编译原理任务二文档

1. 整体介绍

1.1 项目说明

本项目基于ANTLR语言识别工具实现了一个PL/0语言编译器。项目使用语言为`Java`，使用`IDEA`作为集成开发环境。本项目主要通过给ANTLR工具提供定义好的语法文件，让其自动生成**词法分析器**和**语法分析器**，并利用生成的词法分析和语法分析相关代码完成中间代码生成这一过程。

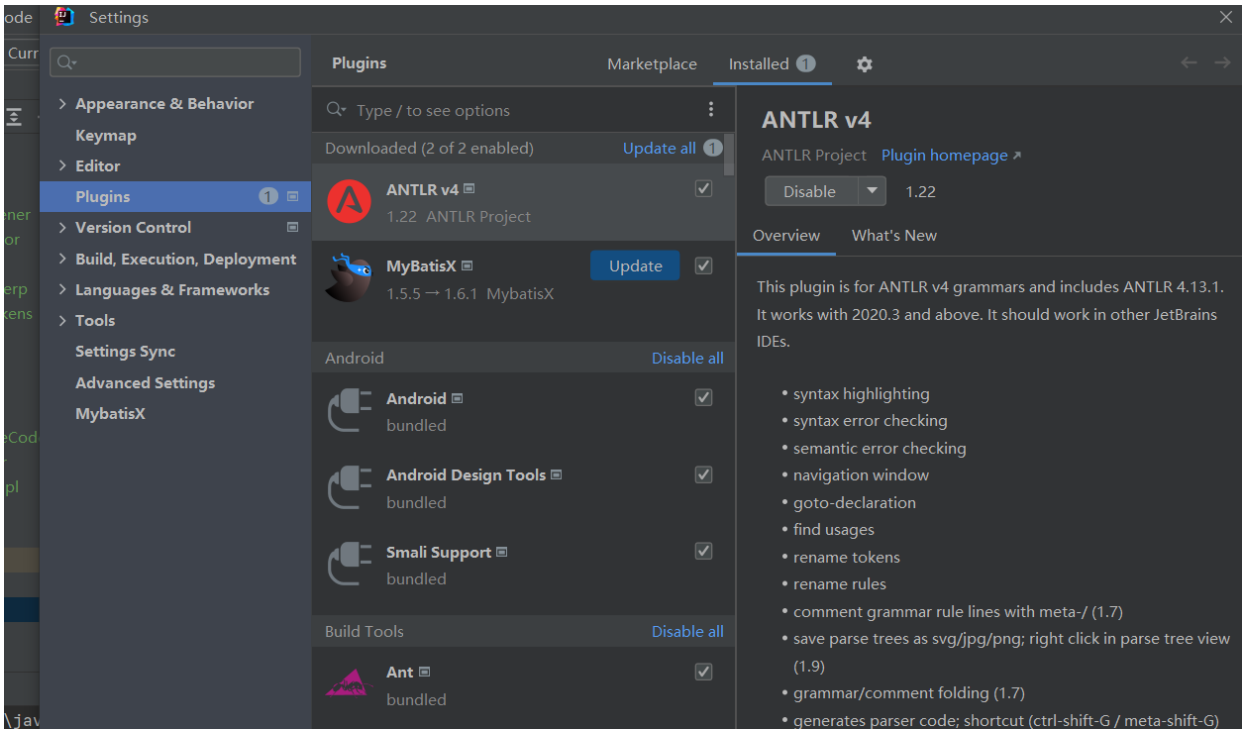
本项目中我们使用的Java开发工具包（JDK）版本为`JDK 21`，并整合了`1.22`版本的ANTLR V4插件。

1.2 ANTLR工具配置

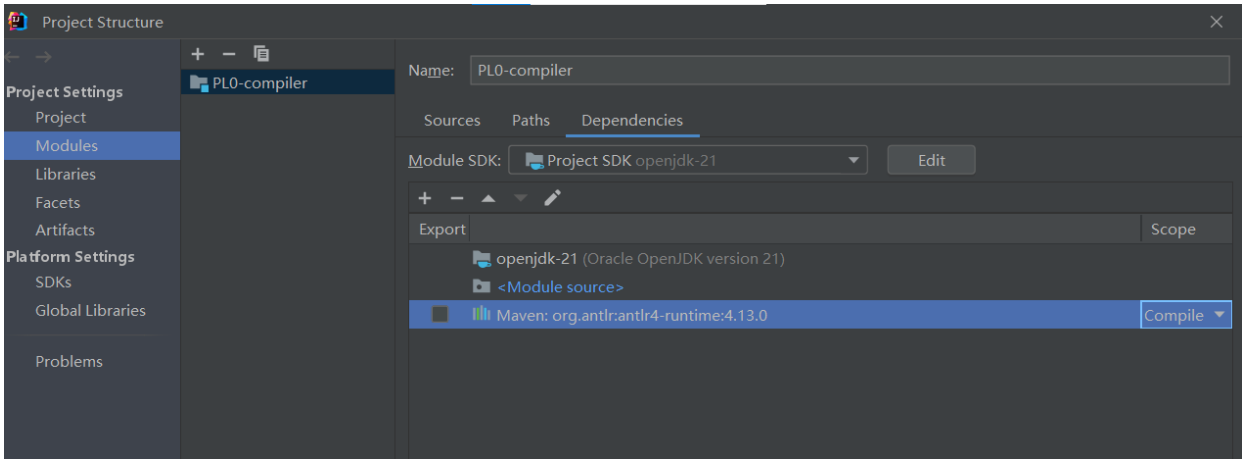
ANTLR4（ANother Tool for Language Recognition）是一种强大的语言识别工具，用于生成词法分析器和语法分析器。它采用基于**LL(*)**的语法分析算法，支持诸如词法规则、语法规则和语法动作等高度灵活的语法规范定义。ANTRL4引入的自动语法分析树创建与遍历机制，极大地提高了语言识别程序的开发效率。

本项目中由于是在IDEA这个集成式开发环境中使用ANTLR4，因此可以通过安装插件的方式来引入。

在`File`->`Settings`->Plugins->`MarketPlace`中搜索`ANTLR V4`下载安装即可，我使用的版本为1.22。



为了便于后续使用，在创建我们的`PL0_Complier`项目之后，需要导入对应的ANTLR依赖。首先需要下载ANTLR的Jar文件，即对应的Java可执行文件。点击`File`->`Project Structure`。在该窗口中选择`Dependencies`下方的+号，选择`JARs or Directories`，浏览找到刚才下载的文件，将其加入到依赖中，如图所示。



配置成功。

2. 语法规则文件

ANTLR4的使用需要在项目中创建一个以`.g4`为扩展名的ANTLR文件供其读取。它使用类似于EBNF（扩展巴科斯范式）的语法规范。在ANTLR4文件中，可以定义诸如标识符、关键字、操作符、语句、表达式等的规则，以及它们之间的关系。**注意**：在 Antlr 的规则文件中，越是前面声明的规则，优先级越高。

本项目中，我们针对PL/0语言，定义了如下所述的词法规则和语法规则：

2.1 词法规则

与PL/0的词法规则相匹配的，我们在 `.g4` 文件中加入了以下词法规则部分内容来定义词法单元的基本构建

keywords(关键字)

`PROGRAM,WHILE,DO,IF,THEN,BEGIN,END,CONST,VAR`：这些是关键字，用于标识特定的语法结构。

```
PROGRAM: P R O G R A M;
WHILE: W H I L E;
DO: D O;
IF: I F;
THEN: T H E N;
BEGIN: B E G I N;
END: E N D;
CONST: C O N S T;
VAR: V A R;
```

标识符和数字

- `STRING`：用于定义标识符，它以字母开头，后面可以跟随字母和数字的组合。
- `NUMBER`：用于定义数字，表示一个或多个数字的序列。

```
NUMBER
    : [0-9]+
    ;

STRING
    : [a-z] [a-z0-9]*
    ;
```

运算符

- `RELATIONAL_OPERATOR`：用于定义关系运算符，包括等于、不等于、小于、小于等于、大于、大于等于

```
RELATIONAL_OPERATOR: '=' | '<>' | '<' | '<=' | '>' | '>=';
```

除此以外还定义了**片段规则**

- `fragment A`, `fragment B`, ..., `fragment Z`：这些是片段规则，它们用于定义字母的片段，可以在其他规则中引用，以减少规则的重复性。

```
fragment A: 'A';
fragment B: 'B';
fragment C: 'C';
fragment D: 'D';
...
fragment Y: 'Y';
fragment Z: 'Z';
```

上述词法规则与项目要求中的PL/0词法规则基本匹配。

2.2 语法规则

与PL/0的语法规则相匹配的，我们在 `.g4` 文件中加入了以下语法规则部分内容来定义语法的基本结构：

Program 相关

- `program`：定义了整个程序的语法规则，包括程序头部和代码块。
- `program_header`：定义了程序头部的语法规则，包括关键字 `PROGRAM` 和标识符。

```
program: program_header block;
program_header: PROGRAM ident;
```

常量声明相关

- `const_stat`：定义了常量声明的语法规则，包括关键字 `CONST`、常量定义和分号。
- `const_def`：定义了单个常量的语法规则，包括标识符、赋值操作符和数字。

```
const_stat: CONST const_def (',' const_def)* ';';
const_def: ident ':=' number;
```

Block 相关

- `block`：定义了程序的代码块，包括常量声明、变量声明和语句。

```
block: const_stat? var_stat? statement;
```

变量声明相关

- `var_stat`：定义了变量声明的语法规则，包括关键字 `VAR`、变量标识符列表和分号。

```
var_stat: VAR ident (',' ident)* ';';
```

语句相关规则

- `statement`：定义了语句的语法规则，包括赋值语句、复合语句、条件语句、循环语句和空语句。
- `assignstmt`：定义了赋值语句的语法规则，包括标识符、赋值操作符和表达式。
- `beginstmt`：定义了复合语句（使用 `BEGIN` 和 `END` 包裹的语句块）的语法规则。
- `condition`：定义了条件语句中的条件表达式的语法规则，包括两个表达式和关系运算符。
- `ifstmt`：定义了条件语句的语法规则，包括关键字 `IF`、条件和关键字 `THEN`。
- `whilestmt`：定义了循环语句的语法规则，包括关键字 `WHILE`、条件和关键字 `DO`。

```
statement: (assignstmt | beginstmt | ifstmt | whilestmt | emptystmt)?;
emptystmt: ;
assignstmt: ident ':=' expression;
beginstmt: BEGIN statement (';' statement)* END;
condition: expression RELATIONAL_OPERATOR expression;
ifstmt: IF condition THEN statement;
whilestmt: WHILE condition DO statement;
```

表达式相关

- `expression`: 定义了表达式的语法规则，包括可选的符号、项和可选的加法操作符。
- `optionalSign`: 定义了可选的表达式符号（正号或负号）。
- `addOperator`: 定义了加法操作符。
- `multiplyOperator`: 定义了乘法操作符。
- `term`: 定义了表达式中的项，包括因子和可选的乘法操作符。
- `factor`: 定义了表达式中的因子，包括标识符、数字和括号中的表达式。

```
expression: optionalSign term ((addOperator) term)*;
optionalSign: ('+' | '-')?;
addOperator: '+' | '-';
multiplyOperator: '*' | '/';
term: factor ((multiplyOperator) factor)*;
factor: ident | number | '(' expression ')';
```

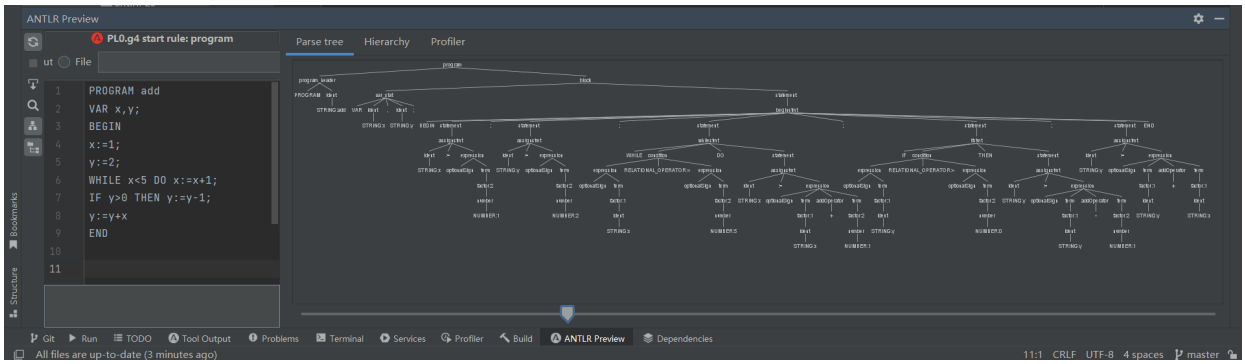
标识符和数字

- `ident`: 定义了标识符，使用 `STRING` 规则。
- `number`: 定义了数字，使用 `NUMBER` 规则。

```
ident: STRING;
number: NUMBER;
```

2.3 语法规则检验

在IDEA底部的工具栏 `ANTLR Preview`中，我们可以使用它来进行语法树的构建，也可以测试我们的语法规则设置是否正确。将项目的PL/0示例代码复制到文本输入框内，可以看到在右侧生成了对应的语法树。没有出现报错，且生成的语法树符合预期，说明我们的语法规则设置正确。

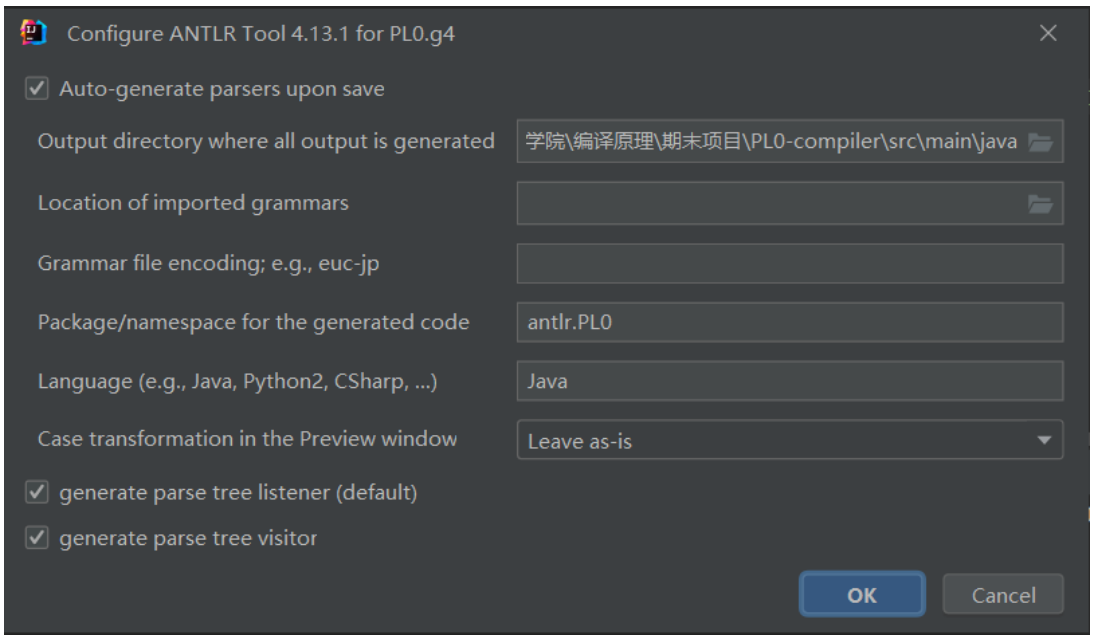


3. 词法分析与语法分析

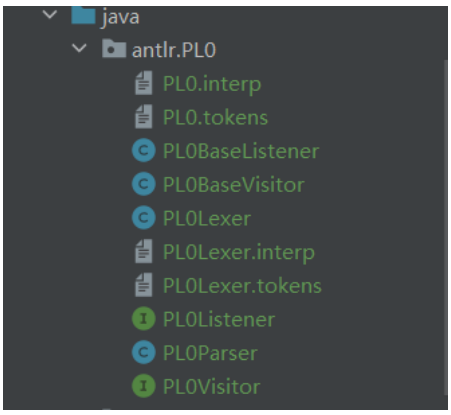
本项目主要考虑利用ANTLR4自动生成的词法分析器和语法分析器来进行词法分析与语法分析阶段。

3.1 ANTLR词法器与语法器构建

使用ANTLR可以自动构建编译器所需的词法分析器和语法分析器。当我们确认过语法规则正确后，右键语法规则文件，选择 `Configure ANTLR`。主要需要输入的配置就是词法器和语法器文件生成的位置以及选择语言为 `Java`。



然后再次右击语法规则文件，选择 `Generate ANTLR Recognizer` 生成对应PL/0语言的词法器和语法器。



3.2 实现原理

- 词法分析：

  词法分析，主要负责将符号文本分组成符号类tokens，把输入的文本转换成词法符号的程序称为词法分析器(lexer)。

  ANTLR的词法分析过程基于**正则表达式**和**有限自动机**。在语法文件中，通过正则表达式定义词法规则，描述了程序中的关键字、标识符、运算符等模式。ANTLR使用这些规则构建有限自动机，其中每个状态表示自动机在识别不同词法单元时的状态。生成的词法分析器能够按照这些规则将源代码分解为一个标记流，包含各个词法单元。

我们可以在词法分析过程进行完后，输出结果中所有的tokens。

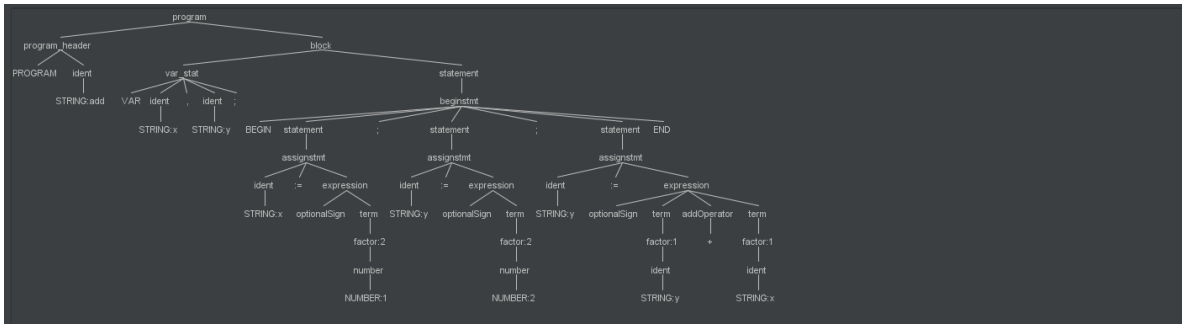
```
[@0,0:6='PROGRAM',<10>,1:0]
[@1,8:10='add',<21>,1:8]
[@2,13:15='VAR',<18>,2:0]
[@3,17:17='x',<21>,2:4]
[@4,18:18=',',<1>,2:5]
[@5,19:19='y',<21>,2:6]
[@6,20:20=';',<2>,2:7]
[@7,23:27='BEGIN',<15>,3:0]
[@8,30:30='x',<21>,4:0]
[@9,31:32=':=',<3>,4:1]
[@10,33:33='-',<5>,4:3]
[@11,34:34='1',<20>,4:4]
[@12,35:35=';',<2>,4:5]
[@13,38:38='y',<21>,5:0]
[@14,39:40=':=',<3>,5:1]
[@15,41:41='2',<20>,5:3]
[@16,42:42=';',<2>,5:4]
[@17,45:49='WHILE',<11>,6:0]
[@18,51:51='x',<21>,6:6]
[@19,52:52='<',<19>,6:7]
[@20,53:53='5',<20>,6:8]
[@21,55:56='DO',<12>,6:10]
```

- 语法分析：

  语法分析，目标就是构建一个语法解析树。语法解析的输入是tokens，输出就是一颗语法解析树。

  ANTLR的语法分析过程基于**LL(k)文法**和**递归下降算法**。在语法文件中，使用LL(k)文法定义了语法规则，描述了程序中的语法结构，如程序、语句、表达式等。ANTLR通过递归下降算法实现语法分析，为每个非终结符生成相应的递归下降子程序。通过向前查看符号，ANTLR选择最佳的匹配规则，并构建语法树，反映源代码的层次结构。

  语法分析的结果实际上就是如下的语法树：



4. 中间代码生成

默认情况下，ANTLR使用内建的遍历器访问生成的语法分析树，并为每个遍历时可能触发的事件生成一个语法分析树监听器接口。除了监听器的方式，还有一种遍历语法分析树的方式:**访问者模式**，在本项目中，我们就是通过实现一个继承自 `PL0BaseVisitor` 的 `PL0VisitorImpl` 类来完成中间代码生成部分的工作，同时我们还定义了一个名为 `IntermediateCode` 的类作为中间代码的数据结构，便于其生成和展示。

4.1 实现原理与特点

我们希望控制遍历语法分析树的过程，通过显式的方法调用来访问子节点。语法中的每条规则对应接口中的一个visit方法。

ANTLR内部为访问者模式提供的支持代码会在根节点处调用该位置的visit方法，接下来，该方法的实现将会调用其内部的visit方法，并将所用的子节点作为参数传递给它，从而继续遍历。

这种访问者模式来遍历语法树有以下优势：

1. 我们可以显示定义遍历语法树的顺序。
2. 不需要与antlr遍历类`ParseTreeWalker`一起使用，可以直接对tree操作。
3. 动作代码与文法产生式解耦，利于文法产生式的重用。
4. visitor方法可以直接返回值，返回值的类型必须一致，不需要使用map这种节点间传值方式，效率高。

4.3 具体实现

4.3.1 访问语法树

在ANTLR中，我们使用访问者模式对语法树进行遍历操作。为此，我们编写了一个`PLOVisitorImpl`类，该类继承自ANTLR生成的`PLOBaseVisitor`类。该类将用于遍历和处理语法树。

对于语法规则中的每个结构，我们在`PLOVisitorImpl`类中override相应的访问方法。这些方法会在遍历语法树时被调用。我们定义了特定语法结构上执行的操作。对于语法规则中每一部分的处理，我们都会首先判断`stopTraversal`是否为true，不是再进行后续操作。

在程序的主逻辑中，创建ANTLR生成的解析器和词法分析器的实例，并解析输入文本，得到语法树的根节点。然后，创建编写的`PLOVisitorImpl`类的实例。调用语法树的根节点的`accept`方法，将Visitor传递给它。ANTLR根据语法树的结构逐层调用Visitor类中对应的方法。例如，对于语法规则中的每个规则，Visitor会调用相应的`visit`方法。同时我们选择递归调用`visit`方法以访问当前节点的子节点。这使得可以深入到语法树的更深层次，实现对整个语法结构的完整处理。

下面展示基本的实现逻辑：

- 对语句的处理：

  通过`StatementContext`提取可能存在的不同类型的子语句，包括`beginstmt`、`assignstmt`、`ifstmt`、`whilestmt`和`emptystmt`。判断每个部分是否为空，如果不为空，则遍历该子节点。

```
4 usages  new *
@Override
public T visitStatement(PL0Parser.StatementContext ctx){
    if (stopTraversal) {
        return null;  // 结束遍历
    }
    PL0Parser.BeginstmtContext beginstmt=ctx.beginstmt();
    PL0Parser.AssignstmtContext assignstmt=ctx.assignstmt();
    PL0Parser.IfstmtContext ifstmt=ctx.ifstmt();
    PL0Parser.WhilestmtContext whilestmt=ctx.whilestmt();
    PL0Parser.EmptystmtContext emptystmt=ctx.emptystmt();
    if(beginstmt!=null)
        visitBeginstmt(beginstmt);
    else if(assignstmt!=null)
        visitAssignstmt(assignstmt);
    else if(ifstmt!=null)
        visitIfstmt(ifstmt);
    else if(whilestmt!=null)
        visitWhilestmt(whilestmt);
    return null;
}
```

- 对begin语句的处理：通过`BeginstmtContext`提取包含在`BEGIN`语句块中的语句列表，对其进行遍历并递归处理其中的每个语句。

```
@Override
public T visitBeginstmt(PL0Parser.BeginstmtContext ctx){
    if (stopTraversal) {
        return null;  // 结束遍历
    }
    List<PL0Parser.StatementContext> statementList=ctx.statement();

    for(PL0Parser.StatementContext statement : statementList){
        visitStatement(statement);
    }
    return null;
}
```

- 对常量声明的处理：

  常量的声明部分涉及两个函数`visitConst_stat()`和`visitConst_def`。我们首先获取常量定义列表并遍历其中的每一个常量定义子节点，然后在`visitConst_def`中遍历到每一个常量标识符，先利用`:=`、标识符和数字生成对应的中间代码，并将常量的值加入到提前声明的`constMap`中。

```java
@Override
public T visitConst_stat(PL0Parser.Const_statContext ctx){
    if (stopTraversal) {
        return null;  // 结束遍历
    }
    List<PL0Parser.Const_defContext> constDefContexts=ctx.const_def();
    for(PL0Parser.Const_defContext constDefContext : constDefContexts){
        visitConst_def(constDefContext);
    }
    return null;
}
```

```java
@Override
public T visitConst_def(PL0Parser.Const_defContext ctx){
    if (stopTraversal) {
        return null;  // 结束遍历
    }
    PL0Parser.IdentContext ident=ctx.ident();
    PL0Parser.NumberContext number=ctx.number();
    if(constMap.containsKey(ctx.ident())){
        // 输出错误提示到标准错误流
        System.err.println("Error: Constant "+ ctx.ident().getText()+ " is already defined.");
        this.stopTraversal=false;
        return null;
    }
    generateCode( operator: ":=", number.getText(), operand2: "_", ident.getText());
    //加入常量Map
    constMap.put(ident.getText(),number.getText());
    return null;
}
```

- 对变量声明的处理：对变量的处理类似，只是不需要在声明中给其指定值。直接在 `varMap` 中存储遍历到的变量名。
- 对赋值语句的处理。首先可以直接获取目标变量，再访问右侧的表达式子节点，并用 `expression` 的结果生成赋值语句的中间代码。最后将变量值添加到 `varMap` 中即可。
- 对表达式的处理。

  在对表达式的处理过程中，我们实现了 `visitExpression`。首先，它从语法树中获取了表达式中的项和加法运算符的列表。然后通过判断是否存在加法运算符，分别处理了不涉及加减法运算的情况和涉及加减法运算的情况。

  - 在不涉及加减法运算的情况下，该方法判断是否存在可选的前置符号（正号或负号），若存在则生成对应的中间代码，并更新临时变量；若不存在则直接调用 `visitTerm` 处理第一个项并返回结果。
  - 在涉及加减法运算的情况下，方法通过循环遍历项和运算符列表，逐一生成中间代码。在第一轮循环中，处理了第一个项，考虑了前置的可选符号。在后续循环中，处理了后续项和运算符，每次生成中间代码时都使用了前一个循环生成的临时变量。

  最终，方法返回的结果是最后一轮循环生成的临时变量，该临时变量包含了整个表达式的计算结果。

```java
@Override
public T visitExpression(PL0Parser.ExpressionContext ctx){
    if (stopTraversal) {
        return null;  // 结束遍历
```

```
    }
    //获取term列表
    List<PLOParser.TermContext> termList=ctx.term();

    //获取加法运算符列表
    List<PLOParser.AddOperatorContext> oprList=ctx.addOperator();

    //若为纯数字或变量(无加减法运算)
    if(oprList.isEmpty()){
        if(!ctx.optionalSign().getText().equals("")){
            generateCode(ctx.optionalSign().getText(),(String)
visitTerm(termList.get(0)),"_",updateTempVar());
        }
        else {
            return visitTerm(termList.get(0));
        }
    }
    else{
        for(int i=0;i< termList.size()-1;i++){
            if(i==0){
                //判断是否含有前置可选+、-
                if(!ctx.optionalSign().getText().equals(""))
                    generateCode(ctx.optionalSign().getText(),(String)
visitTerm(termList.get(i)),"_",updateTempVar());
                else
                    generateCode(oprList.get(i).getText(),(String)
visitTerm(termList.get(i)),
(String)visitTerm(termList.get(i+1)),updateTempVar());
            }
            else {
                generateCode(oprList.get(i).getText(),this.tempVar,
(String)visitTerm(termList.get(i+1)),updateTempVar());
            }
        }

    }
    return (T) this.tempVar;
}
```

- 对项的处理：

  我们使用`visitTerm`方法处理 PLO 语言中的项。如果项只包含一个因子，直接调用`visitFactor`处理并返回结果。如果项包含多个因子，通过循环遍历因子列表和乘法运算符列表，逐一生成中间代码。在第一轮循环中，判断是否为首项，直接生成中间代码，后续循环使用前一个循环生成的临时变量。最终返回的结果是最后一轮循环生成的临时变量，包含整个项的计算结果，并且可以支持后续的编译过程。

```
@Override
public T visitTerm(PLOParser.TermContext ctx){
    if (stopTraversal) {
        return null;  // 结束遍历
```

```
    }
    List<PL0Parser.FactorContext> factorList=ctx.factor();
    List<PL0Parser.MultiplyOperatorContext> multiplyList=ctx.multiplyOperator();
    if(multiplyList.isEmpty()){
        return (T) visitFactor(factorList.get(0));
    }
    else{
        for(int i=0;i< factorList.size()-1;i++){
            if(i==0){
                //判断是否为首项（不需要使用tempVar)
                generateCode(multiplyList.get(i).getText(),(String)
visitFactor(factorList.get(i)),
(String)visitFactor(factorList.get(i+1)),updateTempVar());
            }
            else {
                generateCode(multiplyList.get(i).getText(),this.tempVar,
(String)visitFactor(factorList.get(i+1)),updateTempVar());
            }
        }
    }

    return (T) this.tempVar;
}
```

- 对因子的处理：

  我们使用`visitFactor`方法处理 PL0 语言中的因子。它首先检查因子是否是一个已声明但未赋值的变量，如果是则输出错误提示。然后返回因子的文本表示，即变量名或常数。

```
@Override
public T visitFactor(PL0Parser.FactorContext ctx) {
    if (stopTraversal) {
        return null;  // 结束遍历
    }
    if(ctx.ident()!=null&&varMap.containsKey(ctx.getText())&&varMap.get(ctx.getText())==null){
        // 输出错误提示到标准错误流，包含行号和字符位置信息
        Token errorToken = ctx.getStart();
        System.err.println("Error at line " + errorToken.getLine() + ", position " + errorToken.getCharPositionInLine() +
                ": Using unassigned variable: " + ctx.getText());
        this.stopTraversal=true;
        return null;
    }
    else if(ctx.ident()!=null&&!varMap.containsKey(ctx.ident().getText())){
        // 输出错误提示到标准错误流，包含行号和字符位置信息
        Token errorToken = ctx.getStart();
        System.err.println("Error at line " + errorToken.getLine() + ", position " + errorToken.getCharPositionInLine() +
                ": Using undefined variable: " + ctx.getText());
        this.stopTraversal=true;
        return null;
    }
    return (T) ctx.getText();
}
```

- 对条件语句的处理：

  我们使用`visitIfstmt`方法处理 PL0 语言中的`if`语句。它首先获取`if`语句中的条件和语句部分。然后，调用`visitCondition`处理条件，生成相应的中间代码。接着，生成一个临时地址用于存储当前中间代码的地址，然后将该临时地址存入`tempAddress`中。相当于此时中间代码已经生成，但跳转结果仍等待回填。

如果 `if` 条件满足，执行下面的语句，即调用 `visitStatement` 处理 `if` 语句的主体部分。在处理完主体后，补全生成刚才的中间代码，使用 `this.addressCount` 作为目标地址，表示跳转到 `if` 语句结束的位置。

```
@Override
public T visitFactor(PL0Parser.FactorContext ctx) {
    if (stopTraversal) {
        return null;  // 结束遍历
    }
    if(ctx.ident()!=null&&varMap.containsKey(ctx.getText())&&varMap.get(ctx.getText())==null){
        // 输出错误提示到标准错误流，包含行号和字符位置信息
        Token errorToken = ctx.getStart();
        System.err.println("Error at line " + errorToken.getLine() + ", position " + errorToken.getCharPositionInLine() +
                ": Using unassigned variable: " + ctx.getText());
        this.stopTraversal=true;
        return null;
    }
    else if(ctx.ident()!=null&&!varMap.containsKey(ctx.ident().getText())){
        // 输出错误提示到标准错误流，包含行号和字符位置信息
        Token errorToken = ctx.getStart();
        System.err.println("Error at line " + errorToken.getLine() + ", position " + errorToken.getCharPositionInLine() +
                ": Using undefined variable: " + ctx.getText());
        this.stopTraversal=true;
        return null;
    }
    return (T) ctx.getText();
}
```

- 对循环语句的处理：

  `visitWhilestmt` 方法实现了 我们PL0 语言中的 `WHILE` 循环语句的访问和处理逻辑。首先，它获取了 `while` 循环的条件和主体语句。然后，记录了当前地址作为条件判断位置。接着，调用 `visitCondition` 处理循环的条件，生成相应的中间代码。

  同条件语句类似，在处理条件的基础上，方法生成一个临时地址，并将该临时地址存储在 `tempAddress` 中，表示当前中间代码已经生成，但目标地址等待回填。随后，生成一条条件跳转的中间代码，如果条件不满足，跳转到之前记录的条件判断位置。接着，调用 `visitStatement` 处理循环的主体语句。并生成中间代码表示进入循环，跳转到 `condition` 判断处。

  最后，补全生成刚才不满足跳转的中间代码，跳转到的地址为 `this.addressCount` ，即退出循环。这样的实现使得代码能够正确处理 `while` 循环语句，包括条件的判断、主体语句的处理以及循环跳转目标的生成。生成的中间代码为后续的编译过程提供了正确的循环控制指令序列。

```
@Override
public T visitWhilestmt(PL0Parser.WhilestmtContext ctx){
    if (stopTraversal) {
        return null;  // 结束遍历
    }
    PL0Parser.ConditionContext condition = ctx.condition();
    PL0Parser.StatementContext statement = ctx.statement();
    int conditionAddress=this.addressCount;
    visitCondition(condition);
    //不满足while条件的,先生成地址，等待回填
    int tempAddress=addressCount;
    //该中间代码已经生成，等待回填
    this.addressCount++;
    //条件满足时执行下面的语句(访问statement)
    visitStatement(statement);
    //进入循环，跳到condition判断处
    generateCode( operator: "j", operand1: "_", operand2: "_",Integer.toString(conditionAddress));
    generateCode(tempAddress, operator: "j", operand1: "_", operand2: "_",Integer.toString(this.addressCount));
    return null;
}
```

- 对条件的处理：

我们使用`visitCondition`方法处理 PL0 语言中的条件。首先，它获取条件中的两个表达式，并使用条件中的关系运算符生成一条条件跳转的中间代码。该中间代码含义为：如果条件满足，跳转到当前`addressCount`加上 2 的位置，即符合且需要执行的位置；否则，继续执行下一条指令。

同时我们考虑了条件的两个表达式，通过调用`visitExpression`处理每个表达式，确保正确地生成了相应的中间代码。生成的中间代码中包含了条件的关系运算符和两个表达式的计算结果，以及一个用于条件跳转目标的新的临时地址。

```
@Override
public T visitCondition(PL0Parser.ConditionContext ctx){
    if (stopTraversal) {
        return null;  // 结束遍历
    }
    List<PL0Parser.ExpressionContext> expressionList=ctx.expression();
    generateCode( operator: "j"+ctx.RELATIONAL_OPERATOR().getText(),(String) visitExpression(expressionList.get(0)),(String) vis
    return null;
}
```

4.3.2 错误处理

ANTLR中本身已经提供了相对完善的词法分析和语法分析错误处理机制。在词法分析中，ANTLR通过识别词法错误并生成对应的错误节点，将错误信息与语法树关联起来。这样，用户可以通过遍历语法树来检测和处理词法错误，确保错误信息得以捕获和报告。

在语法分析中，ANTLR同样提供了健壮的错误处理机制。当输入文本不符合语法规则时，ANTLR会尝试进行错误恢复，继续分析并尽可能多地找到后续语法结构。它还允许用户定义自定义的错误处理策略，包括错误恢复、报告和处理方式。

在本项目中，我们所需要实现的主要是对中间代码生成部分的错误处理逻辑。我们使用了一个布尔型变量`stopTraversal`作为是否结束遍历的标志，一旦出现错误，我们会使用`System.err.println()`函数打印出错误的位置、原因等信息，并结束遍历语法树。

主要错误处理内容分为以下几点：

1. 当某个变量未被定义而被赋值或使用时识别错误并报错；

```
else if(!varMap.containsKey(target)){
    // 获取错误位置的 Token 对象
    Token errorToken = ctx.ident().getStart();
    // 输出错误提示到标准错误流，包含行号和字符位置信息
    System.err.println("Error at line " + errorToken.getLine() + ", position " +
            errorToken.getCharPositionInLine() + ": Using undefined variable " + target);
    this.stopTraversal=true;
    return null;
}
```

```
else if(ctx.ident()!=null&&!varMap.containsKey(ctx.ident().getText())){
    // 输出错误提示到标准错误流，包含行号和字符位置信息
    Token errorToken = ctx.getStart();
    System.err.println("Error at line " + errorToken.getLine() + ", position " + errorToken.getCharPositionInLine() +
            ": Using undefined variable: " + ctx.getText());
    this.stopTraversal=true;
    return null;
}
```

2. 当某个变量未被赋值而被直接用于为其他变量赋值时识别错误并报错；

```
if(ctx.ident()!=null&&varMap.containsKey(ctx.getText())&&varMap.get(ctx.getText())==null){
    // 输出错误提示到标准错误流，包含行号和字符位置信息
    Token errorToken = ctx.getStart();
    System.err.println("Error at line " + errorToken.getLine() + ", position " + errorToken.getCharPositionInLine() +
            ": Using unassigned variable: " + ctx.getText());
    this.stopTraversal=true;
    return null;
}
```

3. 当重复定义变量或常量时报错；

```
if(constMap.containsKey(identContext.getText())){
    // 获取错误位置的 Token 对象
    Token errorToken = identContext.getStart();
    // 输出错误提示到标准错误流，包含行号和字符位置信息
    System.err.println("Error at line " + errorToken.getLine() + ", position " +
            errorToken.getCharPositionInLine() + ": Constant " + identContext.getText() + " is already defined.");
    this.stopTraversal=true;
    return null;
}
else if(varMap.containsKey(identContext.getText())){
    // 获取错误位置的 Token 对象
    Token errorToken = identContext.getStart();
    // 输出错误提示到标准错误流，包含行号和字符位置信息
    System.err.println("Error at line " + errorToken.getLine() + ", position " +
            errorToken.getCharPositionInLine() + ": Variable " + identContext.getText() + " is already defined.");
    this.stopTraversal=true;
    return null;
}
```

4. 当给常量重复赋值时报错。

```
if(constMap.containsKey(target)){
    // 获取错误位置的 Token 对象
    Token errorToken = ctx.ident().getStart();
    // 输出错误提示到标准错误流，包含行号和字符位置信息
    System.err.println("Error at line " + errorToken.getLine() + ", position " +
             errorToken.getCharPositionInLine() + ": Constant " + target + " cannot be reassigned.");
    this.stopTraversal=true;
    return null;
}
```

4.3.3 中间代码类

为了便于后续生成和输出中间代码，我们在项目中设计了一个中间代码类，以便于将该数据结构封装。该类主要包含以下设计：

首先是类内属性：

- `address`（地址）：用于标识中间代码的位置。这是一个整数，代表中间代码的地址信息。目的是为了追踪和定位中间代码的执行位置。
- `operator`（操作符）：表示中间代码执行的操作或指令。这是一个字符串，可能对应于像加法、乘法等基本操作。目的是为了确定执行的具体操作。
- `operand1`、`operand2`（操作数1和2）：表示操作的输入值。这是字符串类型，可能是变量名、常数或其他中间结果。目的是为了存储操作所需的输入信息。
- `result`（结果）：表示操作的输出结果。同样是一个字符串，可能是新的变量或者用于存储中间结果的地方。目的是为了存储执行操作后的结果。

除了类内基本的构造方法和Getter 和 Setter 方法等，我们还主要设计了下面两种方法来实现中间代码的生成和输出。

- `generateFormatCode()` 方法: 生成格式化的字符串，包括地址和操作以及它们的操作数和结果。这有助于以易读的形式展示中间代码，方便调试和理解。目的是为了提供一个用户可读的中间代码表示。
- `printFormatCode()` 方法: 将格式化后的代码打印到控制台或者重定向到文件中。这方便了查看中间代码的过程，可用于调试和验证。目的是为了方便显示格式化的中间代码。

5. 系统测试

5. 系统测试

同任务一类似的，我们设计了四种测试案例来进行我们的系统测试。

5.1 测试一：基本案例

```
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
```

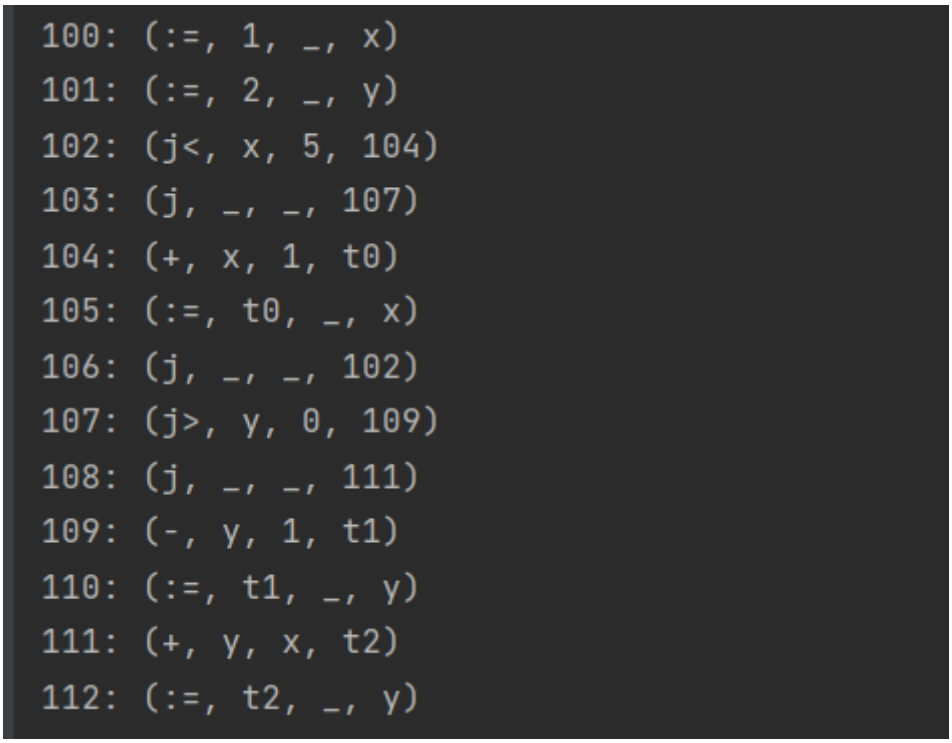
结果如下：

```
100: (:=, 1, _, x)
101: (:=, 2, _, y)
102: (j<, x, 5, 104)
103: (j, _, _, 107)
104: (+, x, 1, t0)
105: (:=, t0, _, x)
106: (j, _, _, 102)
107: (j>, y, 0, 109)
108: (j, _, _, 111)
109: (-, y, 1, t1)
110: (:=, t1, _, y)
111: (+, y, x, t2)
112: (:=, t2, _, y)
```

5.2 测试二：多循环嵌套案例

```
PROGRAM mw
VAR x, y;
BEGIN
x:=0;
y:=0;
WHILE x<10 DO
WHILE y<9 DO
WHILE x<8 DO
BEGIN
WHILE y<7 DO
BEGIN
x:=y+1;
y:=x+1
END;
x:=y
END;
END
```

结果如下：



5.3 测试三：变量未赋值错误案例

```
PROGRAM add
VAR x,y;
BEGIN
y:=2;
WHILE x<5 DO x:=x+1;
IF y>0 THEN y:=y-1;
y:=y+x
END
```

会出现变量未赋值的错误提示，并显示错误位置（行列数）以及错误原因：

```
Error at line 5, position 6: Using unassigned variable: x
```

5.4 测试四：常量重复赋值

```
PROGRAM add
CONST a:=1;
VAR x,y;
BEGIN
a:=3;
x:=1;
y:=2;
WHILE x<5 DO x:=x+1;
IF y>0 THEN y:=y-1;
y:=y+x
END
```

会出现常量重复赋值的错误提示，并显示错误位置（行列数）以及错误原因：

```
Error at line 5, position 0: Constant a cannot be reassigned.
```