同濟大學

编译原理期末项目：PL/0 编译器实现

小组成员

严文昊	2050605
李恒鑫	2050735
王蔚达	2151300
夏尧民	2152826

指导老师

高珍

一、 项目概述

PL/0 是一种教学级编程语言，专为编译原理的学习而设计，它通过简化的语法和结构，使我们能够轻松掌握词法分析、语法分析和代码生成等编译过程的基础概念，适用于实现教学目的的编译器项目。

本项目旨在通过实现 PL/0 编译器和应用现代编译工具，综合培养理论知识与实践技能。完成这两个任务不仅验证了对编译原理知识的掌握，也锻炼了使用实际工具进行软件开发的能力，为未来在计算机科学和软件工程领域的深入学习和研究打下坚实基础。

任务一的核心目标是实现一个简单的 PL/0 编译器，掌握并应用编译器构造的基础算法。通过选择适当的高级程序设计语言和算法，团队将开发一个能够将 PL/0 源代码转换为三地址中间代码的编译器，同时确保整个编译过程在一次遍历中完成，并采用语法制导翻译技术。任务二要求利用现代 compiler-compiler 工具如 Flex/Bison 或 ANTLR，进行编译器的二次开发。这将不仅加深对编译原理的理解，也将提升使用现代软件开发工具进行编译器设计的实践能力。

二、 团队组成与分工

姓名	学号	分工
严文昊	2050605	使用 ANTLR 生成 PL/0 语言的编译器
李恒鑫	2050735	语法分析算法实现
王蔚达	2151300	词法分析算法实现
夏尧民	2152826	中间代码生成代码实现

我们使用 Github 进行协同编程，仓库链接为：*https://github.com/tjuDavidWang/HW-Compilers*

三、 需求分析

3.1 PL/0 语言简述

PL/0 语言以其精简的设计和清晰的结构特色，专为编译器教学环境打造。它去除了复杂语言的繁琐特性，保留了编程语言的核心元素，如条件控制、循环、变量声明等，使之成为理解和实现编译过程的理想选择。

3.1.1 词法规则

类别	规则	举例
关键字	预定义的保留词，用于特定语法结构	PROGRAM, BEGIN, END, CONST, VAR, WHILE, DO, IF, THEN
标识符	以字母开头，后跟字母或数字的字符串	x, variable1
整数	由数字组成的序列	123, 4567
算符	用于数学和逻辑操作的符号	+, -, *, /
界符	用于分隔语句和表达式的符号	;, ,, (,), :=, =, <>, >, >=, <, <=

3.1.2 语法规则

语法元素	定义	描述
<程序>	<程序>→<程序首部> <分程序>	定义了 PL/0 程序的整体结构。
<程序首部>	<程序首部>→PROGRAM <标识符>	指定程序的开始，并定义程序名称。
<分程序>	<分程序>→[<常量说明>][<变量说明>]<语句>	包含可选的常量和变量声明，以及程序主体。
<常量说明>	<常量说明>→CONST <常量定义>{，<常量定义>};	定义了一个或多个常量。
<常量定义>	<常量定义>→<标识符> := <无符号整数>	为常量赋值。
<无符号整数>	<无符号整数>→<数字>{<数字>}	由一串数字组成的整数。
<变量说明>	<变量说明>→VAR <标识符>{，<标识符>};	声明一个或多个变量
<标识符>	<标识符>→<字母>{<字母> | <数字>}	标识符的构成
<复合语句>	<复合语句>→BEGIN <语句>{; <语句>} END	由 BEGIN 和 END 关键字包围的一系列语句，语句间用分号;分隔。
<语句>	<语句>→<赋值语句> | <条件语句 >| <循环语句> | <复合语句> | <空语句>	表示 PL/0 中的基本语句，可以是赋值、条件、循环、复合语句或空语句。

<赋值语句>	<赋值语句>→<标识符>:=<表达式>	将表达式的值赋给一个标识符，使用:=作为赋值符号。
<表达式>	<表达式>→[+|-]项|<表达式> <加法运算符><项>	数学表达式，可以是单个项或由加法运算符+或-连接的多个项。
<项>	<项>→<因子> | <项><乘法运算符> <因子>	表达式的一部分，可以是单独的因子或由乘法运算符*或/连接的多个因子。
<因子>	<因子>→<标识符> |<无符号整数> | (<表达式>)	表达式中的基本单位，可以是标识符、无符号整数或括号内的表达式。
<加法运算符>	<加法运算符>→ + | -	用于表达式中的加法(+)或减法(-)操作。
<乘法运算符>	<乘法运算符>→ * | /	用于表达式中的乘法(*)或除法(/)操作。
<条件语句>	<条件语句>→IF <条件> THEN <语句>	基于特定条件执行特定语句，使用 IF ... THEN 结构。
<循环语句>	<循环语句>→WHILE <条件> DO <语句>	根据条件重复执行语句，使用 WHILE ... DO 结构。
<条件>	<条件>→<表达式> <关系运算符> <表达式>	由两个表达式和一个关系运算符组成的条件判断表达式。
<关系运算符>	<关系运算符>→ = | <> | < | <= | > | >=	用于比较两个表达式的值，包括=、<>、<、<=、>和>=。
<字母>	<字母>→a | b …| x | y | z	表示 PL/0 语言中可以使用的字母，范围从 a 到 z。
<数字>	<数字>→0 | 1 | … | 8| 9	表示 PL/0 语言中的数字，范围从 0 到 9。

注：[ ]中的项表示可选，{ }中的项表示可重复若干次。

3.2 任务一：编译算法实现

目标:

实现一个功能齐全的 PL/0 编译器，该编译器能够读取 PL/0 语言编写的源代码，并转换为对应的三地址中间代码。

功能要求:

(1) **词法分析器:**

可以使用 Thompson 算法、子集法、等价状态法等算法实现。

a. 能够识别并标记 PL/0 语言中的关键字、标识符、整数、算符和界符。

b. 必须正确处理空格、制表符和换行符等空白字符。

c. 应能报告词法错误，并给出错误位置。

(2) 语法分析器:

可以使用递归下降分析法、预测分析程序、LR 分析法等算法实现。

a. 根据 PL/0 的语法规则进行语法分析，构建抽象语法树（AST）。

b. 在发现语法错误时，需能够提供错误信息并尽可能进行错误恢复。

(3) 中间代码生成器:

可以使用属性文法、翻译子程序等方法实现。

a. 从 AST 生成三地址代码，这些代码应清晰、高效，并易于后续代码优化。

b. 生成的代码需要保持源程序的逻辑结构和数据流。

性能要求:

- 编译器的性能需满足一遍扫描的要求，尽可能减少编译时间。
- 内存使用应当优化，确保编译器适用于内存受限的环境。

可靠性要求:

- 编译器需要有较强的鲁棒性，对于不同的错误输入能够给出合理的反馈。

3.3 任务二：编译工作使用

目标:

利用现代 compiler-compiler 编译工具辅助编译器的开发，提升开发效率，确保项目的可维护性和扩展性。

功能要求:

(1) Flex/Bison 或 ANTLR 的选择与应用:

- 选择适合项目需求的编译工具，Flex/Bison 适合 UNIX 环境下工作，ANTLR 适合跨平台的项目。
- 利用选定工具的特性简化词法和语法分析器的开发。

(2) 编译器生成:

- 使用工具生成的编译器应该完全兼容 PL/0 语言的语法规则。
- 生成的编译器需要能够被轻松地集成到开发环境中。

性能要求:

- 利用工具提高编译器的开发效率，减少从编写到测试的周期。
- 生成的编译器应当具有高效的运行时间和合理的资源消耗。

可靠性要求：

- 使用编译工具生成的编译器需要能够稳定运行，在各种情况下均能提供准确的编译结果。

四、 编译算法实现

4.1 词法分析

词法分析是编译过程中的第一阶段，负责将输入的字符序列转换为符号（token）序列。本节将详细介绍 PL/0 编译器中词法分析器的设计与实现。

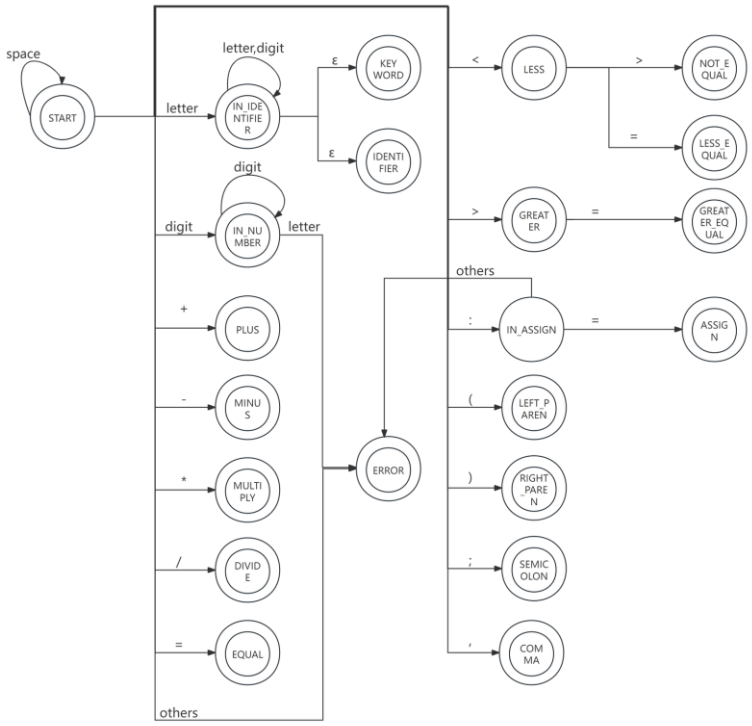
4.1.1 设计概述

总体上来说，我们使用 Thompson 算法根据词法规则构造 NFA，使用子集法将 NFA 转化成 DFA，最后用等价状态法 DFA 化简。在代码中以自动机的形式，利用 DFA 的状态转移，完成词法分析。词法分析器的主要功能包括：

- 识别关键字和标识符
- 识别整数和操作符
- 忽略空白字符
- 报告错误，并指出错误位置

4.1.2 状态设计

根据 3.1.1 中的词法规则，设计了如下图的由确定有限自动机（DFA）控制的词法分析器，该 DFA 精确地定义了在识别 PL/0 语言的词法元素时应当遵循的路径。

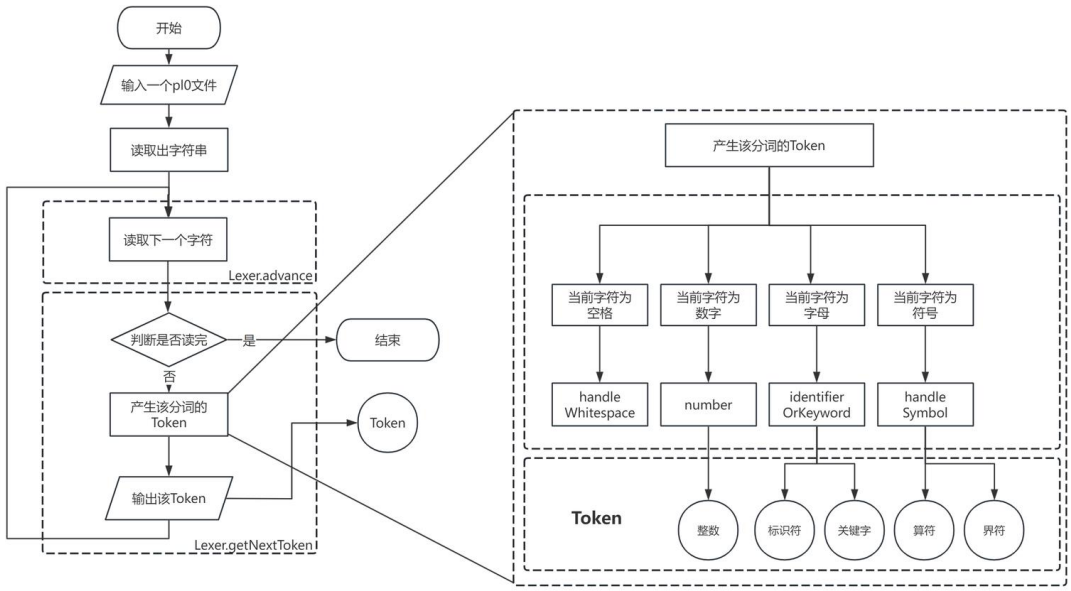


- **起始状态：**所有符号的识别始于 START 状态。该状态根据输入的字符类型，转移到不同的状态。例如，空格会使状态机保持在 START 状态，而字母会触发转移到 IN_IDENTIFIER 状态。
- **标识符和关键字：**当状态机处于 IN_IDENTIFIER 时，连续的字母或数字会保持该状态，直到遇到非字母数字字符。此时，状态机会判断已收集的字符串是关键字（KEYWORD）还是普通标识符（IDENTIFIER）。
- **数字：**从 START 状态接收到数字字符时，状态机转移到 IN_NUMBER 状态，并在此状态中继续累积数字，直至遇到非数字字符。
- **操作符和界符：**对于如+、-、*、/等单字符操作符，状态机直接从 START 转移到对应的状态（如 PLUS、MINUS 等）。而对于如:=、<>、<=、>=这样的多字符操作符，状态机会经历一系列状态（如 IN_ASSIGN、IN_LESS、IN_GREATER 等）以识别整个符号。
- **错误处理：**对于不符合任何合法转移条件的字符，状态机会转移到 ERROR 状态，该状态表示输入不符合 PL/0 语言的定义。
- **结束状态：**每个接受状态都对应着一个或一组特定的词法符号，如 ASSIGN、LESS 等。一旦状态机到达这些状态，就会生成一个相应的 Token，并重置到 START 状态以准备下一个符号的识别。

通过上图的 DFA，我们可以清晰地理解词法分析器如何一步步地从源代码文本中识别出词法符号。每个状态都设计得尽可能简洁，以确保分析过程既高效又可靠。

4.1.3 核心功能实现

为了便于后续部分使用词法分析的相关代码，在词法分析部分的 Lexer 类中提供了 Lexer.getNextToken()的接口用于每次从 Lexer.source 中读取一部分字符串并生成相应的 Token（即(token name, value)）。



- **字符处理：**advance()方法用于读取下一个字符，并更新行号和列号以便错误报告。
- **空白处理：**handleWhitespace()方法用于跳过空白字符，保持状态机回到 START 状态。
- **标识符和关键字处理：**identifierOrKeyword()方法用于从源代码中识别标识符或关键字，并生成相应的符号。
- **数字处理：**number()方法用于识别整数，并生成数值符号。
- **符号处理：**handleSymbol()方法用于识别操作符和界符，如+、−、*、/、:=等。

4.1.4 错误处理

分析器在遇到非法字符或者符号组合时会抛出 PL0Exception 异常，异常信息中包含错误的字符和位置，以便于调试和错误修正。目前有的是

- 输入非法字符；
- 输入″:″后并没有紧跟着″=″形成″:=″；
- 数字后面紧跟着字母，在词法规则下违规。

4.1.5 代码结构

代码主要由两个类构成：Token 和 Lexer。Token 类用于表示一个符号，包括符号类型(TokenType)和值(String)。Lexer 类负责整个词法分析过程，包括字符的读取、符号的生成以及错误的处理。

4.2 语法分析

语法分析是编译过程中的第二阶段，负责将词法分析器得到的 token 识别为语法表达式。由于本项目是一遍编译，在语法分析的过程中会同步生成中间代码，而不是生成语法树后统一生成中间代码。本部分只介绍语法分析部分。

4.2.1 设计概述

鉴于 PL0 语言大致符合 LL(1)文法，本语法分析器采用**递归下降分析法**的结构。

语法分析器的主要功能包括：

- 识别文法表达式
- 当发现语法错误时，报错

4.2.2 代码架构

本模块实现于 Parser 类。

Parser 类具有三个 private 属性值：

- token　当前符号
- lexer　词法分析器
- end_flag 指示是否发生复合语句嵌套，用于复合语句解析

语法分析方法的分割原则，是每个语法表达式对应一个函数：

- BeginParse 程序，启动语法分析
- ProgramHead 程序首部
- SubProgram 分程序
- ConstDeclaration 常量声明
- ConstDefinition 常量定义
- VarDeclaration 变量声明
- Statement 语句
- AssignmentStat 赋值语句
- ConditionalStat 条件语句

- LoopStat 循环语句
- CompoundStat 复合语句
- Expression 表达式
- Item 项
- Factor 因子
- Condition 条件

除了上述用于语法分析的函数，还有工具函数：

- getNextToken 调用词法分析器，获取下一个 token

4.2.3 错误处理

分析器在遇到非法文法时会抛出 PL0Exception 异常，异常信息中包含错误信息和位置，以便于调试和错误修正。具体逻辑是，在每个文法分析函数中，如果发现下一个 token 是非法的，就会报错。例如，在赋值语句分析函数 AssignmentStat 中，若赋值符号:=没有在正确的位置出现，就会报错：“赋值语句格式错误”，同时标明出错的行与列。

4.2.4 设计细节

原文法中，表达式和项的文法，由于两个候选式的首字符集相同，不严格符合 LL(1)文法，不易采用递归下降分析结构。因此，在经过一系列分析后，改编了原先的文法，由：

〈表达式〉 => [+|-]项|〈表达式〉〈加法运算符〉〈项〉

〈项〉 => 〈因子〉|〈项〉〈乘法运算符〉〈因子〉

改编为：

〈表达式〉 => [〈加法运算符〉] 项 { 〈加法运算符〉 项 }

〈项〉 => 〈因子〉 { 〈乘法运算符〉〈因子〉 }

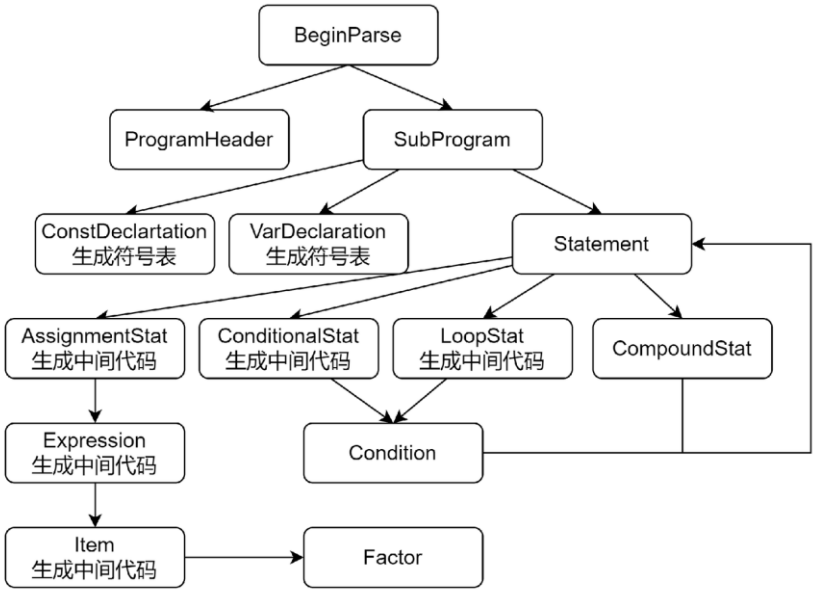
经改编后的文法，可以简单地使用循环来实现语法单元分析。

4.3 中间代码生成

中间代码生成是编译过程中的第三阶段，由于本项目是一遍编译，在语法分析的过程中会同步生成中间代码，本部分将介绍本项目如何在语法分析对应生成式的函数中生成中间代码

4.3.1 设计概述

中间代码生成的主要工作是根据语法分析生成的语法树来生成对应的代码。在本项目中，语法分析部分不显式生成语法树，而是利用语法分析过程中的函数调用过程构成的调用树充当语法树。



4.3.2 代码架构

本模块实现于 Parser 类的成员函数和 Parser 类成员函数实现所在的 Parser.cpp 中。语法分析使每个语法表达式对应一个函数，其中，需要生成中间代码的函数有：

- AssignmentStat 赋值语句
- ConditionalStat 条件语句
- LoopStat 循环语句
- Expression 表达式
- Item 项

除了上述用于会生成中间代码的函数，还有需要写入到变量表的函数：

- VarDeclaration 变量声明
- ConstDefinition 常量定义

此外，还定义了部分工具函数和工具变量用于辅助解析，工具变量如下：

- IdentitierTable 标识符表，用于存储变量和常量的名称
- TempTable 临时变量表，用于存储临时变量的名称
- tempCount 临时变量计数，用于生成名称不同的临时变量
- IR 中间代码数组，以四元元组方式存储中间代码以待输出

工具函数如下：

- emit 中间代码生成函数，用于根据参数生成中间代码写入 IR
- Parser::Output 输出函数，用于将符号表以 CSV 表格的形式输出，将中间代码以 TXT 文档的格式输出

4.3.3 错误处理

本项目采用命令行参数形式指定输入输出，格式如下：

Usage: compiler <InputSourceFile> -oIR <IRFile> -oIT <ITFile>

因此需要对命令行参数进行解析和错误处理：

- 当用户没有输入源代码文件名时，报错没有源代码输入文件；
- 当用户输入的参数错误时，提示用户如何正确输入参数；
- 当用户输入的参数后面没有紧跟对应文件名时，报错缺乏对应的文件名。

每次报错后均会跟随有使用说明和编译终止说明。

4.3.4 设计细节

本项目临时变量格式为“TempVar 数字”的格式，数字部分由 tempCount 进行计算，每次生成；临时变量时，程序会判断当前即将生成的临时变量名称是否已存在于标识符表或临时变量表，如果已经存在同名称的变量，则将 tempCount+1，生成一个新名称的临时变量，重复该过程直到生成的临时变量不与其他标识符和临时变量冲突为止。

五、 编译工具使用

5.1 编译算法实现

5.1.1 解决方案整体介绍

ANTLR（全称：ANother Tool for Language Recognition）是一种流行的语言识别工具，使用 Java 语言编写， 基于 LL(*)解析方式，使用自上而下的递归下降分析方法。 通过输入语法描述文件来自动构造自定义语言的词法分析器、语法分析器和树状分析器等各个模块。 ANTLR 使用上下无关文法描述语言。ANTLR 是基于用户提供的语法规则文件，自动生成相应词法/语法分析器的一个工具，并提供给用户后续改造、加工的接口。

我们采用 Jetbrains IDEA 的 ANTLR 插件，在.g4 文件中定义语法规则后，即可用此插件的 Preview 窗口，输入要分析的语言得到一颗语法树。同时，配置此插件的代码生成路径后，即可自动生成一套基于此语法的分析器代码。我们通过重写这部分代码，即可生成我们想要的中间代码格式。

5.1.2 工具客户化工作

通过老师给出的语法规则，我们可以直接将代码的语法和词法描述转换成满足.g4文法的词法定义以及语法定义。

词法定义：

```
// 词法规则
PROGRAM : 'PROGRAM';
BEGIN   : 'BEGIN';
END     : 'END';
CONST   : 'CONST';
VAR     : 'VAR';
WHILE   : 'WHILE';
DO      : 'DO';
IF      : 'IF';
THEN    : 'THEN';
// 标识符
ID      : LETTER (LETTER | DIGIT)* ;
// 整数
INT     : DIGIT+ ;
// 算符和界符
PLUS    : '+';
MINUS   : '-';
STAR    : '*';
SLASH   : '/';
ASSIGN  : ':=';
EQUAL   : '=';
NOTEQUAL: '<>';
GT      : '>';
GTEQ    : '>=';
LT      : '<';
LTEQ    : '<=';
LPAREN  : '(';
RPAREN  : ')';
SEMI    : ';';
COMMA   : ',';
```

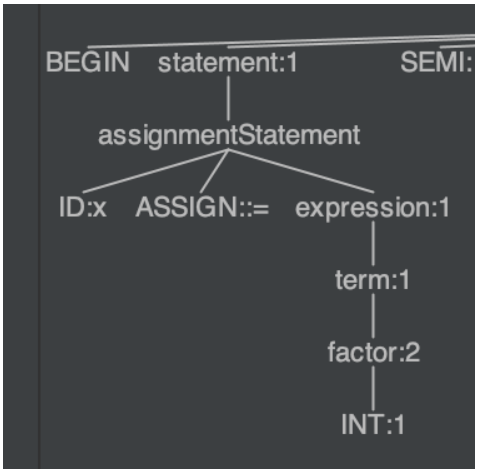
语法定义：

```
// 语法规则
// 程序入口
program         : programID programBody;  // programBody 表示分程序
programID   : PROGRAM ID;
// 程序主体
programBody      : constDeclaration? varDeclaration? statement;
// 前两项为可选项
// 常量声明
constDeclaration: CONST constDefinition (COMMA constDefinition)*
SEMI;
// 常量定义
constDefinition : ID ASSIGN INT;
// 变量声明
varDeclaration  : VAR ID (COMMA ID)* SEMI;

// 语句定义(含有递归)
statement       : assignmentStatement
                | conditionalStatement
                | loopStatement
                | compoundStatement
                | ; // 空语句

compoundStatement: BEGIN statement (SEMI statement)* END;
assignmentStatement: ID ASSIGN expression;
expression       : (PLUS | MINUS)? term
                | expression addOp term;
term             : factor
                | term mulOp factor;
factor           : ID
                | INT
                | LPAREN expression RPAREN;
addOp            : PLUS
                | MINUS;
mulOp            : STAR
                | SLASH;
conditionalStatement: IF condition THEN statement;
loopStatement    : WHILE condition DO statement;
condition        : expression relOp expression;

// 关系运算符
relOp            : EQUAL
```

语法树是中间代码的非线性表示，得到一棵语法树后，其实就相当于得到了中间代码的大体结构。在 ANTLR 中，我们可以通过对语法树的深度优先遍历来生成中间代码。在 ANTLR 中，遍历语法树有两种形式，第一种是访问者(Vistor)模式， 也即对每个节点进行操作并返回一个值；第二种是监听者模式，也即在进入节点和离开节点时触发事件。我们采用第一种模式，在深度优先遍历时，按照指定顺序进入子结点，并控制如何根据此结点信息和子结点信息生成中间代码。

比如对于赋值语句，孩子节点的信息就有标志符和表达式；最左子树是标识符，因此直接去获取变量的名称就可以知道这是哪个变量。最右子树是表达式，对于表达式需要进入这颗子树，来获取这颗子树的真正对应的地址的表示。之后将其拼接，即可得到三地址代码。

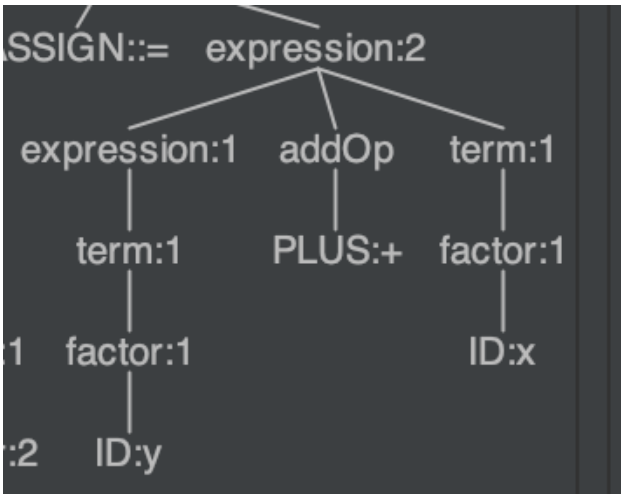
<赋值语句>→<标识符>:=<表达式>




```
@Override
    public String
visitAssignmentStatement(pl0Parser.AssignmentStatementContext
ctx) {
        // Handle assignment statement
        String var = ctx.ID().toString(); // 获取变量名
        String exprCode = visit(ctx.expression()); // 访问表达式并
获取其代码
        emitCode(var + " = " + exprCode); // 生成赋值语句的三地址代码
        return null;
    }
```

对于表达式语句，则要看它有几个孩子，如果只有一个孩子，直接返回孩子的地址

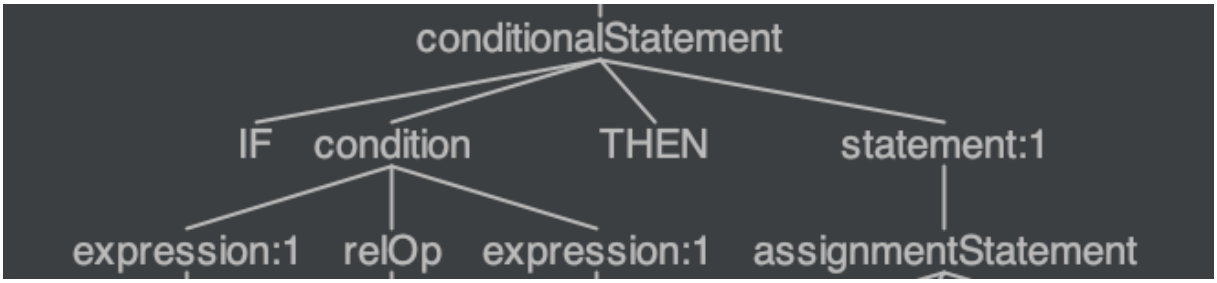
代码即可。如果有多个孩子，则要依次去访问左右侧运算符。




```
@Override
public String visitExpression(pl0Parser.ExpressionContext ctx) {
    if (ctx.getChildCount() == 1) {
        return visit(ctx.getChild(0)); // 直接返回单个子项的代码
    } else {
        String left = visit(ctx.getChild(0)); // 访问左侧子表达式
        String op = ctx.getChild(1).getText(); // 获取操作符
        String right = visit(ctx.getChild(2)); // 访问右侧子表达式
        String temp = newTemp(); // 生成新的临时变量
        emitCode(temp + " = " + left + " " + op + " " + right);
// 生成三地址代码
        return temp;
    }
}
```

这两个例子代表了基本的不含循环的遍历实现。

对于循环语句和条件语句，则需要处理回填位。在这里以条件语句为例，研究在 ANTLR 中如何实现条件的检查。为了能在后续进行回填，我们需要生成两个标签，记录下条件语句当前的行数，并在条件语句失败后，直接跳到之后的行数。对于这个前后的关系，我们直接在遍历时进行控制，先给代表条件判断正确的标签赋值，进入 statement 去产生接下来的代码。在退出 statement 之后，代码行数已经产生了变化，此时再给代表条件判断错误的标签进行赋值：




```
@Override
public String
visitConditionalStatement(pl0Parser.ConditionalStatementContext
ctx) {
    int labelRight = newLineLabel(); // 用于条件成立时的跳转
    int labelWrong = newLineLabel(); // 用于条件不成立时的跳转

    // 生成条件检查代码
    String conditionCode = visit(ctx.condition());
    emitCode("if " + conditionCode + " goto $" + labelRight);

    // 生成跳转到 else 部分的代码
    emitCode("goto $" + labelWrong);

    // 给第一个标签赋值
    labelValues.set(labelRight, codeCount -1);  // else 部分，emit
Code 之后会多一个，本身有++，所以要减 1 才能表示下一个
    visit(ctx.statement());

    // 给第二个标签赋值
    labelValues.set(labelWrong, codeCount -1);
    return null;
```

六、 测试与验证

6.1 任务 1：正常代码测试

下面的代码包含全部的文法：

```
PROGRAM add
CONST a:=1 ,b:=2;
VAR x,y;
BEGIN
    x:=a+b*a*b+b;
    y:=2;
    WHILE y-(a+1)+1<(x+1)*y*x DO x:=(x+1)*y/x;;
    IF y-(a+1)+1>(x+1)*y*x THEN y:=y-(a+1)+1;
    BEGIN
    END;
    y:=y+x
END
```

编译上述代码，正常编译，得到目标代码文件：

```
100 (*,b,a,TempVar1)
101 (*,TempVar1,b,TempVar2)
102 (+,a,TempVar2,TempVar0)
103 (+,TempVar0,b,TempVar3)
104 (:=,TempVar3,_,x)
105 (:=,2,_,y)
106 (+,a,1,TempVar5)
107 (-,y,TempVar5,TempVar4)
108 (+,TempVar4,1,TempVar6)
109 (+,x,1,TempVar7)
110 (*,TempVar7,y,TempVar8)
111 (*,TempVar8,x,TempVar9)
112 (j<,TempVar6,TempVar9,114)
113 (j,_,_,119)
114 (+,x,1,TempVar10)
115 (*,TempVar10,y,TempVar11)
116 (/,TempVar11,x,TempVar12)
117 (:=,TempVar12,_,x)
118 (j,_,_,106)
119 (+,a,1,TempVar14)
120 (-,y,TempVar14,TempVar13)
121 (+,TempVar13,1,TempVar15)
122 (+,x,1,TempVar16)
123 (*,TempVar16,y,TempVar17)
124 (*,TempVar17,x,TempVar18)
125 (j>,TempVar15,TempVar18,127)
126 (j,_,_,131)
127 (+,a,1,TempVar20)
128 (-,y,TempVar20,TempVar19)
129 (+,TempVar19,1,TempVar21)
130 (:=,TempVar21,_,y)
131 (+,y,x,TempVar22)
132 (:=,TempVar22,_,y)
```

观察生成的目标代码，表达式处理正常：

```
x:=(x+1)*y/x
```

翻译为：

```
114 (+,x,1,TempVar10)
115 (*,TempVar10,y,TempVar11)
116 (/,TempVar11,x,TempVar12)
```

空语句处理正常，未翻译；

循环语句处理正常：

```
WHILE y-(a+1)+1<(x+1)*y*x DO x:=(x+1)*y/x;
```

翻译为：

```
106 (+,a,1,TempVar5)
107 (-,y,TempVar5,TempVar4)
108 (+,TempVar4,1,TempVar6)
109 (+,x,1,TempVar7)
110 (*,TempVar7,y,TempVar8)
111 (*,TempVar8,x,TempVar9)
112 (j<,TempVar6,TempVar9,114)
113 (j,_,_,119)
114 (+,x,1,TempVar10)
115 (*,TempVar10,y,TempVar11)
116 (/,TempVar11,x,TempVar12)
117 (:=,TempVar12,_,x)
118 (j,_,_,106)
```

条件语句处理正常：

```
IF y-(a+1)+1>(x+1)*y*x THEN y:=y-(a+1)+1;
```

翻译为：

```
119 (+,a,1,TempVar14)
120 (-,y,TempVar14,TempVar13)
121 (+,TempVar13,1,TempVar15)
122 (+,x,1,TempVar16)
123 (*,TempVar16,y,TempVar17)
124 (*,TempVar17,x,TempVar18)
125 (j>,TempVar15,TempVar18,127)
126 (j,_,_,131)
127 (+,a,1,TempVar20)
128 (-,y,TempVar20,TempVar19)
129 (+,TempVar19,1,TempVar21)
130 (:=,TempVar21,_,y)
```

符号表处理正常：

```
CONST a:=1 ,b:=2;
VAR x,y;
```

符号表为：

```
Name,Type
a,CONST
b,CONST
x,VAR
y,VAR
```

6.2 任务 1：错误处理测试

6.2.1 非法符号

错误代码：

```
PROGRAM add
CONST a:=1 ,b:=2;
VAR x,y;
BEGIN
    y:=0 || 1;
END
```

编译结果：

```
非法字符'|' at line 5 at col 11
```

6.2.2 := 之间有空格

错误代码：

```
PROGRAM add
CONST a:=1 ,b:=2;
VAR x,y;
BEGIN
    y: =0;
END
```

编译结果：

```
非法输入 ':',你要输入 ':=' at line 5 at col 7
```

6.2.3 使用未声明变量

错误代码：

```
PROGRAM add
CONST a:=1 ,b:=2;
VAR x,y;
BEGIN
    y=0;
END
```

编译结果：

```
赋值语句格式错误 at line 5 at col 7
```

6.2.4 其他错误

关于错误处理，我们还实现了以下详尽的的错误处理，篇幅有限，不再附上示例：

1. 非法的关系运算符
2. 条件语句格式错误
3. 循环语句格式错误
4. 缺少右括号
5. 缺少因子或因子格式错误
6. 符号不可被识别
7. 赋值语句格式错误
8. 复合语句格式错误
9. 变量说明格式错误
10. 常量说明格式错误，缺少分号
11. 常量定义格式错误
12. 无法识别语句种类
13. 程序首部缺少关键字 PROGRAM
14. 程序首部缺少标识符

6.3 任务 2：正常代码测试

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

输出：

（左为中间代码，右为符号表）

main.java | mediaCode.txt | symbol.txt

```
0: x = 1
1: y = 2
2: if x < 5 goto 4
3: goto 7
4: t0 = x + 1
5: x = t0
6: goto 2
7: t1 = x + y
8: t2 = y * t1
9: t3 = t2 - x
10: y = t3
11: if y > 0 goto 13
12: goto 15
13: t4 = y - 1
14: y = t4
15: t5 = y + x
16: y = t5
```

```
x var
y var
```

6.4 任务 2：错误处理测试

6.4.1 未声明变量测试

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

对应错误处理处：

```
if (!varRecord.contains(varOrConstName) && !constRecord.contains(varOrCo
    System.err.println("Variable '" + varOrConstName + "' not declared")
    return null;
}
if(constRecord.contains(varOrConstName))
{
    System.err.println("Const Variable '" + varOrConstName + "' can not
    return null;
}
```

输出

```
 IDEA Ultimate.app/Contents/bin -Dfile.encoding=UTF-8 -classpath /U
Variable 'z' not declared

Process finished with exit code 0
```

6.4.2 因子为空

注：代码中所有因为为空导致语法的情况，都打印为... Syntax Error，...为遍历时出错的节点的类型。

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

对应错误处理处：

```
blic String visitFactor(pl0Parser.FactorContext ctx)

  if (ctx.ID() != null) {
      // 标识符
      return ctx.ID().getText();
  } else if (ctx.INT() != null) {
      // 整数
      return ctx.INT().getText();
  } else if (ctx.LPAREN() != null) {
      // 括号内的表达式
      return visit(ctx.expression());
  }
  else {
      System.err.println("Factor Syntax Error");
  }
  return null;
```

输出

```
Factor Syntax Error

Process finished with exit code 0
```

七、 总结与反思

编译器中，语法分析程序是核心，一般的架构都是在语法分析器中，通过调用词法分析模块和中间代码生成模块，来实现一遍编译的功能。本项目在整体流程上，先编写词法分析器，再编写语法分析器，最后在语法分析器原始代码的基础上嵌入中间代码生成流程。这个流程有一定的合理性，一步一步地搭建整个编译器，方便测试。但如果可以先编写好词法分析模块和中间代码模块，最后在编写语法分析器时利用充分设计的两模块 API，将语法分析模块和中间代码模块完全解耦合，或许可以大大提高编译器的可拓展性、代码的可读性。

参考文献

1. 程序设计语言 编译原理（第 3 版）陈火旺等编著(387 pages), 国防工业出版社
2. Compilers Principles Techniques and Tools(2nd Edition); Alfred V. Aho 等编著
3. https://canvas.tongji.edu.cn/courses/77943/files
4. https://blog.csdn.net/qq_44850725/article/details/114217589
5. https://courses.cs.washington.edu/courses/cse401/02sp/pl0/proj.html
6. https://www.bilibili.com/video/BV1yo4y197uk/?spm_id_from=333.337.search-card.all.click&vd_source=54848bbaacc95a6670b0f8ac0228b019
7. https://github.com/topics/pl0
8. https://www.cnblogs.com/ZJUT-jiangnan/archive/2013/12/27/3494501.html
9. https://canvas.tongji.edu.cn/files/3132089
10. https://www.antlr.org/