelUnit.java
│               LabelValue.java
│               WhileUnit.java
│
└─utils
        Pair.java
        ReadFile.java
        Regex.java
```

2.编译器总体设计

2.1 总体结构

我的编译器项目结构主要分为五个部分：

- AST。AST包含了BuildAST.java和各种AST语法树语法成分，BuildAST获取初级语法树的语法成分，构建AST语法树便于代码生成。
- Frontend。编译器前端，包括以下成分：
    - ExceptionController：收集全部的异常错误，可以按行号进行排序
    - Grammar：进行语法分析
    - GrammarNode：语法分析节点，进行语法分析的具体构造对象
    - HandleException：处理异常错误，返回MyException实例对象。
    - IoFile：文件读入输出工具
    - Lexer：词法分析器，调用源代码输入工具，进行最初的词法分析。
    - LexType：词法分析种类，对标SysY词法规定。
    - MyException：异常错误类，保留错误处理所需的详细信息。
    - Token：词法节点，包含了词法种类、词法原值、在源文件的位置等。
    - VisitAST：扫描语法树，获取符号表和进行错误处理。
- Ir。编译器代码生成部分，包含以下成分：
    - Component。Model的组成成分，包括了GlobalFunc：函数，GlobalVar：全局变量和Model：包括以上结构，这就形成了代码生成的总体结构。
    - IRInstr。包括了为了代码生成所归纳的所有用得到的Ir类，通过生成这些Ir类可以生成最后的llvm_ir代码。
    - Generator：用于生成上述Ir类，进行最后的代码生成。
- SymbolTablePackage：包含了所有的构造符号表的相关类。
- Compiler：编译器主类。

2.2 接口设计

```
public static void main(String[] args) {
        ExceptionController ec = new ExceptionController();
        ArrayList<Token> tokens = getLexer(ec);
        GrammarNode ast = getGrammar(tokens,ec);
        VisitAST visitAST = new VisitAST(ast,ec);
        BlockSymbolTable table = visitAST.getSymbolTableAndHandleError();
        if (!visitAST.getOutputError().equals("")) {
            return;
        }
        BuildAST buildAST = new BuildAST(ast);
        ASTroot asTroot = buildAST.getRoot();
        Generator generator = new Generator(table,asTroot);
        Model model = generator.run();
    }
```

首先异常错误收集器ExceptionController ec贯彻词法分析、语法分析、错误处理的全部分析过程。

getLexer进行了语法分析。

```
public static ArrayList<Token> getLexer(ExceptionController ec) {
        StringBuilder stringBuilder = IoFile.readFileByBytes("testfile.txt");
        if (stringBuilder != null) {
            // source:源程序字符串
            String source = stringBuilder.toString();
            Lexer lexer = new Lexer(source,ec);
            lexer.begin();
            Token token = lexer.next();
            StringBuilder output = new StringBuilder();
            ArrayList<Token> tokens = new ArrayList<>();
            while (!token.getToken().equals("\0")) {
                // System.out.println(token.getLexType() + " " + token.getToken());
                output.append(token.getLexType()).append("
").append(token.getToken()).append('\n');
                tokens.add(token);
                token = lexer.next();
            }
            IoFile.outputContentToFile_testTokens(output.toString());
            return tokens;
        }
        return null;
    }
```

`StringBuilder stringBuilder = IoFile.readFileByBytes("testfile.txt")` 该代码读入了testfile.txt的源代码。随后传入lexer改造函数实例，`Token token = lexer.next();` 在while中反复调用，获取每个词法成分，直到检测到\0代表结束。最后返回tokens词法分析结果。

`GrammarNode ast = getGrammar(tokens,ec);` 将上文获得的词法分析结果进行进一步的语法分析。

```
public static GrammarNode getGrammar(ArrayList<Token> tokens,ExceptionController ec) {
        Grammar grammar = new Grammar(tokens,ec);
        GrammarNode ast = grammar.grammarStart();
        return ast;
    }
```

grammarStart方法启动语法分析，最终返回初级语法树ast

```
public GrammarNode grammarStart() {
        GrammarNode main = getCompUnit();
        StringBuilder output = new StringBuilder();
        outputParser(main,output);
        IoFile.outputContentToFile(output.toString());
        return main;
    }
```

main节点是总的节点，包含了全局变量、函数和main函数。

```
public GrammarNode getCompUnit() {
        ArrayList<GrammarNode> grammarNodes = new ArrayList<>();
        //先检测是不是全局声明
        while (tokenLength - nowIndex > 2 &&
                (nowTokenTypeCompare(nowIndex,LexType.CONSTTK) ||
                        (nowTokenTypeCompare(nowIndex,LexType.INTTK) &&
!nowTokenTypeCompare(nowIndex+1,LexType.MAINTK) &&
!nowTokenTypeCompare(nowIndex+2,LexType.LPARENT)
                        )
                )
        ) {
            //第一层：确保不越界
            //第二层，如果开头是const，肯定是全局声明
            //第三层，如果是int开头，后面紧跟着不能是main，其次后面第二个不能是左括号
            //这样就可以确保是全局变量声明
            grammarNodes.add(getDecl());
        }

        //检测函数
        while (tokenLength - nowIndex > 2 &&
                (nowTokenTypeCompare(nowIndex,LexType.INTTK) ||
nowTokenTypeCompare(nowIndex,LexType.VOIDTK))
                    && !nowTokenTypeCompare(nowIndex+1,LexType.MAINTK)
                        && nowTokenTypeCompare(nowIndex+2,LexType.LPARENT)
        ) {
            grammarNodes.add(getFuncDef());
        }

        grammarNodes.add(getMainFuncDef());
        return new GrammarNode("CompUnit",grammarNodes);
    }
```

这就是总的语法节点main的结构，详细的语法成分通过get形函数进行递归分析，比如getDecl，getFuncDef，get函数后面跟着语法成分名字几乎完全按照SysY规定的文法结构，比较清楚直观。

获得ast后进行错误处理，错误处理的主体成分在visitAST里，visitAST类中一边进行错误处理，一边进行构造符号表。

```
VisitAST visitAST = new VisitAST(ast,ec);
BlockSymbolTable table = visitAST.getSymbolTableAndHandleError();
if (!visitAST.getOutputError().equals("")) {
    return;
}
```

通过`getSymbolTableAndHandleError`方法进行错误处理和构造符号表后，如果有错误，即`visitAST.getOutputError()`不是空串，就直接return不需要继续分析。

```
BuildAST buildAST = new BuildAST(ast);
ASTroot asTroot = buildAST.getRoot();
```

这两行代码构建了AST语法树

```
public ASTroot getRoot() {
        ArrayList<VarDeclStmt> globalVarDeclStmt = new ArrayList<>();
        ArrayList<FuncDeclStmt> funcDeclStmts = new ArrayList<>();
        FuncDeclStmt mainFuncDef = null;
        for (GrammarNode rootNode : main.getNodes()) {
            if (compareNode(rootNode,"Decl")) {

getVarDeclStmt(true,false,globalVarDeclStmt,rootNode.getNodes().get(0));
            } else if (compareNode(rootNode,"FuncDef")){
                getFuncDef(funcDeclStmts, rootNode);//最后一个定义的函数是main
            } else if (compareNode(rootNode,"MainFuncDef")) {
                mainFuncDef = getMainFuncDef(rootNode);
            }
        }

        return new ASTroot(globalVarDeclStmt,funcDeclStmts,mainFuncDef);
    }
```

最后进行代码生成，把AST语法树结构导入Generator。

```
Generator generator = new Generator(table,asTroot);
Model model = generator.run();
```

2.3 文件结构

D:\OO\COMPILE\CPY-S-COMPILER\SRC | Compiler.java | src.zip | +---AST | AddExpStmt.java | ArrDeclStmt.java | AssignStmt.java | ASTroot.java | BasicBlock.java | BreakStmt.java | BuildAST.java | CallExpr.java | CompoundStmt.java | ComputeStmt.java | CondStmt.java | ContinueStmt.java | EqExp.java | ForStmt.java | FuncDeclStmt.java | IfStmt.java | LAndExp.java | LorExp.java | LVal.java | MulExpStmt.java | Op.java | RelExp.java | ReturnStmt.java | Stmt.java | UnaryExpStmt.java | VarDeclStmt.java | +---Frontend | ExceptionController.java | Grammar.java | GrammarNode.java | HandleException.java | IoFile.java | Lexer.java | LexType.java | MyException.java | Token.java | VisitAST.java | +---Ir | | Generator.java | | | +---Component | | GlobalFunc.java | | GlobalVar.java | | Model.java | | | ---IRInstr | AddIR.java | AllocaIR.java | BrIR.java | CallIR.java | FuncDeclIR.java | GetelementptrIR.java | GlobalVarIr.java | IcmpIR.java | Instr.java | JumpUnit.java | LabelLineIR.java | LibFunctionIR.java | LoadIR.java | MulIR.java | Pair.java | ReturnIR.java | SdivIR.java | SremIR.java |

StoreIR.java | SubIR.java | ZextIR.java | ---SymbolTablePackage BlockSymbolTable.java FuncSymbolTable.java SymbolTable.java SymbolType.java Value.java VarSymbolTable.java

3、词法分析设计

3.1 编码前的设计

词法分析是编译器的第一个环节，因此编码前的设计基本上就是搭框架以及写导入源文件内容的工具类代码。

```
public static StringBuilder readFileByBytes(String fileName) {
        File file = new File(fileName);
        FileInputStream in = null;
        StringBuilder stringBuilder = new StringBuilder();
        try {
            in = new FileInputStream(file);
            int tmp;
            while ((tmp = in.read()) != -1) {
                // System.out.print((char)tmp);
                stringBuilder.append((char)tmp);
            }
            stringBuilder.append('\0');
            return stringBuilder;
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }
```

lexer准备以分隔符为界，重点关注第一个字符，达到区分的效果。

3.2 编码完成之后的修改

代码重点在于：

- 编写保留字表

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

- 读入字符函数next

```
public Token next() {
        while(isSpace()) {
            getChar();
        }
        if (isDigit()) {
            return getDigit();
        }
        if (isLetter() || c =='_') {
            return getIdentOrReserveWord();
        }

        if (c == '!') {
            getChar();
            if (c == '=') {
                getChar();
                return new Token("!=", line, LexType.NEQ);
            } else {
                return new Token("!",line,LexType.NOT);
            }
        }

        if (c == '=') {
            getChar();
            if (c == '=') {
                getChar();
                return new Token("==",line,LexType.EQL);
            } else {
                return new Token("=",line,LexType.ASSIGN);
            }
        }

        if (c == '+') {
            getChar();
            return new Token("+",line,LexType.PLUS);
        }

        if (c =='-') {
            getChar();
            return new Token("-",line,LexType.MINU);
        }

        if(c == '*') {
            getChar();
            return new Token("*",line,LexType.MULT);
        }
        if(c == '/') {
            getChar();
            if (c == '/') {
                solveLineComments();
                return next();
```

```
            } else if(c == '*') {
                solveBlockComments();
                return next();
            } else {
                return new Token("/",line,LexType.DIV);
            }
        }

        if (c == '&') {
            getChar();
            if (c == '&') {
                getChar();
                return new Token("&&",line,LexType.AND);
            }
        }

        if (c == '|') {
            getChar();
            if (c == '|') {
                getChar();
                return new Token("||", line,LexType.OR);
            }
        }

        if (c == ';') {
            getChar();
            return new Token(";",line,LexType.SEMICN);
        }

        if (c == '"') {
            return solveFormatString();
        }

        if (c == '%') {
            getChar();
            return new Token("%",line,LexType.MOD);
        }

        if (c == ',') {
            getChar();
            return new Token(",",line,LexType.COMMA);
        }

        if (c == '<') {
            getChar();
            if (c == '=') {
                getChar();
                return new Token("<=",line,LexType.LEQ);
            } else {
                return new Token("<",line,LexType.LSS);
            }
```

```
        }

        if (c == '>') {
            getChar();
            if (c == '=') {
                getChar();
                return new Token(">=",line,LexType.GEQ);
            } else {
                return new Token(">",line,LexType.GRE);
            }
        }

        if (c == '(') {
            getChar();
            return new Token("(",line,LexType.LPARENT);
        }

        if (c == ')') {
            getChar();
            return new Token(")",line,LexType.RPARENT);
        }

        if (c == '[') {
            getChar();
            return new Token("[",line,LexType.LBRACK);
        }

        if (c == ']') {
            getChar();
            return new Token("]",line,LexType.RBRACK);
        }

        if (c == '{') {
            getChar();
            return new Token("{",line,LexType.LBRACE);
        }

        if (c == '}') {
            getChar();
            return new Token("}",line,LexType.RBRACE);
        }

        if (c == '\0') {
            return new Token("\0",line,null);
        }
        //debug
        System.out.println("no found!");
        return null;
    }
```

最终结果是将分隔符分开的字符串识别出token，集合tokens序列将为下面的语法分析准备。

4、语法分析设计

4.1 编码前的设计

```
public Grammar(ArrayList<Token> tokens, ExceptionController ec) {
        this.tokens = tokens;
        nowIndex = 0;
        tokenLength = tokens.size();
        this.ec = ec;
    }
```

除了储存词法分析得到的tokens，nowIndex用于记录下标，遍历的同时避免因为ll(k)文法

越界。

以getCompUnit为开始，进行全局变量-函数-main函数的识别，编写get型函数，完全按照文法进行递归识别。每个get型函数将会返回一个GrammarNode，这个GrammarNode拥有下属GrammarNode列表，拥有自己的名字——严格按照SysY文法名。这样就按照文法搭建了初级语法树。

4.2 编码完成之后的修改

以Stmt为例子：

```
public GrammarNode getStmt() {
        ArrayList<GrammarNode> grammarNodes = new ArrayList<>();
        if (whetherOutOfBound()) {
            if (nowTokenTypeCompare(nowIndex,LexType.SEMICN)) {
                grammarNodes.add(getLeaf(LexType.SEMICN));
            } else if (nowTokenTypeCompare(nowIndex,LexType.LBRACE)) {
                grammarNodes.add(getBlock());
            } else if (nowTokenTypeCompare(nowIndex,LexType.IFTK)) {
                solveIfStmt(grammarNodes);
            } else if (nowTokenTypeCompare(nowIndex,LexType.FORTK)) {
                solveForStmt(grammarNodes);
            } else if (nowTokenTypeCompare(nowIndex,LexType.BREAKTK)) {
                grammarNodes.add(getLeaf(LexType.BREAKTK));
                GrammarNode tmpNode;
                if ((tmpNode = getLeaf(LexType.SEMICN)) != null) {
                    grammarNodes.add(tmpNode);
                } else {
                    ec.addException(returnException(tokens.get(nowIndex-
1),"i",LexType.SEMICN));
                }
            } else if (nowTokenTypeCompare(nowIndex,LexType.CONTINUETK)) {
                grammarNodes.add(getLeaf(LexType.CONTINUETK));
                GrammarNode tmpNode;
                if ((tmpNode = getLeaf(LexType.SEMICN)) != null) {
                    grammarNodes.add(tmpNode);
                } else {
                    ec.addException(returnException(tokens.get(nowIndex-
1),"i",LexType.SEMICN));
                }
            } else if (nowTokenTypeCompare(nowIndex,LexType.RETURNTK)) {
                grammarNodes.add(getLeaf(LexType.RETURNTK));
                if (whetherOutOfBound() && !
(nowTokenTypeCompare(nowIndex,LexType.SEMICN) ||
nowTokenTypeCompare(nowIndex,LexType.RBRACE))){
                    grammarNodes.add(getExp());
                }
                GrammarNode tmpNode;
                if ((tmpNode = getLeaf(LexType.SEMICN)) != null) {
                    grammarNodes.add(tmpNode);
                } else {
                    ec.addException(returnException(tokens.get(nowIndex-
1),"i",LexType.SEMICN));
                }
            } else if (nowTokenTypeCompare(nowIndex,LexType.PRINTFTK)) {
                solvePrintfStmt(grammarNodes);
            } else {
                //LVal '=' Exp ';'
                //Exp ';' //有Exp的情况 只有一个;的情况前面写了
                // LVal '=' 'getint''('')'';'
                if (tokenLength - nowIndex > 1 && searchAssignForStmt()) {
```

```
                    //说明是LVal的情况
                    grammarNodes.add(getLVal());
                    grammarNodes.add(getLeaf(LexType.ASSIGN));
                    if (whetherOutOfBound() &&
nowTokenTypeCompare(nowIndex,LexType.GETINTTK)) {
                        grammarNodes.add(getLeaf(LexType.GETINTTK));
                        grammarNodes.add(getLeaf(LexType.LPARENT));

                        GrammarNode tmpNode;
                        if ((tmpNode = getLeaf(LexType.RPARENT)) != null) {
                            grammarNodes.add(tmpNode);
                        } else {
                            ec.addException(returnException(tokens.get(nowIndex-
1),"j",LexType.RPARENT));
                        }

                        GrammarNode tmpNode2;
                        if ((tmpNode2 = getLeaf(LexType.SEMICN)) != null) {
                            grammarNodes.add(tmpNode2);
                        } else {
                            ec.addException(returnException(tokens.get(nowIndex-
1),"i",LexType.SEMICN));
                        }

                    } else {
                        grammarNodes.add(getExp());
                        GrammarNode tmpNode;
                        if ((tmpNode = getLeaf(LexType.SEMICN)) != null) {
                            grammarNodes.add(tmpNode);
                        } else {
                            ec.addException(returnException(tokens.get(nowIndex-
1),"i",LexType.SEMICN));
                        }
                    }
                } else {
                    // 说明是Exp的情况
                    grammarNodes.add(getExp());
                    GrammarNode tmpNode;
                    if ((tmpNode = getLeaf(LexType.SEMICN)) != null) {
                        grammarNodes.add(tmpNode);
                    } else {
                        ec.addException(returnException(tokens.get(nowIndex-
1),"i",LexType.SEMICN));
                    }
                }
            }
        }
        return new GrammarNode("Stmt",grammarNodes);
    }
```

按照这样的方式搭建初级语法树后，进行递归输出：首先遍历GrammarNode的自身的列表，递归遍历完之后，输出自身文法名字。

```
public void outputParser(GrammarNode main,StringBuilder output) {
        ArrayList<GrammarNode> nodes = main.getNodes();
        for (GrammarNode node : nodes) {
            if (node.getNodes() == null) { //查询到叶节点
                Token token = (Token)node;
                output.append(token.getLexType()).append("
").append(token.getToken()).append('\n');
                //System.out.println(token.getLexType() + " " + token.getToken());
            }
            else {
                outputParser(node,output);
            }
        }
        if (main.getNodeName().equals("BlockItem") ||
                main.getNodeName().equals("Decl") ||
                    main.getNodeName().equals("BType")
        ) {
            return;
        } else {
            //System.out.println();
            //Frontend.IoFile.outputContentToFile();
            output.append('<').append(main.getNodeName()).append('>').append('\n');
        }
    }
```

5、错误处理设计

5.1 编码前的设计

设计一个visitAST方法，遍历语法分析器得到的初级语法树，构建符号表，并在遍历、构造符号表的同时，在可能出现错误的地方插入错误处理方法，~~~这难免会使代码变得屎山~~。

5.2 编码完成之后的修改

这里主要以处理函数为例，展示符号表搭建

```
public void handleFuncDef(BlockSymbolTable fatherTable, GrammarNode node) {
        int funcLine = 0;
        String returnType = null;
        String funcName = null;
        FuncSymbolTable funcSymbolTable = null;
        BlockSymbolTable paras = null; //函数块表
        Token token = null;
        for (GrammarNode funcDefNode : node.getNodes()) {
            if (funcDefNode.getNodeName().equals("FuncType")) {
                token = (Token) funcDefNode.getNodes().get(0);
                returnType = token.getToken();
            } else if (funcDefNode instanceof Token &&
((Token)funcDefNode).compareLexType(LexType.IDENFR)) {
                token = (Token) funcDefNode;
                funcName = token.getToken();
                funcLine = token.getLineNum();
                paras = new BlockSymbolTable(nowLevel,funcLine,"paras",fatherTable);
                funcSymbolTable = new
FuncSymbolTable(nowLevel,SymbolType.function,funcLine,funcName,returnType,paras);
                addSymbol(fatherTable,funcSymbolTable); //先加入符号表，防止定义递归调
用，后面只有para，调用setpara方法设置
                nowLevel++; //
            } else if (funcDefNode.getNodeName().equals("FuncFParams")) {
                boolean hasFuncPara = false; //最后一个函数参数录入符号表
                String paraName = null;
                int paraLine = 0;
                int bracket = 0;
                int n1 = 0;
                int n2 = 0;
                for (GrammarNode para : funcDefNode.getNodes()) {
                    if (para.getNodeName().equals("FuncFParam")) {
                        bracket = 0; //更新brack
                        n1 = 0;
                        n2 = 0;
                        for (GrammarNode paraDetail : para.getNodes()) {
                            if (!(paraDetail instanceof Token)) {
                                continue;
                            }
                            token = (Token) paraDetail;
                            if (token.compareLexType(LexType.IDENFR)) {
                                paraName = token.getToken();
                                paraLine = token.getLineNum();
                                hasFuncPara = true;
                            } else if (token.compareLexType(LexType.LBRACK)) {
                                bracket++;
                            }/* else if
(token.compareLexType(Frontend.LexType.INTCON)) { //不管数组值
                                if (bracket == 1) {
                                    n1 = Integer.parseInt(token.getToken());
                                } else if(bracket == 2){
```

```
                                    n2 = Integer.parseInt(token.getToken());
                                }
                            }*/
                        }
                    }  else if (para instanceof Token && ((Token)
para).compareLexType(LexType.COMMA)) {
                        // 检测到逗号的时候
                        VarSymbolTable paraVar = new
VarSymbolTable(nowLevel,SymbolType.var,paraLine,false,paraName,n1,n2,bracket,true,fals
e);
                        addSymbol(paras,paraVar);
                    }
                }
                if (hasFuncPara) { // 最后一个参数
                    VarSymbolTable paraVar = new
VarSymbolTable(nowLevel,SymbolType.var,paraLine,false,paraName,n1,n2,bracket,true,fals
e);
                    addSymbol(paras,paraVar);
                }
                funcSymbolTable.setParams(paras);
            } else if (funcDefNode.getNodeName().equals("Block")) {
                handleBlock(funcDefNode,returnType,paras);
            }
        } // for遍历funcDef
    }// 函数主体
```

当遇到函数idenfr后，创建符号表：

```
paras = new BlockSymbolTable(nowLevel,funcLine,"paras",fatherTable);
```

同时将函数加入总表中：

```
funcSymbolTable = new
FuncSymbolTable(nowLevel,SymbolType.function,funcLine,funcName,returnType,paras);
                addSymbol(fatherTable,funcSymbolTable); //先加入符号表，防止定义递归调
用，后面只有para，调用setpara方法设置
```

这里录入函数形参：

```
else if (para instanceof Token && ((Token) para).compareLexType(LexType.COMMA)) {
                        // 检测到逗号的时候
                        VarSymbolTable paraVar = new
VarSymbolTable(nowLevel,SymbolType.var,paraLine,false,paraName,n1,n2,bracket,true,fals
e);
                        addSymbol(paras,paraVar);
                    }
```

6、代码生成设计

6.1 编码前的设计

生成代码之前，我的想法在初级语法树的基础上，先生成规范的AST语法树，编写BuildAST类和众多语句类，搭建AST语法树，这样在进行真正的代码生成的时候，面对的就是已经经过整理的、符合人类简述的语句。

生成AST树后，根据语句进行代码生成。搭建IR类，这些类可以保存llvm_ir语句的详细信息，重写的genIr方法可以根据保存的信息生成最终的ir语句。

6.2 编码完成之后的修改

代码生成主要难点：

- 数组。数组难点在于可能会遇到不同维度的数组，处理这些数组需要列举所有可能；另外，如果数组是函数形参的时候，分配的是下一维指针。这就相比直接调用更加麻烦——要一次调用、少一层维度。

函数形参：

```
public GlobalFunc handleGlobalFunctions(FuncDeclStmt funcDeclStmt,SymbolTable
symbolTable) {
        this.blocks = funcDeclStmt.getBlocks();
        nowReg = 0;//函数从0开始
        String name = funcDeclStmt.getName();
        String type = funcDeclStmt.getReturnType();
        FuncSymbolTable funcSymbolTable = (FuncSymbolTable) symbolTable;
        FuncDeclIR funcDeclIR = funcDeclStmt.getIr();
        BlockSymbolTable paras = funcSymbolTable.getParams();
        if (paras != null) {
            for (SymbolTable para : paras.getSymbolTables()) {
                if (para.compareType(SymbolType.var) &&
((VarSymbolTable)para).isParameter()) {
                    int tag = getReg();
                    para.setTag("%"+tag);
                } else {
                    break;
                }
            }
        }
        String funTag = "%" + getReg();
        BasicBlock funBasicBlock = new BasicBlock(funTag,0);
        this.nowBasicBlock = funBasicBlock;
        /*
        if (paras != null) {
            for (SymbolTable para : paras.getSymbolTables()) {
                if (para.compareType(SymbolType.var) &&
((VarSymbolTable)para).isParameter()) {
                    VarSymbolTable varSymbolTable = (VarSymbolTable)para;//屎山。。。。
                    String tag = "%" + getReg();
                    //这里的思路是，形参先把tag设置成他的值寄存器（实际上应该要存储地址寄存
器但是还没有分配），分配了地址寄存器后，再改回来
                    if (varSymbolTable.getBracket() == 0) {
                        this.nowBasicBlock.inputInstr(new AllocaIR(tag, "i32"));
                        this.nowBasicBlock.inputInstr(new StoreIR(para.getTag(), tag,
"i32"));
                        para.setTag(tag);//改回来
                    } else if (varSymbolTable.getBracket() == 1) {
                        this.nowBasicBlock.inputInstr(new AllocaIR(tag, "i32*"));
                        this.nowBasicBlock.inputInstr(new StoreIR(para.getTag(), tag,
"i32*"));
                    } else if (varSymbolTable.getBracket() == 2) {

                    }
                }
            }
        }
         */
        ArrayList<VarDeclStmt> parasOfFuncDeclStmt = funcDeclStmt.getParas();
        if (parasOfFuncDeclStmt != null) {
```

```
            for (VarDeclStmt varDeclStmt : parasOfFuncDeclStmt) {
                String tag = "%" + getReg();
                VarSymbolTable para = paras.searchVar(varDeclStmt.getName(), paras,
true);
                if (!(varDeclStmt instanceof ArrDeclStmt arrDeclStmt)) {
                    this.nowBasicBlock.inputInstr(new AllocaIR(tag, "i32"));
                    this.nowBasicBlock.inputInstr(new StoreIR(para.getTag(), tag,
"i32"));
                } else if (arrDeclStmt.getBracket() == 1) {
                    para.setN2(arrDeclStmt.getN2());
                    this.nowBasicBlock.inputInstr(new AllocaIR(tag, "i32*"));
                    this.nowBasicBlock.inputInstr(new StoreIR(para.getTag(), tag,
"i32*"));
                } else if (arrDeclStmt.getBracket() == 2) {
                    para.setN1(arrDeclStmt.getN1());
                    para.setN2(arrDeclStmt.getN2());
                    String arrType = getType(0, arrDeclStmt.getN2()) + "*";
                    this.nowBasicBlock.inputInstr(new AllocaIR(tag, arrType));
                    this.nowBasicBlock.inputInstr(new StoreIR(para.getTag(), tag,
arrType));
                }
                para.setTag(tag);//改回来
            }
        }
        handleCompoundStmt(funcDeclStmt.getStmts(),paras);
        this.nowBasicBlock.inputInstr(new ReturnIR(true,null));
        GlobalFunc globalFunc = new GlobalFunc(funcDeclIR,this.blocks);
        blocks.add(this.nowBasicBlock);
        return globalFunc;
    }
```

直接调用：注意是type2的代码部分：

```
public void handleUnaryExpStmt(UnaryExpStmt unaryExpStmt, BlockSymbolTable
fatherTable) {
        int type = unaryExpStmt.getType();
        if (type == 1) { //(Exp)
            handleComputerStmt(unaryExpStmt.getComputeStmt(),fatherTable);
        } else if (type == 2) { //LVal
            LVal lVal = unaryExpStmt.getlVal();
            String name = lVal.getLValName();
            VarSymbolTable var = fatherTable.searchVar(name,fatherTable,false);
            String tag = var.getTag();
            /*
                2维
                a 0 0
                a[1] 0 0    0 1
                a[1][1] 0 1   0 1
                1维
                b 0 0
                b[1] 0 1
                根本原因在于，一维数组和取一维数组中的整数返回的类型是一样的，因此出现了这
样的情况
             */
            if (var.getBracket() == 0) { // 非数组
                this.nowBasicBlock.inputInstr(new LoadIR("%" + getReg(), tag, "i32"));
            } else { //数组
                String storeReg;
                String arrType;
                String arrPointer = var.getTag();
                ArrayList<String> indexs;
                if (var.isParameter()) { //因为原来是指针！所以必须load一次才可以用，这样
变成数组指针！注意变为数组指针后，与下面的区别
                    if (var.getBracket() == 1) {
                        storeReg = "%" + getReg();
                        this.nowBasicBlock.inputInstr(new LoadIR(storeReg, arrPointer,
"i32*"));
                        //之后就是一维数组
                        if (lVal.getBracket() == 1) {
                            indexs = new ArrayList<>();
                            ComputeStmt n2 = lVal.getN2();
                            handleComputerStmt(n2,fatherTable);
                            AddExpStmt addExpStmt = n2.getAddExpStmt();
                            if (addExpStmt.isNum()) {
                                indexs.add(String.valueOf(addExpStmt.getValue()));
                            } else {
                                indexs.add("%" + (nowReg - 1));
                            }
                            String integerPointer = "%" + getReg();
                            this.nowBasicBlock.inputInstr(new
GetelementptrIR(integerPointer,"i32",storeReg,indexs));
                            this.nowBasicBlock.inputInstr(new LoadIR("%" + getReg(),
integerPointer, "i32"));
```

```
                        } else { //0
                            indexs = new ArrayList<>();
                            indexs.add(String.valueOf(0));
                            String integerPointer = "%" + getReg();
                            this.nowBasicBlock.inputInstr(new
GetelementptrIR(integerPointer,"i32",storeReg,indexs));
                        }
                    } else { // 2
                        storeReg = "%" + getReg();
                        arrType = getType(0, var.getN2());
                        this.nowBasicBlock.inputInstr(new
LoadIR(storeReg,arrPointer,arrType + "*"));//注意这里，即使是二维数组，但是是一个指针
                        if (lVal.getBracket() == 2) {
                            indexs = new ArrayList<>();
                            ComputeStmt n1 = lVal.getN1();
                            handleComputerStmt(n1,fatherTable);
                            AddExpStmt addExpStmt = n1.getAddExpStmt();
                            if (addExpStmt.isNum()) {
                                indexs.add(String.valueOf(addExpStmt.getValue()));
                            } else {
                                indexs.add("%" + (nowReg - 1));
                            }
                            ComputeStmt n2 = lVal.getN2();
                            handleComputerStmt(n2,fatherTable);
                            addExpStmt = n2.getAddExpStmt();
                            if (addExpStmt.isNum()) {
                                indexs.add(String.valueOf(addExpStmt.getValue()));
                            } else {
                                indexs.add("%" + (nowReg - 1));
                            }
                            String integerPointer = "%" + getReg();
                            this.nowBasicBlock.inputInstr(new
GetelementptrIR(integerPointer,arrType,storeReg,indexs));
                            this.nowBasicBlock.inputInstr(new LoadIR("%" + getReg(),
integerPointer, "i32"));
                        } else if (lVal.getBracket() == 1) {
                            indexs = new ArrayList<>();
                            ComputeStmt n2 = lVal.getN2();
                            handleComputerStmt(n2,fatherTable);
                            AddExpStmt addExpStmt = n2.getAddExpStmt();
                            if (addExpStmt.isNum()) {
                                indexs.add(String.valueOf(addExpStmt.getValue()));
                            } else {
                                indexs.add("%" + (nowReg - 1));
                            }
                            indexs.add(String.valueOf(0));
                            String integerPointer = "%" + getReg();
                            this.nowBasicBlock.inputInstr(new
GetelementptrIR(integerPointer,arrType,storeReg,indexs));
                        } else if (lVal.getBracket() == 0) {
                            ;//只要load就行
```

```
                        }
                    }
                } else {
                    if (var.getBracket() == 2) { //原本是二维数组(不是形参指针)
                        arrType = getType(var.getN1(), var.getN2());
                        if (lVal.getBracket() == 0) { //a
                            indexs = new ArrayList<>();
                            indexs.add(String.valueOf(0));
                            indexs.add(String.valueOf(0));
                            storeReg = "%" + getReg();
                            this.nowBasicBlock.inputInstr(new
GetelementptrIR(storeReg,arrType,arrPointer,indexs));
                        } else if (lVal.getBracket() == 1) { //a[1]
                            indexs = new ArrayList<>();
                            ComputeStmt n2 = lVal.getN2();
                            handleComputerStmt(n2,fatherTable);
                            AddExpStmt addExpStmt = n2.getAddExpStmt();
                            if (addExpStmt.isNum()) {
                                indexs.add(String.valueOf(0));
                                indexs.add(String.valueOf(addExpStmt.getValue()));
                            } else {
                                indexs.add(String.valueOf(0));
                                indexs.add("%" + (nowReg - 1));
                            }
                            storeReg = "%" + getReg();
                            this.nowBasicBlock.inputInstr(new
GetelementptrIR(storeReg,arrType,arrPointer,indexs));
                            arrPointer = storeReg;
                            storeReg = "%" + getReg();
                            arrType = getType(0, var.getN2());
                            indexs = new ArrayList<>();
                            indexs.add(String.valueOf(0));
                            indexs.add(String.valueOf(0));
                            this.nowBasicBlock.inputInstr(new
GetelementptrIR(storeReg,arrType,arrPointer,indexs));
                        } else if (lVal.getBracket() == 2) { //a[1][1]
                            indexs = new ArrayList<>();
                            ComputeStmt n1 = lVal.getN1();
                            handleComputerStmt(n1,fatherTable);
                            AddExpStmt addExpStmt1 = n1.getAddExpStmt();
                            if (addExpStmt1.isNum()) {
                                indexs.add(String.valueOf(0));
                                indexs.add(String.valueOf(addExpStmt1.getValue()));
                            } else {
                                indexs.add(String.valueOf(0));
                                indexs.add("%" + (nowReg - 1));
                            }
                            storeReg = "%" + getReg();
                            this.nowBasicBlock.inputInstr(new
GetelementptrIR(storeReg,arrType,arrPointer,indexs));
                            arrPointer = storeReg;
```

```
                            arrType = getType(0, var.getN2());
                            indexs = new ArrayList<>();
                            ComputeStmt n2 = lVal.getN2();
                            handleComputerStmt(n2,fatherTable);
                            AddExpStmt addExpStmt2 = n2.getAddExpStmt();
                            if (addExpStmt2.isNum()) {
                                indexs.add(String.valueOf(0));
                                indexs.add(String.valueOf(addExpStmt2.getValue()));
                            } else {
                                indexs.add(String.valueOf(0));
                                indexs.add("%" + (nowReg - 1));
                            }
                            storeReg = "%" + getReg();
                            this.nowBasicBlock.inputInstr(new
GetelementptrIR(storeReg,arrType,arrPointer,indexs));
                            String integerPointer = "%" + (nowReg-1);
                            String loadStore = "%" + getReg();
                            this.nowBasicBlock.inputInstr(new LoadIR(loadStore,
integerPointer, "i32"));
                        }
                    } else { //原本是一维数组
                        arrType = getType(0, var.getN2());
                        if (lVal.getBracket() == 0) {
                            indexs = new ArrayList<>();
                            indexs.add(String.valueOf(0));
                            indexs.add(String.valueOf(0));
                            storeReg = "%" + getReg();
                            this.nowBasicBlock.inputInstr(new
GetelementptrIR(storeReg, arrType, arrPointer, indexs));
                        } else {
                            ComputeStmt n2 = lVal.getN2();
                            handleComputerStmt(n2, fatherTable);
                            AddExpStmt addExpStmt = n2.getAddExpStmt();
                            indexs = new ArrayList<>();
                            if (addExpStmt.isNum()) {
                                indexs.add(String.valueOf(0));
                                indexs.add(String.valueOf(addExpStmt.getValue()));
                            } else {
                                indexs.add(String.valueOf(0));
                                indexs.add("%" + (nowReg - 1));
                            }
                            storeReg = "%" + getReg();
                            this.nowBasicBlock.inputInstr(new
GetelementptrIR(storeReg, arrType, arrPointer, indexs));
                            String integerPointer = "%" + (nowReg - 1);
                            String loadStore = "%" + getReg();
                            this.nowBasicBlock.inputInstr(new LoadIR(loadStore,
integerPointer, "i32"));
                        }
                    }
                }
```

```
//                String pointer = "%" + (nowReg - 1);
//                this.nowBasicBlock.inputInstr(new LoadIR("%" + getReg(),pointer));
            }
        } else if (type == 3) {
            ;
        } else if (type == 4) {
            CallExpr callExpr = unaryExpStmt.getCallExpr();
            handleCallExpr(callExpr,fatherTable);
        } else if (type == 5) {
            handleUnaryExpStmt(unaryExpStmt.getUnaryExpStmt(),fatherTable);
            String reg1;
            String reg2;
            String reg3;
            UnaryExpStmt subUnaryExpStmt = unaryExpStmt.getUnaryExpStmt();
            if (subUnaryExpStmt.getType() == 3) {
                if (unaryExpStmt.getOp().getLexType() == LexType.PLUS) {
                    reg1 = "0";
                    reg2 = String.valueOf(subUnaryExpStmt.getNumber());
                    reg3 = "%" + getReg();
                    this.nowBasicBlock.inputInstr(new AddIR(reg1,reg2,reg3));
                } else if (unaryExpStmt.getOp().getLexType() == LexType.MINU) {
                    reg1 = "0";
                    reg2 = String.valueOf(subUnaryExpStmt.getNumber());
                    reg3 = "%" + getReg();
                    this.nowBasicBlock.inputInstr(new SubIR(reg1,reg2,reg3));
                } else if (unaryExpStmt.getOp().getLexType() == LexType.NOT) {
                    String num = String.valueOf(subUnaryExpStmt.getNumber());
                    String compareReg = "%" + getReg();
                    this.nowBasicBlock.inputInstr(new
IcmpIR(compareReg,"eq",num,"0"));
                    this.nowBasicBlock.inputInstr(new ZextIR("%" +
getReg(),compareReg));
                }
            } else {
                if (unaryExpStmt.getOp().getLexType() == LexType.PLUS) {
                    reg1 = "0";
                    reg2 = "%" + (nowReg-1);
                    reg3 = "%" + getReg();
                    this.nowBasicBlock.inputInstr(new AddIR(reg1,reg2,reg3));
                } else if (unaryExpStmt.getOp().getLexType() == LexType.MINU) {
                    reg1 = "0";
                    reg2 = "%" + (nowReg-1);
                    reg3 = "%" + getReg();
                    this.nowBasicBlock.inputInstr(new SubIR(reg1,reg2,reg3));
                } else if (unaryExpStmt.getOp().getLexType() == LexType.NOT) {
                    String beforeAns = "%" + (nowReg-1);
                    String compareReg = "%" + getReg();
                    this.nowBasicBlock.inputInstr(new
IcmpIR(compareReg,"eq",beforeAns,"0"));
                    this.nowBasicBlock.inputInstr(new ZextIR("%" +
getReg(),compareReg));
```

```
                }
            }
        }
        return ;
    }
```

- 标签系统。标签系统和基本块息息相关，一个基本块的进入是标签，出口是跳转语句。跳转语句可以是br，也可以是return。注意br还可能是continue和break，这样后面的代码有一部分都要省略。标签的回填系统也是很大的难点，我写了个JumpUnit协助回填系统。

以if为例：

```
public void handleIfStmt(IfStmt ifStmt, BlockSymbolTable fatherTable, int
numberOfBlockSymbolTable) {
        CondStmt condStmt = ifStmt.getCondStmt();
        Stmt thenStmt = ifStmt.getThenStmt();
        Stmt elseStmt = ifStmt.getElseStmt();

        JumpUnit ifUnit = new JumpUnit();

        String ifCross = handleBlockEnd();
        handleBlockBegin(ifCross);
        handleCondStmt(condStmt,fatherTable,ifUnit);
        //处理if块语句;
        handleStmt(thenStmt,fatherTable,numberOfBlockSymbolTable);
        //if块收尾
        String endReg = "%" + getReg();
        //回填之前失败的跳转
        ifUnit.setLorExpFailCrossBrLine(endReg);
        if (this.nowBasicBlock.getEndOfBlock() == null) { //没有continue和break、return
            BrIR end = new BrIR("if's end br");//if执行结束的无条件跳转
            this.nowBasicBlock.inputInstr(end);
            this.nowBasicBlock.setEndOfBlock(end,"blockBr");
            ifUnit.setIfBr(end);//如果有continue和break、return,这个地方就是空的不需要回
填
        }

        //
        int newBlockNo = this.nowBasicBlock.getBlockNo() + 1;
        BasicBlock basicBlock = new BasicBlock(endReg,newBlockNo);
        this.blocks.add(this.nowBasicBlock);
        this.nowBasicBlock = basicBlock;
        LabelLineIR beginLabel = new LabelLineIR(endReg);
        this.nowBasicBlock.inputInstr(beginLabel);

        if (elseStmt == null) {
            if (ifUnit.getIfBr() != null) { //ifUnit.getIfBr是空，说明因为由continue和
break导致没填
                ifUnit.getIfBr().setUnconditionalJump_reg(endReg);//回填之前if结束的跳
转
            }
        } else if (elseStmt != null) {
//            if (thenStmt instanceof CompoundStmt || thenStmt instanceof IfStmt ||
thenStmt instanceof ForStmt) { //if块是 块
//                handleStmt(elseStmt,fatherTable,numberOfBlockSymbolTable+1);
//            }
//            else { // if块只是语句
//                handleStmt(elseStmt, fatherTable, numberOfBlockSymbolTable);
//            }
            //else结束
            int numOfIfBlocks = checkHasBlock(thenStmt);
            handleStmt(elseStmt, fatherTable, numberOfBlockSymbolTable +
```

```
numOfIfBlocks);
            String elseEnd = "%" + getReg();
            if (ifUnit.getIfBr() != null) {
                ifUnit.getIfBr().setUnconditionalJump_reg(elseEnd);//回填之前if结束的跳
转
            }
            if (this.nowBasicBlock.getEndOfBlock() == null) {
                BrIR end = new BrIR(elseEnd);
                this.nowBasicBlock.inputInstr(end);
                this.nowBasicBlock.setEndOfBlock(end,"blockBr");
            }
            blocks.add(this.nowBasicBlock);
            //新块
            this.nowBasicBlock = new
BasicBlock(elseEnd,this.nowBasicBlock.getBlockNo() + 1);
            this.nowBasicBlock.inputInstr(new LabelLineIR(elseEnd));
        }
    }
```

- 另外一个难点是数字的直接调用，在unaryExp就要识别出纯数字类型，纯数字不仅包括了k这样，还有(k)((k))(((k)))……这样的。