编译器设计文档

> 班级：202111
>
> 学号：20373420
>
> 姓名：张铭轩

一、参考编译器介绍

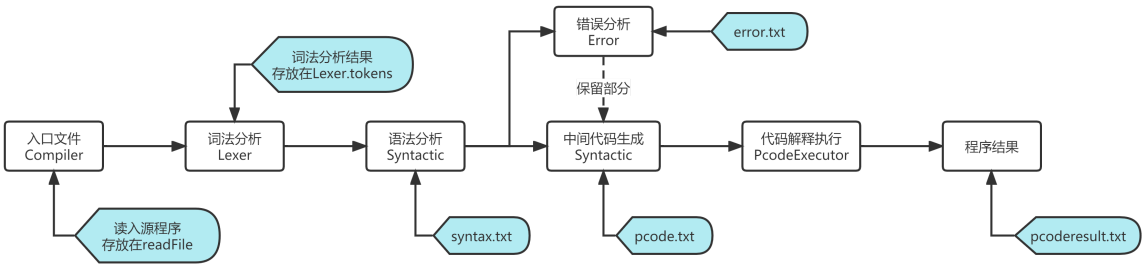
在语法分析及之前的部分参考了pl0-compiler官方文档。其编译过程采用一趟扫描方式，以语法、语义分析程序为核心，词法分析程序和代码生成程序都作为一个过程，当语法分析需要读单词时就调用词法分析程序，而当语法、语义分析正确，需要生成相应的目标代码时，则调用代码生成程序。表格管理程序实现变量，常量和过程标识符的信息的登录与查找。出错处理程序，对词法和语法、语义分析遇到的错误给出在源程序中出错的位置和与错误 性质有关的编号，并进行错误恢复。，

其中比较重要的参考部分有，`getsym`等函数等词法分析部分，和语法分析结构。

二、编译器总体设计

1.总体结构

我的编译器因为在中间代码生成的时候进行了重构，包括符号表等结构，进行了从新的构造，所以编译器的整体结构，以及每一步的生成结果保存位置如下图。



2.接口设计

整体的运行结构分为三个功能包：语法分析包`lexical`，语法分析包`syntax`和代码生成包`Generator`。其中语法分析包`lexical`的入口文件是`Lexer`，调用方法`lexer`进行词法分析；语法分析包`syntax`的入口文件是`Syntactic`，调用方法`syntax`进行词法分析和中间代码生成（耦合在同一文件中）；代码生成包`Generator`的入口文件是`PcodeGenerator`，调用方法`run`解析中间代码，进行代码运行。

主文件是`Compiler`，入口函数是主函数，具体的整体接口调用顺序如下图：

```java
public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader( fileName: "testfile.txt"));
    String str = "";
    while((str = br.readLine()) != null) {
        readFile.add(str);
        maxLine++;
    }
    br.close();

    Lexer lxr = new Lexer(maxLine, readFile);
    lxr.lexer();
      lxr.printLexer();
    System.out.println("-----------finish lexer-----------");

    Syntactic syn = new Syntactic();
    syn.syntax();
      syn.printSyntax();
      syn.printPcode();
    System.out.println("-----------finish Syntax-----------");

    Scanner scanner = new Scanner(System.in);
    PcodeExecutor pe = new PcodeExecutor(Syntactic.getCodes(), scanner);
      System.out.println("-------before run-------");
    pe.run();
    System.out.println("-------finish run-------");
    pe.print();
}
```

其中分别生成一个词法分析器对象、一个语法分析器和一个代码执行器的对象，并调用对象的相关方法进行编译。其中含有`print-`的函数是测试打印函数，可以分别输出每一个步骤的不同分析结果。

3.文件组织

上文已经大致讲过文件的结构，由于错误处理和后面的设计没有结合在一起，此处展示完整编译器（左）和错误处理（右）的文件结构。

1. 整体编译器文件结构
    - **def包**：生成中间代码构造符号表的时候，定义函数（Function）、变量（Symbol）以及层次符号表（Symbols）使用的类，
    - **Generator包**：在执行中间代码的时候，需要用的所有类。`Func`进行函数相关操作；`LabelGenerator`跳转标签类；`Pcode`是中间代码类；`PcodeExecutor`执行生成后的pcode中间代码；`PcodeType`是一个枚举类，枚举所有的pcode类型；`RetrInfo`是函数返回类；Var进行变量相关操作.
    - **lexical包**：`Lexer`是此法分析器，`Sym`记录所有的词法类型，`Token`是单词类，为每一个单词生成一个对象。
    - **syntax包**：`Syntactic`是语法分析器类，`Component`是语法词汇类（最后没有用到）。
    - **Compiler文件**：根据设计规范，作为编译器运行的入口。
2. 错误处理文件结构
    - **lexical包**：同上。
    - **syntax包**：同上。
    - **Compiler文件**：同上。
    - **Error包**：涵盖全部的错误分析部分，包括符号表。`Ident`符号类，`dentList`是符号表类，`ErrorType`是错误列表里存放的对象，`ErrorHandle`是部分错误处理方法。

三、词法分析

1.编码前

1. 任务分析

   在这一部分，需要逐字符读入文件，然后判断字符组成的单词及其类别，输出单词的类别和单词值。

2. 程序设计思路

   首先需要读程序文件，然后遍历整个文件的每一个字母，根据文法规则进行判断。遇到空格的时候查看当前获得的单词值，与词法分析类别码进行对比判断，将结果存起来，在程序分析完后进行输出。

2.编码后

1. 程序结构解释

   - 读入文件：在`Compiler`中直接讲文件内容读入到`static ArrayList<String> readFile = new ArrayList<>();`中，这是一个`String`型`ArrayList`容器，其序号可以转换成为行号。

     ```
     BufferedReader br = new BufferedReader(new FileReader("testfile.txt"));
     String str = "";
     while((str = br.readLine()) != null) {
         readFile.add(str);
         maxLine++;
     }
     br.close();
     ```

   - 创建词法分析器对象

     在`Compiler.java`中创建，其中`Lexer`是词法分析器类，`lxr`是`Lexer`的实例化对象，`lexer`方法是词法分析的入口函数。

     ```
     Lexer lxr = new Lexer(maxLine, readFile);
     lxr.lexer();
     lxr.printLexer();
     ```

   - lexical包：包含词法分析器`Lexer`类，`Sym`类别码类，`Token`单词类，是词法分析功能实现的主要部分。

     - `Lexer`类：实现一个词法分析器，有其对应的属性和方法。

       ```
       public class Lexer {
           public static boolean fileEnd = false;//文件是否分析完毕
           public static boolean formatString = false;//是否处于输出的内容中

           // 类的属性
           private String curLine;     //当前正在分析行的内容
           private char curChar;       //当前正在进行判断的字符
           private int lineNum = 0;    //当前行数
           private int totalLine = 0;  //总行数
           private int charPos = 0;    //curChar在当前行的位置。
           private int length = 0;     //当前行的长度

           //类的方法
           public void printLexer() throws IOException{}//
           public void lexer(){}       //词法分析入口函数
       ```

```
    public void get_sym(){}      //获取一个字符并进行分析
    public void next_char(){}    //获取下一个字符
    public void roll_back(){}    //回滚，curChar变为前一个字符

    //辅助判断函数
    boolean isSpace(char c){}    //curChar是否是空格
    boolean isLetter(char c){}   //curChar是否是字母
    boolean isDigit(char c){}    //curChar是否是数字
    Sym.sym is_reserve(String str){}    //当前获取单词sec是否是保留单词
}
```

- `Sym`类别码类：主体包含两部分，分别是`sym`枚举类，枚举所有的单词类别名称；还有`static Map<String, sym> reserve = new HashMap<>();`用于保存单词拼写和保留单词类别名称的对应关系。

```
public enum sym {
    IDENFR, INTCON, STRCON, MAINTK, CONSTTK,
    INTTK, BREAKTK, CONTINUETK, IFTK, ELSETK,
    NOT, AND, OR, WHILETK, GETINTTK, PRINTFTK,
    RETURNTK, PLUS, MINU, VOIDTK, MULT, DIV,
    MOD, LSS, LEQ, GRE, GEQ, EQL, NEQ, ASSIGN,
    SEMICN, COMMA, LPARENT, RPARENT, LBRACK,
    RBRACK, LBRACE, RBRACE, NONETYPE, SYMEND;

    public String toString() {
        return this.name();
    }
}
```

- `Token`单词类：包含的属性有单词的类别码，单词的内容，和单词所在的行数。答案以Token的实例化对象的形式，保存在`public static Vector<Token> tokens = new Vector<>();`以便于最后的集体输出。

```
public class Token {
    private Sym.sym sym;
    private String content;
    private int lineNum;
}
```

2. 关键设计讲解
    - **仅含1个字符的单词**：对于字符本身即可成为单词的符号，常见于运算符（比如加减符号）和括号，在读到对应的字符时直接加入词法分析结果集即可

```
if (curChar == '+') {
    next_char();
    tokens.add(new Token(Sym.sym.PLUS, "+", lineNum));
} else if (curChar == '-') {
    next_char();
    tokens.add(new Token(Sym.sym.MINU, "-", lineNum));
}else if (curChar == '{') {
    next_char();
    tokens.add(new Token(Sym.sym.LBRACE, "{", lineNum));
} else if (curChar == '}') {
    next_char();
    tokens.add(new Token(Sym.sym.RBRACE, "}", lineNum));
}
```

- **包含2个字符的单词**：常见于两个字符组成的单词，其中第一个字符也能单独形成单词的情况。处理这种情况，采取多向前读一位来判断具体的单词类型，并将正确的类型添加到`tokens`中

```
if (curChar == '=') {
    next_char();
    if (curChar == '=') {
        next_char();
        tokens.add(new Token(Sym.sym.EQL, "==", lineNum));
    } else {
        tokens.add(new Token(Sym.sym.ASSIGN, "=", lineNum));
    }
}
```

- **无固定长度的特殊类型单词**：包括数字，变量名及标准输出字符串等。处理方式为，在该种情况中继续向前读入，直到可以判断该符号串结束（如数字类型读至非数字结束，标准输出字符串读至后一个”时）

```
if (curChar == '\"') {
    sec += curChar;
    next_char();
    while (curChar != '\"') {
        sec += curChar;
        next_char();
    }
    sec += curChar;
    next_char();
    tokens.add(new Token(Sym.sym.STRCON, sec, lineNum));
}
```

- **注释处理**：注释分为两种，一种是单行注释，一种是多行注释。（其中跟除法的判断有耦合）

  单行注释`//`：处理方式比较简单，利用词法分析器具有的属性，使得`charPos = length;`，这样在使用`next_char`获取下一个字母时，会因为已经读到当前行的最后一个字母而进入下一行。

多行注释 `/**/`：判断到`/*`时继续向后读取，直至获取到`*/`结束。其中因为读取下一个字符的模式设计，需要防止`/***/`情况的出现，在读取到一个`*`而下一个字符并非`/`时，需要回滚一个字符重新进行注释结束判断。

```
if (curChar == '/') {
    next_char();
    if (curChar == '/') {
        charPos = length;                                    // 单行注释的
结束
        next_char();
    } else if (curChar == '*') {
        while (!fileEnd) {
            next_char();
            if(curChar == '*') {
                next_char();
                if(curChar == '/') {
                    next_char();
                    break;                                   // 多行注释的
结束
                }
                roll_back();// in case /***/
            }
        }
    } else {
        tokens.add(new Token(Sym.sym.DIV, "/", lineNum));    // 除法的判断
}
```

四、语法分析

1.编码前

1. 任务分析

   在这一部分，需要上一部分获取到的单词，根据文法规则，判断出其组成的语法成分。

2. 程序设计思路

   因为没理解语法树的结构，所以采取了递归下降的办法分析语法。递归下降的主要思路，其实就是深度遍历一个语法树，当识别到特定的词法符号的时候，跳转进入对应的语法分析函数，然后进行对应的语法部分分析。当这个语法部分识别到最后一个符号的时候，回到调用此函数的部分，此时已经分析过的部分就不在分析，整个过程是一个持续向后分析的过程。

3. 语法预处理

   ```
   符号[...]表示方括号内包含的为可选项
   符号{...}表示花括号内包含的为可重复 0 次或多次的项
   ```

   这两个是语法分析的主要前序处理可选项和重复多次的问题，这两种基本上都用括号内的第一个单词进行判断，如果存在括号内第一个单词，那么可选项至少出现一次，然后循环处理内部即可。

   这种处理方式也可以**解决左递归问题**，以`AddExp`为例

   ```
   加减表达式 AddExp → MulExp | AddExp ('+' | '−') MulExp // 1.MulExp 2.+ 需覆盖
   3.-需覆盖
   ```

其中就可以化简成以下模式，然后就可以通过循环来处理了。

```
加减表达式 AddExp → MulExp { ('+' | '−') MulExp }
```

其余`RelExp`，`MulExp`等类似结构文法的处理，参照`AddExp`的模式即可。

4. 整体结构的设计

```
编译单元 CompUnit → {Decl} {FuncDef} MainFuncDef // 1.是否存在Decl 2.是否存在FuncDef
```

这一句话就是整体的编译单元架构，然后根据第一个单词的类型，在变量定义、方程定义和主函数定义中判断，并跳转进对应的编译模块。

```
while(curToken.getSym() == Sym.sym.INTTK ||
        curToken.getSym() == Sym.sym.CONSTTK ||
        curToken.getSym() == Sym.sym.VOIDTK){
    if(curToken.getSym() == Sym.sym.VOIDTK) {
        FuncDef();
    } else if (curToken.getSym() == Sym.sym.CONSTTK) {
        Decl();
    } else if (curToken.getSym() == Sym.sym.INTTK) {
        next_token();
        if(curToken.getSym() == Sym.sym.IDENFR) {
            next_token();
            if(curToken.getSym() == Sym.sym.LPARENT) {
                prev_token();
                prev_token();
                FuncDef();
            } else {
                prev_token();
                prev_token();
                Decl();
            }
        }
        else if(curToken.getSym() == Sym.sym.MAINTK) {
            prev_token();
            MainFuncDef();
//                    mainEnd = true;
        }
        else{
            // error handle
        }
    }
}
```

2.编码后

1. 整体结构的处理

根据文法的规定，可以看到整个程序的定义必须按照变量定义、函数定义和主函数定义来进行，这样原来的反复跳转的主体结构就可以按照规定好的顺序进行化简。化简后，按照文法规定的顺序，一次处理变量定义、函数定义和主函数定义，步骤如下：

```
//        Decl
```

```
while (curToken.getSym() == Sym.sym.INTTK || curToken.getSym() ==
Sym.sym.CONSTTK) {
    if (curToken.getSym() == Sym.sym.CONSTTK) {
        ConstDecl();
    } else {
        next_token();
        if (curToken.getSym() == Sym.sym.IDENFR) {
            next_token();
            if (curToken.getSym() == Sym.sym.LPARENT) {
                prev_token();
                prev_token();
//                          FuncDef(); go to next cycle
                break;
            } else {
                prev_token();
                prev_token();
                VarDecl();
            }
        } else {
            prev_token();
            break;
        }
    }
}

//        FuncDef
while (curToken.getSym() == Sym.sym.INTTK || curToken.getSym() ==
Sym.sym.VOIDTK) {
    if (curToken.getSym() == Sym.sym.INTTK) {
        next_token();
        if (curToken.getSym() == Sym.sym.IDENFR) {
            prev_token();
            FuncDef();
            continue;
        } else {
            prev_token();
            break; // define main function
        }
    } else {
        FuncDef();
        continue;
    }
}

MainFuncDef();
```

2. 递归下降实现步骤

   入口函数，语法分析器通过next_token函数获取第一个token，然后开始从编译单元进入进行语法分析。

```
public void syntax() {
    next_token();
    CompUnit();
}
```

功能性函数：在语法分析的过程中，调用词法获取token的部分。

```
//获取下一个token
public void next_token();

//倒退一个token，并清除语法分析容器中添加的部分，回到对应token进入语法分析前的状态
public void prev_token();

//把当前获取的token放入语法分析结果的容器中，在next_token中调用
void pushToken();

//把语法成分按照规定的模式放入语法分析结果的容器中
void pushSyn(String str);

//表示是某个保留字符的token，用于判断当前的token是不是某种特定字符
boolean is_token(Sym.sym token);
```

递归下降部分函数，给每一个语法成分建立一个分析方法，然后按照文法规定进行调用，以下是递归下降的主体各个方法：

```
public void CompUnit()
public void Decl()
private void ConstDecl()
private void ConstDef()
private void ConstInitVal()
private void VarDecl()
private void VarDef()
private void InitVal()
private void Exp()
private void ConstExp()
public void FuncDef()
public void FuncFParams()
private void FuncFParam()
public void FuncRParams()
public void MainFuncDef()
public void Block()
private void BlockItem()
private void Stmt()
private void Cond()
private void LOrExp()
private void LAndExp()
private void EqExp()
private void RelExp()
private void AddExp()
private void MulExp()
private void UnaryExp()
private void PrimaryExp()
private void LVal()
```

左递归的具体编码实现，以`AddExp`为例：

```
private void AddExp() {
    MulExp();// 分析第一个MulExp
    pushSyn("AddExp");

    while (curToken.getSym() == Sym.sym.PLUS ||
```

```
            curToken.getSym() == Sym.sym.MINU) {
        // 如果存在加号或者减号，代表每个符号后面还有一个后续MulExp，进入循环分析即可
        // 对应AddExp → AddExp ('+' | '-') MulExp
        next_token();
        MulExp();
        pushSyn("AddExp");
    }
    // 不存在加减号，则直接跳过可选重复部分，结束AddExp分析，对应AddExp → MulExp
}
```

3. 回溯问题的处理

Stmt语句的处理，在递归下降的部分，因为无法判断以 `IDENFR` 开头的文法有三种情况，分别是 `LVal '=' Exp ';' | Exp | LVal '=' 'getint''('')'';'` 首先需要先通过等于号，把 `Exp` 区分出来，这里需要涉及一个回溯，因为一开始是用 `LVal` 的模块往下分析的，所以分析出来的结果不是 `Exp` 会有的结构。回退的时候，`prev_token` 会把当前token到上一个token（包含在其中）的所有部分remove掉，然后重新按照 `Exp` 进行分析。

这个位置prev_token的回退清理逻辑很重要，要么会有进入错误分析模块产生的多余语法成分被输出。

```
if(curToken.getSym() == Symbol.sym.IDENFR) {
    next_token();
    int ansN = ans.size();
    int tokenN = tokenPos;
    if(curToken.getSym() != Symbol.sym.LPARENT) {
        //Unfinish
        prev_token();
        LVal();
        // 识别不到等于号，则不是LVal开头，回退回退！！
        if(curToken.getSym() != Symbol.sym.ASSIGN) {
            int tmpN = tokenPos;
            // 看往前多读了多少个token然后全部清除掉
            for(int i = 0; i < (tmpN - tokenN)+1; i++) {
                prev_token();
            }
            Exp();
            is_token(Symbol.sym.SEMICN);
        }
        else {
            // LVal '=' Exp ';'
            // LVal '=' 'getint''('')'';'
            next_token();
            if(curToken.getSym() == Symbol.sym.GETINTTK) {
                next_token();
                is_token(Symbol.sym.LPARENT);
                is_token(Symbol.sym.RPARENT);
            }
            else {
                Exp();
            }
            is_token(Symbol.sym.SEMICN);
        }
    }
    else {
        prev_token();
```

```
            Exp();
            is_token(Symbol.sym.SEMICN);
        }
    }
```

五、错误处理

1.编码前

1. 错误类型分析

   错误一共有13类，其中a类可以单独处理，在formatString中单独加入一个字符判断函数即可。b、c类有关于函数或者变量名，d、e类有关于函数的参数，f、g关于函数返回的问题，h是跟变量类型有关，i至k类是关于缺乏配对符号的问题，l也是有关于输出的问题，m是关于循环的问题。

   所以可以把错误分成大致四类：

   - a+l：可以在formatString中一起进行处理
   - b~e+h：这四类都使用符号表进行处理
   - f+g+m：使用全局变量标记代码块的层次，然后通过相关的数值判断是否有相关问题
   - i~k：非常简单的符号缺失问题，可以和`is_token`函数结合

2. 结构设计

   把所有的东西都放进了Error包中，然后语法分析的同时做错误分析，相关的错误处理方法、符号表类调用Error包中的内容即可。

2.编码后

1. 有关于`formatString`错误处理【a+l】

   比较好从原来的语法分析中拆分出来，所以放在了`ErrorHandle`类中进行处理，判断有无非法符号，出错就添加错误，同时结束`formatString`的错误分析（一行只报一次错误）。遍历`formatString`同时，计算格式字符串（%d）的数量。然后在语法分析原程序中，计算表达式的个数，进行匹配，完成两类错误的处理。

2. 符号缺失问题【i-k类】

   直接写进`is_token`函数，用if-else结构拆分不同的报错类型，很容易处理。

3. 有关符号表的处理

   符号对象的内容，这是符号所记录的所有属性，但是由于函数、变量混记，所以对于有些变量声明的时候用不到某些属性，所以构造方法做了一个多态，对应不同符号声明的需求。

```
public class Iden {
    String str;                                 // 符号名
    int layerID;                                // 所在的代码块层次
    int dim = 0;                                // 如果是数组函数，记录数组维度
    idenType type = idenType.NULL;              // 变量类型——应下方枚举类
    BType btype = BType.unknown;                // 字节变量类型——应下方枚举类
    ArrayList<param> params = new ArrayList<>(); //如果有传参，记录参数的数组——应下方参数类

    public enum idenType {
        var, constVar, func, returnParam, NULL
    };//
    public enum BType {
        intType, voidType, unknown
```

```
    };

    public static class param {
        String name; // 参数名称（其实没什么用）
        int dim = 0; // 参数只有int型，所以直接记录维数了
    }
}
```

符号表的相关属性和方法如下：

P.S.没有做分层多维符号表是因为符号本身记录了自己所在的层数，删除符号表的时候，参照符号属性中所在的代码块层次进行相关操作的判断就可以

```
public class IdenList {
    public static ArrayList<Iden> idenList = new ArrayList<>();

    // 添加符号的多态
    public static void addIden(Token t, int layer, int d,
                               Iden.idenType it, Iden.BType bt)
    public static void addIden(Token t, int layer,
                               Iden.idenType it, Iden.BType rt,
                               ArrayList<Iden.param> params)
    // 检查是否有重复定义--处理b类错误
    public static boolean checkIden(Iden iden, int layer, int lineNum)
    // 检查是否有所用标识符定义--处理c类错误
    public static boolean checkIdenExist(Iden iden, int lineNum,
Iden.idenType it)
    // 检查是否是常数--处理h类错误
    public static boolean checkConst(String name, int lineNum)
    // 代码块结束时，弹出该块中所有的符号
    public static void popIden(int layer)
}
```

d类问题使用计数器，在RParam函数中做计算和虚参表长度做对比就可以。

e类问题使用函数`checkParam`将当前获得的参数类型和该函数符号生命中的虚参列表进行对比

```
boolean checkParam(Token token, Iden.param param){
        if(token.getSym() == Symbol.sym.INTCON ){
            if(param.getDim() != 0 ) {
                add_error(token.getLineNum(), "e");
                return false;
            }
            return true;
        }// 整型数的判断

        int size = IdenList.idenList.size();
        for(int i = 0; i < size; i++) {
            Iden tmp = IdenList.idenList.get(i);
            if(tmp.getStr().equals(token.getStr())) {   // 在符号表里找到充当实
参的IDENFR的声明
                if(tmp.getBtype() != Iden.BType.intType ||
                        tmp.getDim()-curDim != param.getDim()) {// 用实参的声
明维数和调用维数计算，匹配虚参维数
                    add_error(token.getLineNum(), "e");
                    return false;
                }else {
```

```
                    return true;
                }
            }
        }
        return false;
    }
}
```

4. 返回值问题【f、g类】

   编码前的设计思路，可能会在传参的时候出现问题，现在只需要匹配返回值所在的层次，并判断是否需要返回值、是否检测到返回值即可。返回值的判断设置在每一个Block结束之后的位置，特别注意，return的报错要在右大括号的行数，这里需要单独记录行数。

5. 循环问题【m类】

   编码前的设计思路问题在于，处理思路是设置全局变量，看当前的循环层数是不是非0，非0表示该语句一定在某一个循环中，此时break和continue的出现就一定是合理的。

六、代码生成

1.编码前

1. 尝试生成了一棵语法树，然后对语法树进行遍历，通过深度遍历进行代码的生成。
2. 符号表

   继承了错误处理的符号表，进行编写中间代码生成。

3. 编码的设计

   按照栈式结构进行设计，放置在语法分析里，一边进行语法分析一边生成中间代码。中间代码设置了比较多的内容，计划根据不同的三元式类型对不同的属性部分进行赋值，具体内容如下

```
public class Pcode {
    private CodeType type;
    private int value1 = 0;
    private int value2 = 0;
    private String label1;
    private String label2;
}
```

2.编码后

AST确实建立起来了，然后尝试做了一下MIPS，做崩溃了，然后直接转Pcode。AST放弃使用了，感觉跟递归下降在遍历层面也没有什么区别，还让代码结构更加的复杂了，既然选择了Pcode就直接用最简单的办法进行分析就行了。

同时，因为上面错误处理也说了，他的`Iden`设计的不是很好，函数和变量都混杂在一起

Pcode定义

直接采用最简单的三元式结构，操作符+操作数1+操作数2，然后运算结果直接压栈（也是不需要四元式，跟mips的区别）

```
public class Pcode {
    private CodeType type;
    private Object value1 = null;
    private Object value2 = null;
}
```

PCode代码有三个属性：第一个是PCode的类型，是一个枚举类，具体内容查看`PcodeTYpe`类。`value1`和`value2`可以是整型数Integer或者字符串String或者为空null，两者的具体内容取决于PCode的类型。

Pcode生成

1. 对于普通的运算，在识别两个操作数之后，对操作符进行压栈，以AddExp为例

```
private int AddExp() {
    int dim = 0;
    dim = MulExp();
    pushSyn("AddExp");
    while (curToken.getSym() == Sym.sym.PLUS ||
            curToken.getSym() == Sym.sym.MINU) {
        Sym.sym op = curToken.getSym();
        next_token();
        MulExp();

        switch (op) {
            case PLUS:
                codes.add(new Pcode(PcodeType.ADD));
                break;
            case MINU:
                codes.add(new Pcode(PcodeType.SUB));
                break;
        }

        pushSyn("AddExp");
    }
    return dim;
}
```

2. if while等使用跳转标签的结构

   对于需要使用标签的结构，进行labelGenerator，放置在label列表中，在适当的位置进行跳转，以if为例

```
if (curToken.getSym() == Sym.sym.IFTK) {
// 先生成一个临时的label列表，包含索引（String）和具体的label内容（String）
    HashMap<String, String> tmp = new HashMap<>();
    tmp.put("if", labelGenerator.getLabel("if"));
    tmp.put("else", labelGenerator.getLabel("else"));
    tmp.put("if_end", labelGenerator.getLabel("if_end"));
    tmp.put("if_block", labelGenerator.getLabel("if_block"));
// 在ifLabel的列表中添加新建的HashMap，作为本层选择的标签区域
    ifLabels.add(tmp);
// 开始分析if语句，把刚才创建的标签区域中的唯一一个if标签取出做pcode
    codes.add(new Pcode(PcodeType.LABEL, ifLabels.get(ifLabels.size() -
1).get("if")));
```

```
    next_token();

    is_token(Sym.sym.LPARENT);
    Cond("if");
    is_token(Sym.sym.RPARENT);
// 条件语句判断结束后，如果栈顶为0跳转到else中
    codes.add(new Pcode(PcodeType.JZ, ifLabels.get(ifLabels.size() -
1).get("else")));
// 这里是if语句的代码块开始声明
    codes.add(new Pcode(PcodeType.LABEL, ifLabels.get(ifLabels.size() -
1).get("if_block")));

    Stmt();
// 这里是if语句的代码块结束声明
    codes.add(new Pcode(PcodeType.JMP, ifLabels.get(ifLabels.size() -
1).get("if_end")));

// 这里是else语句的代码块结束声明
    codes.add(new Pcode(PcodeType.LABEL, ifLabels.get(ifLabels.size() -
1).get("else")));


    if (curToken.getSym() == Sym.sym.ELSETK) {
        next_token();
        Stmt();
    }
// 这里是else语句的代码块结束声明
    codes.add(new Pcode(PcodeType.LABEL, ifLabels.get(ifLabels.size() -
1).get("if_end")));
// 执行完毕后弹出该label区域
    ifLabels.remove(ifLabels.size() - 1);
}
```

3. 短路求值

对于OR语句，判断到第一个1就结束判断直接跳转；对于AND语句，判断到第一个0就直接跳转。操作就是在每一个条件表达式后加一个跳转的判断，使得每一个条件表达式计算完毕之后能够直接跳转

以OR为例

```
private void LOrExp(String from) {
// 条件标签的计数生成
    int cntCond = 0;
    String label = labelGenerator.getLabel("cond_" + cntCond);
    cntCond++;

    LAndExp(from, label);
// 放置标签，表示开始第n个条件
    codes.add(new Pcode(PcodeType.LABEL, label));
// 如果当前栈顶是1直接跳转对应模块
    if (from.equals("if")) {
        codes.add(new Pcode(PcodeType.JNZ, ifLabels.get(ifLabels.size() -
1).get("if_block")));
    } else if (from.equals("while")) {
        codes.add(new Pcode(PcodeType.JNZ,
whileLabels.get(whileLabels.size() - 1).get("while_block")));
    }
```

```
        pushSyn("LOrExp");

        while (curToken.getSym() == Sym.sym.OR) {
            ……此处分析类似上文
            pushSyn("LOrExp");
        }

        for (int i = codes.size() - 1; i >= 0; i--) {
            Pcode tmp = codes.get(i);
            if (tmp.getType() == PcodeType.JNZ) {
                codes.remove(i);
                break;
            }
        }//最后一个跳转，因为存在调用Cond模块的模块已经加了跳转，所以删除一个。
    }
```

PCode解释器

首先解释器运行之前，需要对一些特殊内容标记位置，包括主函数的开始位置，每个label在pcode列表中的位置，和每个函数所在位置，这样进行跳转的时候直接查找对应的列表就可以了。这个功能在PcodeExecutor的构造函数中完成。

```
public PcodeExecutor(ArrayList<Pcode> codes, Scanner scanner) {
    this.codes = codes;
    this.scanner = scanner;
    for (int i = 0; i < codes.size(); i++) {
        Pcode code = codes.get(i);
        // get main function address
        if (code.getType().equals(PcodeType.MAIN)) {
            mainAddress = i;
        }
        // get all label
        if (code.getType().equals(PcodeType.LABEL)) {
            labelTable.put((String) code.getValue1(), i);
        }
        //get all function
        if (code.getType().equals(PcodeType.FUNC)) {
            funcTable.put((String) code.getValue1(), new Func(i, (int)
code.getValue2()));
        }
    }
}
```

生成出来的PCode列表一条一条执行，就可以让程序运行起来了。其中Pcode运算大概分为两种，一种是双目运算符的运算，一种是一般指令，接下来会分为两种类型对于PCode的执行进行解释。

运算指令：这个指令比较简单，选择运算栈上，栈顶和次栈顶的两个数字进行运算，次栈顶元素在前，栈顶元素在后。对于普通的加减乘除取余运算，在运算后把结果放置在运算栈顶上。对于比较以及条件运算，成立则置1，不成立则置0。

CodeType	Operation
ADD	+
SUB	-
MUL	*
DIV	/
MOD	%
CMPEQ	==
CMPNE	!=
CMPGT	>
CMPLT	<
CMPGE	>=
CMPLE	<=
AND	&&
OR	||
NOT	!
NEG	-
POS	+

一般指令：这种指令的Value指不一定是数字了，所以不能采取简单的`pop()`和`push()`来直接进行计算，应当通过对不同代码的理解来进行操作。

CodeType	Value1	Value2	Operation
LABEL	标签名称 Label_name	Set a label	
VAR	变量名称 Ident_name	Declare a variety	
PUSH	变量或者数值 Ident_name/Digit	push(value1)	
POP	地址Address	变量名 Ident_name	value2的值赋给value1的地址 *value1 = value2
JZ	跳转标签 Label_name		栈顶为0时跳转
JNZ	跳转标签 Label_name		栈顶不为0时跳转
JMP	跳转标签 Label_name		无条件跳转
MAIN			标志主函数的开始
FUNC			函数声明的开始 Function label
ENDFUNC			函数声明的结束 End of function label
PARA	变量名 Ident_name	Type	函数声明时的参数 Parameters
RET	返回值或者为空 Return value or not		函数返回值 Function return
CALL	函数名Function name		在函数中进行调用
RPARA	类型Type		准备函数跳转时将要携带的参数 Get parameters ready for function call
GETINT			获取一个整型数并放在栈顶 Get a integer and put it into stack top
PRINT	字符串String	格式字符串数量Para num	从栈顶pop内容并输出 Pop values and print.
ARRVAR	变量名 Ident_name	类型Type	为数组设置维度 Set dimension info for array variety
VALUE	变量名 Ident_name	类型Type	获取变量值 Get the variety value
ADDRESS	变量名 Ident_name	类型Type	获取变量地址 Get the variety address

CodeType	Value1	Value2	Operation
PLACEHOLDER			声明数组向运算栈内添加占位符0 Push something to hold places
EXIT			主函数退出 Exit