**markDOWN**

**任务2 编译工具使用**

**1.使用工具**

ANTLR（IDEA中的ANTLR插件，版本为4.13.1）

**2.解决方案的整体介绍**

首先创建PL0.g4语法文件，按照项目要求，写好词法和语法规则，然后进行测试，使用示例程序进行测试，观察输出的语法树是否正确，正确的话，生成.java文件格式的词法、语法分析器等，使用生成的语法词法分析器，重写生成的BaseVisitor中的一些方法来实现中间代码的生成。

**3.具体实现，客户化工作**

按照要求，写好PL0.g4文件

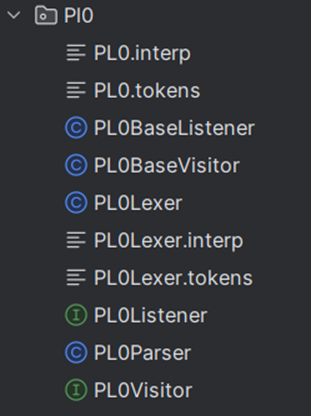


|  |
| --- |
| Java *// PL0.g4*  grammar PL0;  *// 词法规则* PROGRAM: 'PROGRAM'; BEGIN: 'BEGIN'; END: 'END'; CONST: 'CONST'; // ...  *// 语法规则* program: programHeader subProgram;  programHeader: PROGRAM IDENTIFIER ; // ...  *// 忽略空白字符* WS: [ \t\r\n]+ -> skip; |

进行测试，左边为测试程序，右边为输出的语法树



确认无误后，生成语法分析器等文件



可以看到PL0BaseVisitor中的visit方法，其效果均为访问子节点

|  |
| --- |
| Java @SuppressWarnings("CheckReturnValue") public class PL0BaseVisitor<T> extends AbstractParseTreeVisitor<T> implements PL0Visitor<T> {   @Override public T visitProgram(PL0Parser.ProgramContext ctx) { return visitChildren(ctx); }  // visitChildren(ctx)访问子节点 @Override public T visitProgramHeader(PL0Parser.ProgramHeaderContext ctx) { return visitChildren(ctx); }   @Override public T visitSubProgram(PL0Parser.SubProgramContext ctx) { return visitChildren(ctx); }   @Override public T visitConstantDeclaration(PL0Parser.ConstantDeclarationContext ctx) { return visitChildren(ctx); }    // ...   } |

然后是生成中间代码部分的实现。

生成中间代码时，我们需要访问语句部分（包含赋值语句，条件语句，循环语句，复合语句，空语句等），即需要重写这些语句的访问类，其中复合语句的访问类不需要重写，因为其语法规则为

|  |
| --- |
| Java compoundStatement: BEGIN statement (SEMICOLON statement)\* END; |

，则不重写会自动访问其子节点即“语句”部分。

首先创建一个对象来进行存储中间代码，包含以下属性

|  |
| --- |
| Java public class IntermediateCode {  public int start; // 开始地址  public String operate; // 运算符   public String arg1; // 参数1  public String arg2;  public String result; // 结果，地址或者参数 } |

然后是部分访问方法的重写(此处只详细介绍访问循环语句方法的重写，其余的重写思路与其类似，不再介绍)

|  |
| --- |
| Java // 循环语句 @Override public Void visitWhileStatement(PL0Parser.WhileStatementContext ctx) {  String expression0 = ctx.condition().expression(0).getText(); // 获得关系表达式的第一个元素  String operator = ctx.condition().getChild(1).getText(); // 比较符  String expression1 = ctx.condition().expression(1).getText(); // 关系表达式对参数二  labelCounter++;// 输入一条结果前，地址+1  int startLabel = labelCounter; // while的起始地址，之后循环跳回来  codeResult.add(new IntermediateCode(startLabel,"j"+operator, expression0, expression1, String.*valueOf*(startLabel+2)));  labelCounter++; //  int startLabel2 = labelCounter; // while的跳出循环地址   visit(ctx.statement()); // 循环体的内容   labelCounter++;  // 跳回循环开始的地址，继续下一次循环  codeResult.add(new IntermediateCode(labelCounter,"j", "-", "-", String.*valueOf*(startLabel)));  // 跳出循环的指令  codeResult.add(new IntermediateCode(startLabel2,"j", "-", "-", String.*valueOf*(labelCounter+1)));   return null; } |

首先观察循环语句的语法规则

|  |
| --- |
| Java whileStatement: WHILE condition DO statement; |

包含一个关系表达式与DO后面的“语句”

则我们首先需要得到关系表达式的内容

|  |
| --- |
| Java // 关系表达式的语法规则 condition: expression (EQUAL | NOTEQUAL | LESS | LESSEQUAL | GREATER | GREATEREQUAL) expression;  String expression0 = ctx.condition().expression(0).getText(); // 获得关系表达式的第一个expression String operator = ctx.condition().getChild(1).getText(); // 比较符，关系运算符 String expression1 = ctx.condition().expression(1).getText(); // 关系表达式对的参数二，第二个expression |

然后进行中间代码的存储，以测试程序为例子，进行详细说明

|  |
| --- |
| Java labelCounter++;// 输入一条结果前，地址+1 int startLabel = labelCounter; // while的起始地址，之后循环跳回来 codeResult.add(new IntermediateCode(startLabel,"j"+operator, expression0, expression1, String.*valueOf*(startLabel+2))); labelCounter++; // 关系表达式判断为假，跳出循环的中间代码，先不存中间代码，因为不知道跳出后的地址 int startLabel2 = labelCounter; // while的循环跳出的开始地址  visit(ctx.statement()); // 循环体的内容  labelCounter++; // 跳回循环开始的地址，继续下一次循环 codeResult.add(new IntermediateCode(labelCounter,"j", "-", "-", String.*valueOf*(startLabel))); // 跳出循环的中间代码 codeResult.add(new IntermediateCode(startLabel2,"j", "-", "-", String.*valueOf*(labelCounter+1))); |

测试程序

PROGRAM add

VAR x,y;

BEGIN

x:=1;

y:=2;

WHILE x<5 DO x:=x+1;

IF y>0 THEN y:=y-1;

y:=y+x

END

结果

100(:=, 1, -, x)

101(:=, 2, -, y)

102(j<, x, 5, 104)

103(j, -, -, 106)

104(+, x, 1, x)

105(j, -, -, 102)

106(j>, y, 0, 108)

107(j, -, -, 109)

108(-, y, 1, y)

109(+, y, x, y)

首先是102的关系表达式，如果 x<5为假，则执行103跳出循环，跳到106来继续执行，但是106这个地址是根据循环体来决定的（DO的内容），所以103(j, -, -, 106)必须在访问循环体结束后再存，访问循环体结束后，104(+, x, 1, x)即x:=x+1，存入继续下一次循环的中间代码105(j, -, -, 102)，即跳回102继续下一次循环，也可以确定，跳出循环后的地址为105的后面，106，则将103(j, -, -, 106)存入结果中。

主函数的内容，调用visitor，并将结果存入文件中

|  |
| --- |
| Java public static void main(String[] args) {  String filePath = "D://CompilationPrinciples//grammer//src//test.txt"; // 输入文件路径  String outputFilePath = "D://CompilationPrinciples//grammer//src//result.txt"; // 结果文件路径  try {  String sourceCode = *readFromFile*(filePath);  PL0Lexer lexer = new PL0Lexer(CharStreams.*fromString*(sourceCode));  CommonTokenStream tokens = new CommonTokenStream(lexer);  PL0Parser parser = new PL0Parser(tokens);  // 获取语法树的根节点  PL0Parser.ProgramContext tree = parser.program();  // 使用自定义的Visitor遍历语法树，生成中间代码  MyPL0Visitor visitor = new MyPL0Visitor();  visitor.visit(tree);   // 获取生成的中间代码字符串  ArrayList<IntermediateCode> codeResult = visitor.getCodeResult();  Collections.*sort*(codeResult, Comparator.*comparingInt*(IntermediateCode::getstart));   // 按照中间代码的开始地址进行排序  // 输出到文件  *writeToFile*(outputFilePath, codeResult);  // 结果打印到控制台  for (IntermediateCode code : codeResult) {  System.*out*.println(code.toString());  }  } catch (IOException e) {  e.printStackTrace();  } } |

**4.系统测试**

测试程序

PROGRAM add

VAR x,y;

BEGIN

x:=1;

y:=2;

WHILE x<5 DO x:=x+1;

IF y>0 THEN y:=y-1;

y:=y+x

END

结果

100(:=, 1, -, x)

101(:=, 2, -, y)

102(j<, x, 5, 104)

103(j, -, -, 106)

104(+, x, 1, x)

105(j, -, -, 102)

106(j>, y, 0, 108)

107(j, -, -, 109)

108(-, y, 1, y)

109(+, y, x, y)