同济大学

TONGJI UNIVERSITY

基于手工编写与 ANTLR 的 PL/0 编译器实现

——编译原理项目文档

学　　院　软件学院

专　　业　软件工程

学生姓名　吴可非　孙亦菲　陈雨彤　高嘉颖

学　　号　2150271 2152085 2153677 2154308

指导教师　高珍

日　　期　2023 年 12 月 31 日

代码仓库　https://github.com/wukef2425/pl0-compiler-handwritten-and-antlr.git

目 录

目 录

任务 1: 编译算法实现

1 解决方案的整体介绍

需求分析：说明程序任务输入及其范围，输出形式，程序功能，测试数据（正确，错误数据）

1.1 项目需求

选择一种高级程序设计语言，选择课内学习的一种词法分析器构造算法、语法分析器构造算法和中间代码生成算法实现一个小语言 PL/0 的简单编译器。

- 要求编译器的输入为符合 PL/0 语言源程序。输出为中间代码（三地址代码）表示的程序。
- 要求编译器是个一遍的编译程序，词法分析程序作为子程序，需要的时候被语法分析程序调用。
- 要求使用语法制导的翻译技术，在语法分析的同时生成中间代码，并保存到文件中。

Candidate Algorithms:

- a. 词法分析：Thompson 算法、子集法、等价状态法等
- b. 语法分析：递归下降分析法、预测分析程序、LR 分析法等
- c. 中间代码生成：属性文法、翻译子程序等

1.2 程序任务输入及其范围

输入要满足 PL/0 语言的语言规则

1.2.1 词法规则

- 关键字：PROGRAM、BEGIN、END、CONST、VAR、WHILE、DO、IF、THEN。
- 标识符：以字母开头的、由字母和数字组成的字符串。
- 整数：数字开头的数字串。
- 算符、界符：+、-、*、/、:=、=、<>、>、>=、<、<=、(、)、;、,

1.2.2 语法规则

- <程序> →<程序首部 <分程序>
- <程序首部> →PROGRAM <标识符>
- <分程序> →[<常量说明>][<变量说明>]<语句>
- <常量说明> →CONST <常量定义>{，<常量定义>}
- <常量定义> →<标识符>=<无符号整数>
- <无符号整数> →<数字>{，<数字>}
- <变量说明> →VAR<标识符>{，<标识符>}
- <标识符> →<字母>{<字母> | <数字>}
- <复合语句> →BEGIN <语句>{；<语句>} END
- <语句> →<赋值语句> | <条件语句 >| <循环语句> | <复合语句> | <空语句>
- <赋值语句> →<标识符>:=<表达式>
- <表达式> →[+|-] 项 | <表达式> <加法运算符> <项>
- <项> →<因子> | <项><乘法运算符> <因子>
- <因子> →<标识符> |<常量> | (<表达式>)
- <加法运算符> →+ | -
- <乘法运算符> →* | /
- <条件语句> →IF <条件> THEN <语句>
- <循环语句> →WHILE <条件> DO <语句>
- <条件> →<表达式> <关系运算符> <表达式>
- <关系运算符> →= | <> | < | <= | > | >=
- <字母> →a | b | . . . | x | y | z
- <数字> →0 | 1 | . . . | 8 | 9

1.2.3 PL/0 源程序实例

```
1 PROGRAM add
2
```

```
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
```

1.2.4 输入方式

控制台会提示您输入您的代码路径，代码需存储在 txt 格式文件中。输入之后，回车即可。

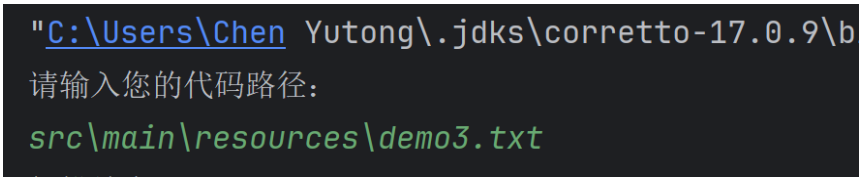


图 1.1 输入方式截图

1.3 输出形式

1.3.1 中间代码输出文件

扫描结束后，控制台会输出提示，提示您中间代码已输出到的文件路径。打开文件，即可看到
输出形式是基于代码地址的三地址代码形式表示。基本格式如下：

<代码地址>　<结果变量> = <操作数> <操作符> <操作数>

三地址代码的基本形式包括赋值语句和算术运算，例如：

1. 赋值语句：`x := y`
2. 二元运算：`x = y op z`
3. 条件跳转：`IF x > y GOTO L`

在这里，`x`、`y`、`z` 可以是变量或常数，而 `op` 可以是各种二元或一元运算符，如加法、减法、乘法、除法等。条件跳转语句用于实现条件控制结构，其中 `L` 是目标标签。

另外，我们不仅会将中间代码输出到文件当中，同样也会打印在控制台，便于使用者进行更加及时和直观的查看。

1.3.2 符号表输出控制台

我们会将符号表输出至控制台。输出形式是一个简单的符号表，用于记录程序中定义的各种符号的信息。每一行代表一个符号的条目，其中包含以下信息：

name（名称）	type（类型）	isAssigned（是否已赋值）
maxcount	constsy	true
a	varsy	true
b	varsy	true

通过这个符号表，可以查看程序中声明的符号以及它们的类型和赋值状态。这对于编译器的各个阶段，特别是代码生成和优化阶段，都是重要的信息。

1.3.3 语法错误输出控制台

语法错误的输出形式提供了有关源代码中错误的信息，输出形式包括：错误发生的行数、错误类型和相关的错误描述。

这种输出形式的有利于帮助使用者定位并修复源代码中的错误，以便成功编译或解释程序。

1.4 程序功能

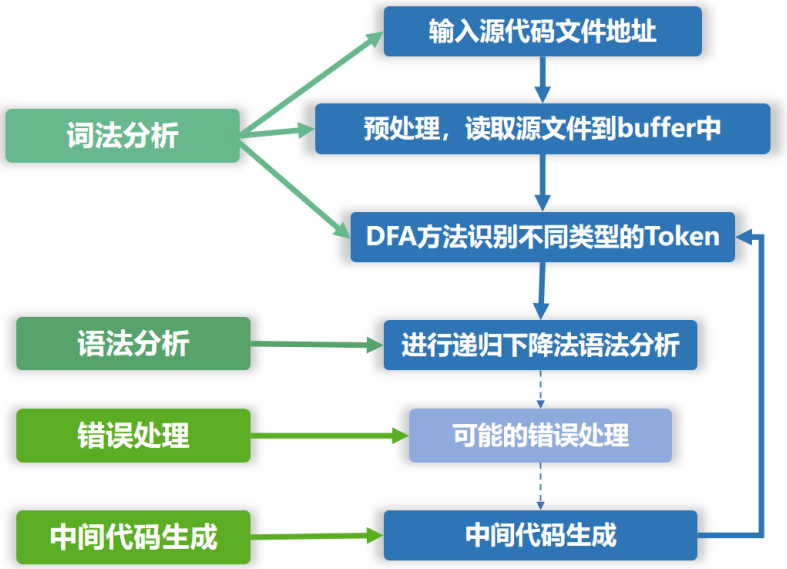


图 1.2 主程序流程图

基于图 1.2 所示，主程序的大致流程概括如下：

1. 由词法分析器（LexAnalysis）读取源代码文件。
2. 词法分析器逐行读取代码文件，将内容存入字符数组 buffer。
3. 使用 DFA 算法进行词法分析，读取 Token，得到 Token 的类型。
4. 使用递归下降法进行语法分析，处理读取的 Token, 并进行可能的错误处理。
5. 生成三地址代码形式的中间代码，再进行读取 Token 的操作，循环工作，直到文件的结束。

1.4.1 词法分析部分

在词法分析部分，功能包括：逐行读取源代码文件，将内容存入字符数组；使用 DFA 算法，逐字符扫描源代码，识别并生成 Token，得到 Token 的类型。具体包括代码预处理、获取下一个符号、获取当前字符、连接字符到当前标记字符串、判断输入字符串是否为关键字或标点符号等功能。此外，还提供了处理错误、获取行号、获取当前符号类型等功能。

1.4.2 语法分析部分

在语法分析部分，使用递归下降法进行语法分析。

在这部分，我们根据词法分析得到的 Token 序列，不断的从 Token 序列中获取第一个 Token，根据语法规则进行递归下降分析，构建抽象语法树。并处理可能的语法错误，进行错误恢复。直到整个源代码文件被完全解析，从而实现了代码的语法分析。

1.4.3 中间代码生成部分

中间代码生成部分负责将经过语法分析得到的抽象语法树转化为三地址代码形式的中间代码。

在这部分，我们将从语法分析得到的抽象语法树开始遍历。针对每个语法树节点，生成相应的三地址代码。递归处理子节点，确保整棵语法树都被遍历。中间代码中可能包含临时变量、标签等，进行了适当的命名和管理。最终得到整个源代码的中间代码表示。

2 概要设计

2.1 数据类型的定义

2.1.1 错误处理部分

- *EnumErrors* 类
  - 枚举值: 该类用于表示符号表中的每一行，包括标识符的名称和类型。
    - illegalChar: 一个非法的字符串。
    - assignEqual：赋值符应该有等号
    - noProgram：没有 PROGRAM 关键字
    - noProgramIdent：PROGRAM 没有标识符
    - symbolTableOverflow：符号表溢出
    - identiOverDefine：标识符重定义
    - undefinedIdent：标识符未定义
    - illegalConstVal：非法的常量值
    - noEqlforConst：没有等号为常量赋值
    - noConstDeclaration：没有常量定义式
    - noCommaSeperateConst：常量声明间没有逗号分隔
    - noCommaSeperateVar：变量声明间没有逗号分隔
    - noVarDeclaration：没有变量定义式
    - constAssigenment：常量不能被赋值
    - noBecomes：缺少赋值符号
    - noRparent：缺少右括号
    - illegalRelationalOperator：非法的关系运算符
    - noThen：IF 语句缺少 THEN
    - noDo：WHILE 语句缺少 DO
    - exInnerFault：表达式内部有误
    - noEnd：复合语句缺少 END
  - 类成员变量：
    - private final String tip: 错误提示信息，用于描述具体的错误类型。
    - private static int num: 记录错误的数量，用于统计整个编译过程中出现的错误数量。
  - 类成员函数：
    - EnumErrors(String tip): 构造函数，接受错误提示信息，初始化错误类型。
    - public static void error(EnumErrors error): 静态方法，用于输出错误信息，包含错误类型和所在行号。
    - public static int getNum(): 静态方法，用于获取发生的错误数量。
  - 类功能概述:
    EnumErrors 枚举类型用于定义编译过程中可能出现的错误类型，并提供了静态方法 ‘error‘ 用于输出错误信息。错误信息包括错误类型和所在的行号，通过调用 ‘error‘ 方法，可以在发现错误时将错误信息输出到控制台。同时，‘getNum‘ 方法用于获取整个编译过程中发生的错误数量。枚举类型的设计使得错误处理部分更为清晰，易于扩展和维护。

2.1.2 词法分析部分

- *LexAnalysis* 类
  - 类成员变量:
    - `private KeywordTable keywordTable`: 用于存储字符表的实例，提供对字符表的访问。
    - `private char ch`: 当前字符，用于存储当前从源代码中读取的字符。
    - `private EnumChar sy`: 当前符号，用于存储当前字符的词法单元类型。
    - `private String strToken`: 当前标记字符串，用于存储正在识别的标识符或常量。
    - `private String filename`: 文件名，存储源代码文件的名称。
    - `private char[] buffer`: 字符缓冲区，存储从源代码文件中读取的字符内容。
    - `private int searchPtr`: 搜索指针，用于追踪在字符缓冲区中的当前位置。
    - `private static int line`: 当前行号，记录当前字符所在的行数。
    - `static int isNewLine`: 判断是否是新的一行的标志，用于辅助行号的计算。

- 类成员函数:
  - `public LexAnalysis(String _filename)`: 构造函数，接受源代码文件名作为参数，初始化符号表和常量表，并设置文件名。
  - `public void preManage()`: 预处理函数，读取源文件内容到字符数组 buffer 中，包括换行符。
  - `public String nextToken()`: 获取下一个符号的主函数，根据当前字符的类型识别标识符、关键字、整数等。
  - `public char getChar()`: 获取当前字符的函数，根据搜索指针判断是否到达文件尾。
  - `public void getBC()`: 去除空格符的函数，用于跳过空格、制表符和换行符。
  - `public void concat()`: 连接字符到当前标记字符串的函数。
  - `public boolean isLetter()`: 判断当前字符是否为字母的函数。
  - `public boolean isDigit()`: 判断当前字符是否为数字的函数。
  - `public String getStrToken()`: 获取当前标记字符串的函数。
  - `public static int getLine()`: 获取当前行号的函数，考虑换行符的影响。
  - `public EnumChar getType()`: 获取当前符号的类型的函数。
  - `public boolean isAllUpperCase(String str)`: 判断字符串中的字符是否全都是大写字母的函数。
  - `public boolean isAllLowerCase(String str)`: 判断字符串中的字符只有小写字母和数字的函数。

- *KeywordTable* 类
  - 类成员变量:
    - `private HashMap keyWord`: 存储关键字及其对应的词法单元类型的哈希表。
  - 类成员函数:
    - `public KeywordTable()`: 构造函数，初始化关键字表，为关键字表添加相应的映射关系。

```
1         public CharTable(){
2         keyWord.put("PROGRAM",EnumChar.procsy);
3         keyWord.put("BEGIN",EnumChar.beginsy);
4         keyWord.put("END",EnumChar.endsy );
5         keyWord.put("IF",EnumChar.ifsy );
6         keyWord.put("THEN",EnumChar.thensy);
7         keyWord.put("CONST",EnumChar.constsy );
8         keyWord.put("VAR",EnumChar.varsy );
9         keyWord.put("DO",EnumChar.dosy );
10        keyWord.put("WHILE",EnumChar.whilesy );
11    }
```

    - `public EnumChar isKeyWord(String key)`: 判断输入的字符串是否是关键字，如果是返回对应的词法单元类型，否则返回 `EnumChar.nul`。
  - 类功能概述:
    KeywordTable 类主要用于存储关键字表。表的初始化发生在构造函数中，通过哈希表的形式存储了关键字的对应关系。类提供了判断输入字符串是否为关键字的方法，并返回相应的词法单元类型。

- *Enumchar* 枚举类
  EnumChar 枚举类型定义了编译器中可能用到的各种词法单元类型，包括关键字、标识符、整数、运算符、括号、标点符号等。以下是枚举值和对应的词法单元类型:
  - 枚举值:
    - nul: 用于表示空类型或错误类型。
    - procsy: 表示关键字 "PROGRAM"。
    - beginsy: 表示关键字 "BEGIN"。
    - endsy: 表示关键字 "END"。
    - constsy: 表示关键字 "CONST"。
    - varsy: 表示关键字 "VAR"。

    - whilesy: 表示关键字 "WHILE"。
    - dosy: 表示关键字 "DO"。
    - ifsy: 表示关键字 "IF"。
    - thensy: 表示关键字 "THEN"。
    - ident: 表示标识符类型。
    - intcon: 表示整数类型。
    - minussy: 表示减号运算符。
    - plussy: 表示加号运算符。
    - multisy: 表示乘号运算符。
    - divsy: 表示除号运算符。
    - becomes: 表示赋值运算符。
    - eql: 表示等于运算符。
    - neq: 表示不等于运算符。
    - gtr: 表示大于运算符。
    - geq: 表示大于等于运算符。
    - lss: 表示小于运算符。
    - leq: 表示小于等于运算符。
    - lparent: 表示左括号。
    - rparent: 表示右括号。
    - comma: 表示逗号。
    - colon: 表示冒号。
    - semicolon: 表示分号。
  - 类功能概述：
    EnumChar 枚举类型用于统一表示编译器中的各类词法单元。每个枚举值对应一个特定的词法单元类型，包括关键字、标识符、整数、运算符、括号、标点符号等。在编写编译器的词法分析部分时，可以使用 EnumChar 枚举类型来明确和规范识别的词法单元类型，便于代码的可读性和维护性。

2.1.3 语法分析部分

- *SymbolTableRow* 类
  - 功能: 该类用于表示符号表中的每一行，包括标识符的名称和类型、有无被赋值。
  - 类成员变量：
    - private String name：标识符的名称
  - 类成员函数：
    - SymbolTableRow(String name, EnumChar type)：带参构造函数，用于初始化名称和类型。被赋值属性初始化为假。
    - getName()：获取标识符的名称
    - setName(String name)：设置标识符的名称
    - getType()：获取标识符的类型
    - setType(EnumChar type)：设置标识符的类型
    - isAssigned()：返回是否被赋值。
    - setIsAssigned()：将这一行的标识符设置为已赋值。
    - print()：打印标识符的名称和类型、有无被赋值。
- *SymbolTable* 类
  - 功能：该类用于表示符号表，包括符号表的行数、每一行的内容以及相关操作。
  - 类成员变量：
    - private int rowMax：最大表长
    - private SymbolTableRow[] table：符号表的数组，存放 SymbolTableRow 对象
    - private int tablePtr：指向符号表中已经填入值最后一项
  - 类成员函数：
    - SymbolTable()：构造函数，初始化符号表数组
    - isFull()：判断符号表是否已满
    - enterTable(String name, EnumChar type)：向符号表中添加一行数据
    - getTablePtr()：获取符号表指针

    - getRow(int i)：获取指定行的符号表内容
    - checkExistence(String name)：检查指定名称的标识符是否存在于符号表中
    - printTable()：打印符号表的内容，包括名称和类型、有无被赋值。
  - *Parser* 类
    - 功能：该类是一个编译器语法分析器，用于对输入的代码进行语法分析，生成符号表和三地址代码。
    - 类成员变量：
      - LexAnalysis lex：用于词法分析的对象
      - SymbolTable STable：符号表对象，用于存储标识符和其类型
      - String filename：保存文件名
      - int tempId：临时变量的编号
      - ThreeAddressCodeGen codeGen：用于生成和存放三地址代码
    - 类成员函数：
      - analysis()：进行语法分析，生成三地址代码和符号表
      - ParserAnalysis()：进行语法分析，返回是否有错误
      - procedure()：处理程序的头部和子程序
      - header()：处理程序头部
      - subprocedure()：处理子程序，包括常量声明、变量声明和语句部分
      - constDeclare()：处理常量声明
      - constDefinition()：处理常量定义
      - varDeclare()：处理变量声明
      - statement()：判断并处理所有类型语句
      - statementPart()：处理语句部分
      - compoundStatement()：处理复合语句 BEGIN/END 间语句部分
      - assignmentStatement()：处理赋值语句
      - ifStatement()：处理条件语句
      - whileStatement()：处理循环语句
      - expression()：处理表达式
      - condition()：处理条件
      - relationalOperator()：处理关系运算符
      - term()：处理项
      - factor()：处理因子
      - enterTable()：处理符号表的添加、重定义、溢出处理
      - locateTableWithChecking()：在符号表中查找标识名，并检查是否存在错误。

2.1.4 中间代码生成部分

- *ThreeAdressCodeGen* 类
  - 功能：该类用于生成和存放三地址代码。
  - 类成员变量：
    - ArrayList<String> code：存放三地址代码的数组列表
    - int beginAddId：起始地址编号
    - int addrId：当前地址编号
  - 类成员函数：
    - emit(String s)：向三地址代码中添加一条指令，并返回地址编号
    - getCurrentAddrId()：获取当前地址编号
    - nextAddr()：生成下一个地址编号
    - setAddrCode(int id, String s)：设置特定地址的指令内容
    - printAllToFile()：打印所有的三地址代码及其地址编号, 并输出到相应文件。

2.2 模块间的调用关系

1. 词法分析模块 (LexAnalysis.java)：这部分负责读取源代码缓存到 buffer 中，并将其分解为 Token，使用的是确定有限自动机（DFA）来识别 Token 的模式，
2. 语法分析器 (Parser.java)：进行一遍扫描的主体，调用词法分析模块的方法，接收其生成的 Token，使用递归下降方法进行过语法分析；调用中间代码生成模块的方法生成中间代码；还会使用错误处理组件来打印在语法分析过程中遇到的错误。

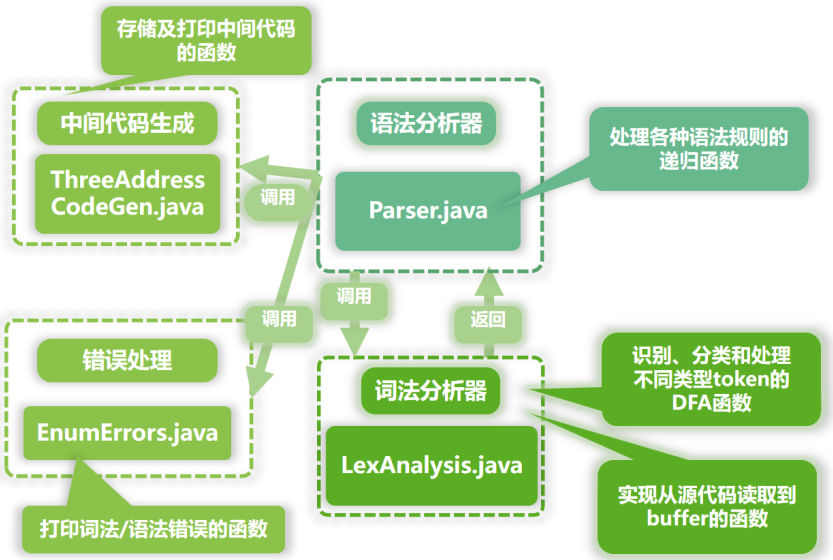


图 2.1 主模块间的相互调用关系图

3. 错误处理模块：这是一个枚举类型，列举所有预定义的编译错误。
4. 中间代码生成模块：这部分将语法分析转换为中间代码，通常这种中间代码是一种三地址代码形式，它是一种更接近于机器码但仍然保持一定层次的抽象。

3 详细设计

本部分着重讲解程序实现过程中的若干重要函数。

3.1 词法分析部分

nextToken() 函数是词法分析程序中的关键部分，用于识别标识符、关键字、整数、运算符等等。以下是对该函数的详细分析：

1. **`isNewLine = 0;` 和 `strToken = "";`**
   - 将 `isNewLine` 标志重置为 0，表示当前不是新的一行。
   - 清空当前标记字符串 `strToken`。
2. **`getBC();`**
   - 调用 `getBC()` 函数，去除空格符，将搜索指针移到非空格字符的位置。
3. **`nextToken();`**
   - 对下一个 Token 使用 DFA 方法分析所属类型，总结词法分析 DFA 算法过程如图 3.1 所示。
   - 其中，对具体 Keyword 所属类别的分析，调用了静态存储关键字的类 KeywordTable 中 isKeyWord 的方法来赋予具体类别。
4. **返回标记字符串 `strToken`。**

3.2 语法分析部分

3.2.1 读取文件与错误处理

在主类 main.java 中读入程序文件。实例化一个分析对象调用 analysis() 函数接口开始编译。

初始化并读入一个错误提示信息枚举类，提供一个 error 函数接口，配合词法分析器的获取行 getLine() 函数，在语法分析器分析过程中条件判断错误类型并调用，实现错误定位提示。

3.2.2 程序处理

如图 3.2 所示。

1. 抽象处理: 别调用 header() 函数和 subprocedure() 函数对过程的头部和子过程进行分开解析，识别头部后移进。

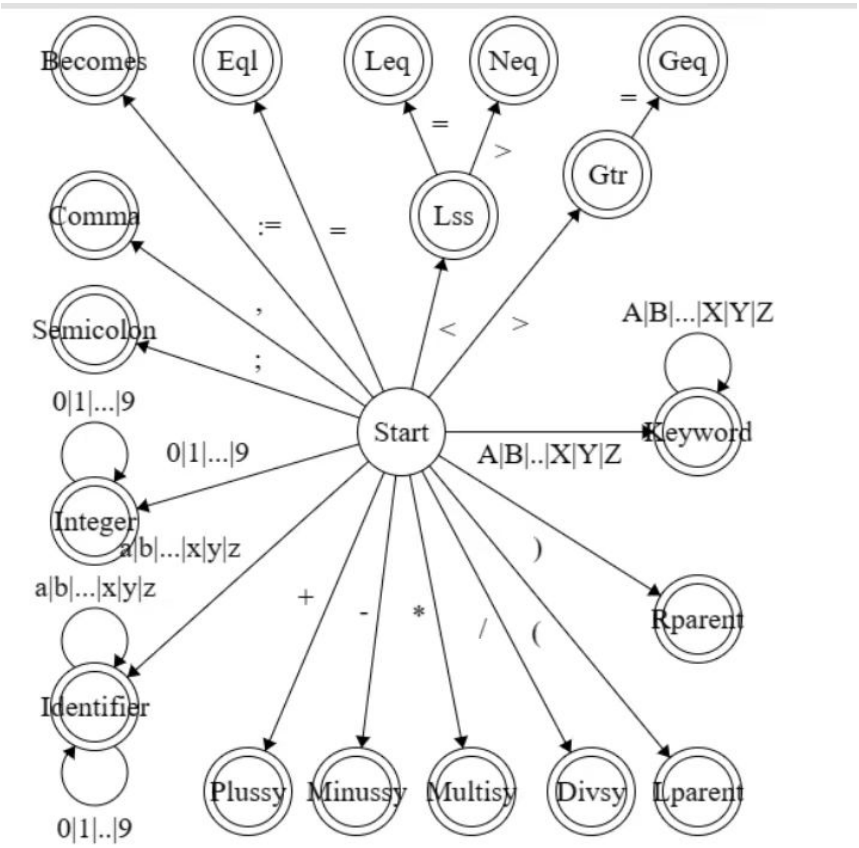


图 3.1 词法分析 DFA 算法过程

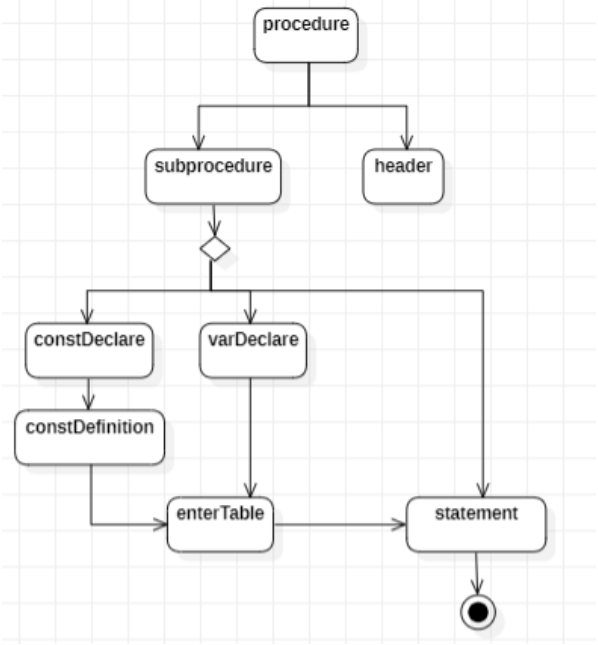


图 3.2 程序处理流程图

2. 程序头处理: 检查下一个词法单元是否为 procsy，如果不是，则报错。获取下一个词法单元作为程序标识符，如果不是标识符，则报错。
3. 子程序处理: 如果下一个词法单元是 constsy，则调用 constDeclare() 解析常量声明。如果下一个词法单元是 varsy，则调用 varDeclare() 解析变量声明。调用 statementPart() 解析语句部分。
    (a) 常量处理:constDefinition() 函数解析常量定义。循环，直到遇到分号结束。constDefinition 函数获取标识符和赋值符号。检查下一个词法单元是否为等号，如果是，则获取无符号整数作为常量值，并在符号表中记录。如果不是等号，则报错。
    (b) 变量处理: 获取标识符并在符号表中记录。循环，直到遇到分号结束。

3.2.3 语句处理

如图 3.3 所示。

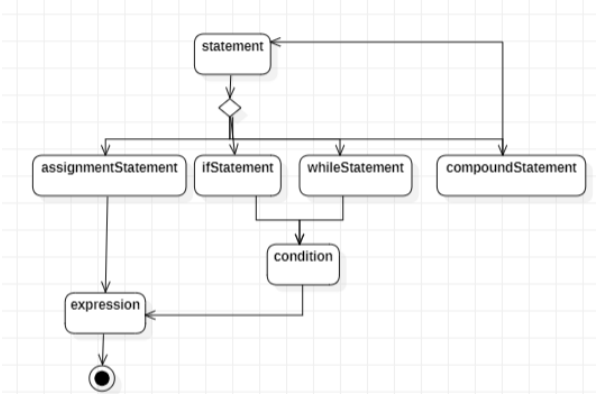


图 3.3 语句处理流程图

1. 复合语句处理: 在复合语句处理中循环逐句处理并在读到分号时移进，直到遇到关键字 END 移进并跳出循环，若复合语句内所有基础语句处理完毕未遇 END 则报错。
2. 基本语句处理: 在基本语句处理中，调用词法分析器接口 getType 读取下一符号并进行条件判断，根据读到的:=,IF,WHILE,BEGIN 等关键字分别调用对应类型函数进一步处理，形成递归调用。
    (a) 赋值语句处理: 若等式左部为常量则报错不能赋值，若读到赋值符号则移进做表达式处理后生成三地址代码，否则报错缺少赋值符。
    (b) 条件语句处理: 从词法分析器中获取下一个词法单元。获取条件语句的条件部分，并生成相应的代码。检查是否有 THEN 关键字，如果没有，则报错。生成跳转语句到真出口的代码。获取假出口的地址，并解析真出口的代码。设置假出口的代码为跳转到真出口的地址。
    (c) 循环语句处理: 从词法分析器中获取下一个词法单元。获取循环语句的条件部分，并生成相应的代码。检查是否有 DO 关键字，如果没有，则报错。生成跳转语句到真出口的代码，并记录 while 语句的地址；获取假出口的地址，并解析真出口的代码；生成跳转语句返回 while 语句的地址。

3.2.4 条件处理

在条件判断语句和循环语句中均调用到条件处理函数。对条件表达式式依次进行左部处理、关系运算符处理、右部处理，其中左右部处理属于表达式处理可分解为含因子的语法项处理。

1. 表达式处理: 从词法分析器中获取当前操作符。如果是加号或减号，则获取下一个词法单元作为项，并生成相应的代码。如果是减号，生成取负指令。如果没有符号，则直接获取项。获取下一个操作符，如果是加号或减号，则循环获取下一个项，并生成相应的代码，使用临时变量保存计算结果，直到没有加号或减号为止。返回表达式的值。
    (a) 项处理: 获取因子作为初始值。获取下一个操作符，如果是乘号或除号，则循环获取下一个因子，并生成相应的代码，使用临时变量保存计算结果，直到没有乘号或除号为止。返回项的值。
    (b) 因子处理: 获取当前词法单元的类型。如果是标识符，则获取其值，并在符号表中进行检查。如果是左括号，则递归调用 expression 函数，并在获取右括号后返回结果。如果是整数常量，则直接返回其值，否则报错。

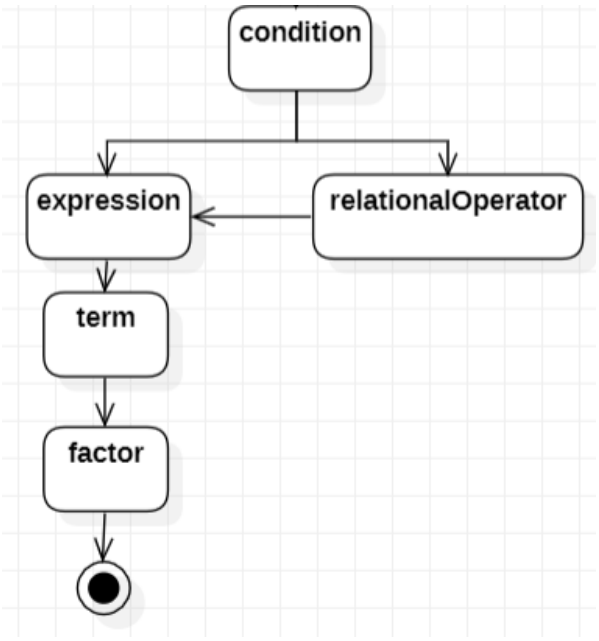


图 3.4 条件处理流程图

(c) 关系运算符: 获取当前词法单元作为关系运算符，并返回对应的字符串表示。

2. 关系运算符: 获取当前词法单元作为关系运算符，并返回对应的字符串表示。

3.3 中间代码生成部分

3.3.1 符号表存储

SymbolTable 类是一个简单的符号表实现，它使用数组来存储符号表的内容。

1. 符号表使用一个 SymbolTableRow 类型的数组来存储符号表的内容。数组的大小由 rowMax 确定，即最大表长为 50。数组的下标从 1 开始，而不是从 0 开始存储符号表的内容。
2. 符号表的行：每一行符号表的内容由 SymbolTableRow 对象表示，包括名称和类型。
3. 初始化：在构造函数中，符号表被初始化为 rowMax+1 行，并且每一行都被赋予空字符串和 Enum-Char.nul 类型。
4. 插入操作：当语法分析器调用 enterTable 方法时，会向符号表中插入一行数据。tablePtr 指向符号表中已经填入值最后一项，每次插入时 tablePtr 增加，并且在数组中存储相应的名称和类型。
5. isFull 方法：可以检查符号表是否已满，即 tablePtr 是否等于 rowMax。
6. 查询操作：checkExistence 方法用于检查符号是否存在于符号表中，遍历符号表中的每一行，查找匹配的名称，并返回其位置。
7. 打印操作：printTable 方法用于打印符号表的内容，包括名称和类型。

3.3.2 三地址代码生成与输出

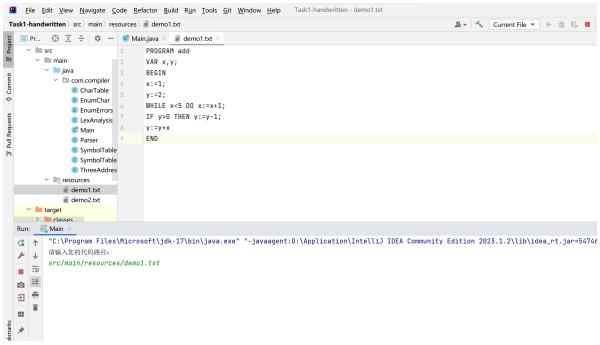
ThreeAddressCodeGen 类实现了中间代码的存储和输出操作。

1. 中间代码存储：该类使用 ArrayList<String>来存储中间代码。每一条中间代码被存储为一个字符串，并添加到 ArrayList 中。存储的过程通过 emit 方法实现，该方法将中间代码字符串添加到 ArrayList 中，并返回一个地址 ID。
2. 地址 ID 管理：类中使用 addrId 来管理地址 ID，beginAddId 作为初始地址 ID。emit 方法添加中间代码时，addrId 增加并返回该地址 ID。nextAddr 方法用于获取下一个地址 ID。
3. 修改中间代码：setAddrCode 方法用于修改指定地址 ID 的中间代码。
4. 输出中间代码：printAll 方法用于将存储的中间代码按照地址 ID 顺序打印输出到控制台。

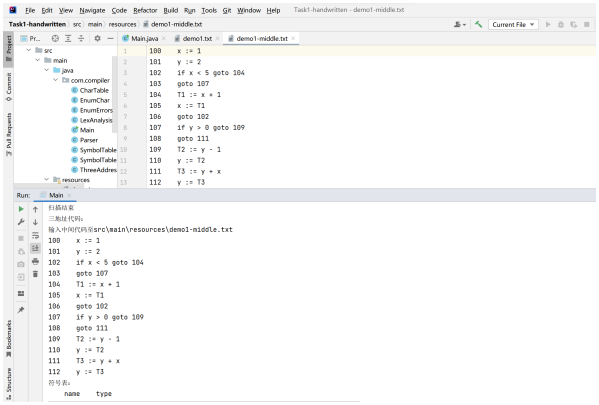
4 系统测试

4.1 示例程序测试

1. 运行程序，输入源代码文件的地址，如图 (a) 所示。



(a) 运行程序并输入源代码文件的地址



(b) 输出相应内容到控制台与文件

2. 程序输出中间代码、符号表（如有语法错误则输出错误）到控制台，并把中间代码保存至文件，如图 (b) 所示。

输入代码为：

```
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
```

输出代码为：

```
100 x := 1
101 y := 2
102 if x < 5 goto 104
103 goto 107
104 T1 := x + 1
105 x := T1
106 goto 102
107 if y > 0 goto 109
108 goto 111
109 T2 := y - 1
110 y := T2
111 T3 := y + x
112 y := T3
```

4.2 针对所有词法规则、语法规则的测试

输入代码为：

```
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
```

输出代码为:

```
PROGRAM SampleCalc
CONST
  maxLimit := 100;
VAR
  num1, num2, count, result;
BEGIN
  num1 := 15;
  num2 := 5;
  count := 0;
  result := 0;

  WHILE count < maxLimit DO
  BEGIN
    IF num1 > num2 THEN
    BEGIN
      result := num1 + num2
    END;

    IF num2 <= num1 THEN
    BEGIN
      result := result - (num1 - num2)
    END;

    IF result >= 50 THEN
    BEGIN
      result := result / 2;
    END;

    IF result < 25 THEN
    BEGIN
      result := result * 2
    END;

    num1 := num1 + 1;
    num2 := num2 + 1;
    count := count + 1
  END
END
```

输出代码为:

```
100 maxLimit := 100
101 num1 := 15
102 num2 := 5
103 count := 0
104 result := 0
105 if count < maxLimit goto 107
106 goto 131
107 if num1 > num2 goto 109
108 goto 111
```

```
109 T1 := num1 + num2
110 result := T1
111 if num2 <= num1 goto 113
112 goto 116
113 T2 := num1 - num2
114 T3 := result - T2
115 result := T3
116 if result >= 50 goto 118
117 goto 120
118 T4 := result / 2
119 result := T4
120 if result < 25 goto 122
121 goto 124
122 T5 := result * 2
123 result := T5
124 T6 := num1 + 1
125 num1 := T6
126 T7 := num2 + 1
127 num2 := T7
128 T8 := count + 1
129 count := T8
130 goto 105
```

5 项目总结

5.1 项目亮点

我们词法分析器的开发基于正则文法，在这一部分，我们设计了精简而高效的数据结构，以优化输入的读取、识别多种类型的词法单元（如标识符、关键字、运算符等）。语法分析器的函数划分及设计充分展现了递归下降程序结构的直观和美观，保持了算法的时空复杂度在一个较低的水平。

本项目的一个亮点是添加了报错的自动定位功能，LexGetLine() 函数获取当前行数，在错误处理 ErrorEnum 类中设计了 error 函数定位错误位置和错误类型，既方便操作者查找修改错误，还提高整体 IO 操作的效率，确保程序高效稳定执行。

5.2 思考与收获

本项目实现了一款轻量级的基于自上而下递归下降分析法的类 pl/0 词法（含语法）分析器。这个项目不仅实现了一个功能完善的词法分析器，而且构建了一个高效的语法分析器，将不同类型语法的分析单独封装为一个函数，逻辑明确，且模块之间的划分和相互调用清晰，提高了团队合作效率。

通过这次项目，我们对《编译原理》这门课程有了更深入的理解。我们不仅清晰地认识到了词法分析器和语法分析器的功能及其关联，还将 First 集、递归下降方法等理论知识应用到了实践中，从而真正理解了这 些概念的实际意义。这个项目的整个设计过程也让我们明白了编译器究竟一步一步是在做什么，应该怎么用代码将编译代码的“东西”实现。当然，这一次的程序开发，也让我们各自的 debug 和 coding 能力加强了很多，例如如何处理文本文件，如何设计一天高效而又功能齐全的状态机，如何降低程序核心函数算法的时空复杂度等等。其次，我们认识到良好的项目设计对于任务的顺利完成至关重要。尤其是在数据结构设计方面，我们的预先规划和不断地推敲设计使得我们在后续实际编码过程中减少了许多困难和返工。而项目早期阶段的明确任务规划、分工和时间规划，保证了工作的高效进行。团队成员间的积极交流和进度共享，是项目成功的关键。因此，这个项目不仅加深了我们对编译原理的理解，还锻炼了我们的设计思维、编程技能和团队合作能力。

任务 2: 编译工具使用

6 解决方案的整体介绍

本项目的主要任务是将 PL0 源代码转化为中间代码，ANTLR 工具用于自动生成词法分析器和语法分析器，极大地简化了解析阶段的工作量。ANTLR 生成的基础访问器类被扩展为 `pl0VisitorImpl`，用于遍历解析树并生成中间代码。

6.1 编译器构建策略

编译器的构建采用了逐步迭代的开发策略，分为以下几个主要阶段：

6.1.1 顺序语句支持

在编译器的初始开发阶段，重点放在处理程序中的顺序语句上。此阶段包括对变量和常量的声明以及赋值语句的处理，这是构建编译器的基础。

6.1.2 控制语句与跳转标签

一旦基本的顺序语句处理稳定后，引入了对控制语句的支持。这包括了 IF 和 WHILE 等结构，需要生成跳转指令来控制程序的执行流。

关键实现思路：

- 跳转标签的引入：每个控制语句都会引入跳转标签，最初这些标签只是占位符。
- 生成中间代码：在解析树中对应控制结构的节点生成特定的中间代码，如逻辑判断和跳转指令。

6.1.3 标签到地址的映射

控制语句的支持之后，下一步是将跳转标签映射到实际的地址。

关键实现思路：

- 删除标签：改为占位符和回填的策略。
- 地址回填：在知道跳转地址后，将占位符替换为具体的地址。

6.1.4 符号表管理与错误处理

最终阶段是引入符号表来跟踪标识符的声明和使用，以及实现语义错误的检测和处理。

关键实现思路：

- 符号表设计：符号表记录了每个标识符的属性、是否赋值、是否使用、是否声明、行号跟踪等。
- 错误检测：编译器在解析过程中检查每个标识符的合法性，包括是否声明、是否重复声明和是否在赋值前使用。

7 工具客户化工作

7.1 循环语句的处理

```java
@Override
public String visitLoopStatement(pl0Parser.LoopStatementContext ctx) {
    // 记录循环条件开始的地址
    int whileCondStart = currentCodeLine;
    // 访问条件，并生成条件计算的中间代码
    String condition = visit(ctx.condition());
    // 记录跳转到循环体的地址的占位符位置
    int gotoDoStartIndex = currentCodeLine;
    emit( code: "IF " + condition + " GOTO " + PLACEHOLDER);
    // 记录跳出循环的跳转指令的占位符位置
    int gotoAfterWhileIndex = currentCodeLine;
    emit( code: "GOTO " + PLACEHOLDER);
    // 访问循环体之前，保存循环体开始的地址
    int doStart = currentCodeLine;
    // 访问循环体
    visit(ctx.statement());
    // 循环体访问完成后，设置结束循环条件后的地址
    // 替换占位符为正确的跳转地址
    intermediateCode.set(gotoDoStartIndex, "IF " + condition + " GOTO " + doStart);
    // 循环体结束后跳转回循环条件判断
    emit( code: "GOTO " + whileCondStart);
    // emit("GOTO " + whileCondStart);做完了才是循环真正结束了
    int afterWhile = currentCodeLine;
    intermediateCode.set(gotoAfterWhileIndex, "GOTO " + afterWhile);
    return null;
}
```

图 7.1 visitLoopStatement 方法截图

记录循环条件的起始地址 `int whileCondStart = currentCodeLine`,生成循环条件的中间代码 `String condition = visit(ctx.condition())`，放置跳转到循环体和跳出循环的占位符 `int gotoDoStartIndex = currentCodeLine; int gotoAfterWhileIndex = currentCodeLine`，记录循环体的起始地址 `int doStart = currentCodeLine`，访问循环体并生成中间代码 `visit(ctx.statement())`，替换跳转占位符 `intermediateCode.set(gotoDoStartIndex, "IF " + condition + " GOTO " + doStart);`，循环体结束后跳转回条件判断 `emit("GOTO " + whileCondStart);`，处理循环结束后的跳转。最后一句 emit 做完之后才能回填 `intermediateCode.set(gotoAfterWhileIndex, "GOTO " + afterWhile);`。

7.2 变量声明的处理

visitVarDeclaration 方法处理变量声明，首先提取上下文中的变量列表 `ctx.ident()`，然后遍历每个变量标识符，获取变量名和行号，并尝试在符号表中声明。若存在声明异常，则报告错误。接着，将所有变量名连接成一个逗号分隔的字符串，并格式化为标准的变量声明输出 `System.out.println("VAR " + joinedVariableNames + ";")`。该方法最终返回 `null`，表示其主要作用是更新符号表和输出变量声明而非返回值。

7.3 表达式节点处理

visitExpression 方法通过初始化 `result` 变量来开始处理表达式。它检查前缀正负号，并设置 `prefixNum` 来指示其存在。对于有前缀正负号的表达式，会相应地处理第一个项的结果，并使用中间代码来表示项的正负。后续项与前面的加法运算符一起被处理，累积结果存储在新的临时变量中，通过 `emit` 方法输出中间代码。最后，方法返回最终的表达式结果存储在 `result` 变量中，其中 `processTerm` 函数用于辅助处理加法和减法操作。

7.4 项和因子的处理

visitTerm 函数递归地处理表达式中的项（由因子组成），生成与乘除有关的中间代码。具体步骤如下：

1. 初始化 `result` 存储项的结果。
2. 当项只包含单个因子时，调用 `visit(ctx.factor(0))` 并将结果赋予 `result`。
3. 对于多因子项，首个因子结果存入 `result`，随后迭代处理剩余因子：
   - 提取乘除运算符。
   - 访问下一因子，将结果赋予 `nextTerm`。
   - 生成临时变量 `tempVar`，存储中间结果。
   - 输出运算中间代码，更新 `result` 为当前累计结果。
4. 返回 `result`，项的最终结果。

visitFactor 函数访问因子节点，处理不同类型的因子，逻辑如下：

1. 对于标识符因子：
   - 访问标识符节点，获取字符串和行号。
   - 更新符号表，标记标识符已使用，异常时打印信息。
2. 对于数值因子：
   - 直接访问节点，获取数值字符串表示。
3. 对于括号表达式因子：
   - 递归访问括号内表达式节点，获取其值。
4. 根据因子类型，返回相应的字符串表示或 `null`。

这两个函数通过处理表达式中的乘法、除法和括号优先级，确保了中间代码的正确生成和表达式的正确计算。

8 系统测试

8.1 对于复杂表达式的测试

8.2 其余测试

本次系统测试采用了独立的 Task1 和 Task2 两个模块分开验证同一份测试输入源代码，最终生成的中间代码保持一致，从而全面验证了编译系统的正确性。测试包含了示例程序的运行测试、覆盖各种词法和语法规则的测试。可以确定系统满足设计需求，各模块实现正确，能够正确处理各类输入源代码并转换为中间代码。

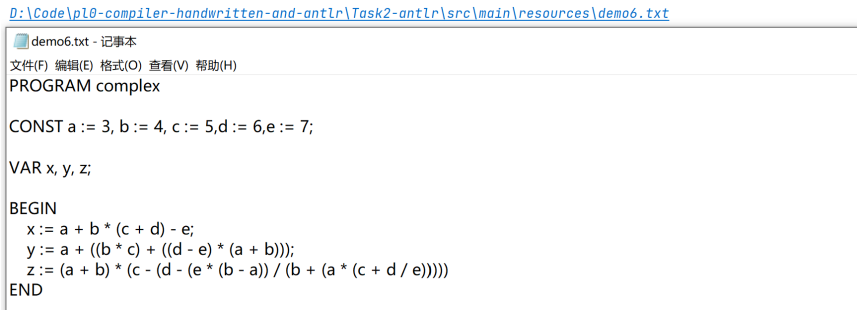


图 8.1 对于复杂表达式的测试-输入

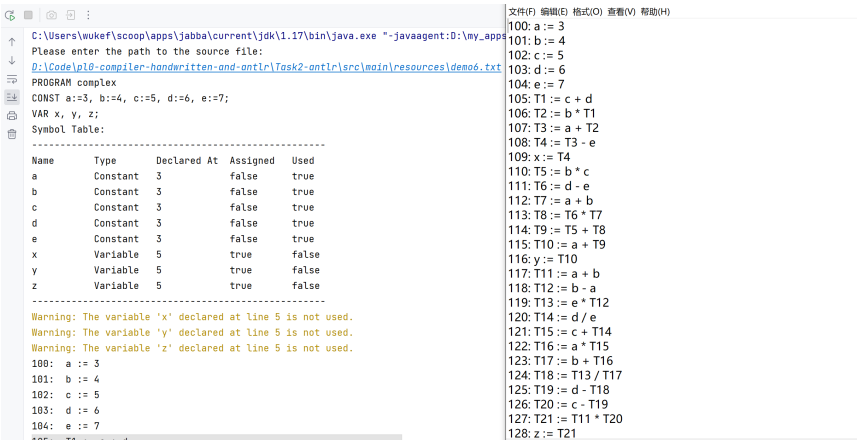


图 8.2 对于复杂表达式的测试-输出

9 项目总结

9.1 项目亮点

1. **详细的注释**：项目中每个类和方法都配有详细的注释，关键方法几乎每一句都有注释，这为后续的代码维护和团队协作提供了便利。注释的准确性和详尽性确保了代码的可读性和可维护性。
2. **自定义 ArrayList 实现**：通过重载 `ArrayList` 的 `set` 和 `indexOf` 方法，`MyArrayList` 类提供了一个基地址偏移功能，这对于 PL/0 编译器内部的内存管理有特别意义，使得数组索引与编译器的内存模型紧密结合。

```
public class MyArrayList<E> extends ArrayList<E> {
    ...
}
```

3. **符号表的有效管理**：`SymbolTable` 类通过内部的 `HashMap` 有效地管理符号信息，支持声明、赋值和使用的检查，以及使用情况的验证，这有助于及时捕捉和报告语义错误。

```
public class SymbolTable {
    ...
}
```

4. **语义错误处理**：在符号表中，对于变量和常量的声明、赋值和使用进行了详细的语义检查，当发现错误时，会抛出异常并提供清晰的错误信息，这有助于开发者快速定位和修正代码中的问题。
5. **代码风格的一致性**：整个项目遵循一致的代码编写规范，从命名约定到代码结构，都体现了专业的代码组织技能。
6. **警告和提示机制**：在未使用变量的检查中，`SymbolTable` 类使用 ANSI 颜色代码提供颜色编码的控制台输出，以提高警告的可见性，帮助开发者注意到可能的代码优化点。

9.2 思考与收获

在本项目中，我深入探索了 PL/0 编译器中对于复杂结构的处理，尤其是在访问和生成循环语句及表达式节点的中间代码时的理解和实践。

1. **循环语句的理解：**循环语句的处理过程中，在生成条件判断的中间代码后，我设置了一个占位符来代表循环体的开始，这是为了确保循环的逻辑正确性。之后我遇到了一个重要的实现细节：循环体内的代码必须完全生成，包括将控制流跳回到循环条件开始的 GOTO 指令。这个实现深化了我对循环控制和跳转语句的理解。
2. **表达式节点的挑战：**在处理表达式节点时，由于表达式的起始可能带有正负前缀符号，对于不同情况处理逻辑复杂。我通过断点调试的方式，逐步跟踪了代码的执行流程，最终顺利地完成了调试过程。这个过程不仅锻炼了我的调试技巧，还增进了我对程序流程控制和异常处理机制的理解。

这个实践经历不仅增强了我的编程技能，还提升了我对编译原理中关键概念的理解，特别是在语法树遍历、中间代码生成方面。在团队合作方面，作为组长，我通过定期组织会议和合理划分任务，体会到如何在深入理解项目需求后，有效地分配和管理团队资源，以达到最佳的工作效果。

References

[1] 陈火旺, 刘春林等. 程序设计语言编译原理（第 3 版）. 国防工业出版社, 2000.

[2] zombo_tany. 实验二：词法分析器的手动实现（基于状态机的词法分析器）. (2023, Sept 05). [Online]. Available: https://blog.csdn.net/qq_46640863/article/details/125671280.

[3] Hugo. Implementation of Semantic Analysis. (2019, Oct 13). [Online]. Available: https://p-grandinetti.github.io/compilers/page/implementation-semantic-analysis/.

[4] antlr. antlr4. [Online]. Available: https://github.com/antlr/antlr4