
# 《编译原理》实验报告

学　　院　计算机工程与科学学院

组　　号　10 组

实验题号　实验五

日　　期　2025 年 05 月 14 日

| 学号 | 姓名 | 主要工作 | 贡献因子 |
|---|---|---|---|
| 22122792 | 王潇 | 实验代码与报告 | 0.25 |
| 22121037 | 钱鹏程 | 实验代码 | 0.25 |
| 22121639 | 何勇乐 | 实验代码与报告 | 0.25 |
| 22121630 | 汪江豪 | 实验代码 | 0.25 |

# 《编译原理》实验报告

## 一、 实验题目

根据 PL/0 语言文法，在实验三的表达式语法分析程序中添加语义处理部分，生成表达式的中间代码，并以四元式序列的形式表示。在已有的语法分析器基础上扩展语义子程序，确保能够正确处理经过语法分析后的合法表达式，并输出其对应的中间代码。

## 二、 实验内容设计与实现

在实验四的语义值计算程序中，使用了 SLR(1)分析法。要完成四元式的产生，只需要在实验四的基础上，在每次进行归约时，根据操作数和操作符产生一个四元式即可。

在实验五的 doReduction 函数中，四元式的生成与语法规则的归约过程紧密结合在一起。每当函数根据当前的文法规则编号进入对应的 case 分支后，它会依据该规则所代表的语法结构和操作语义构造新的符号，并在此过程中记录相应的运算信息，从而形成四元式的基本内容。

代码 1　doReduce 新增加四元式的生成

```
...
new_quad.operator_ = "=";
new_quad.operand1  = fourth;
new_quad.operand2  = "_";
new_quad.result    = "t" + to_string(quadrupleQueue.size() + 1);
...
if (new_status != "") {
    if (isalpha(fourth.at(0))) {

AnalysisStack.push(AnalysisStackElement(stoi(new_status),
          new_symbol, to_string(top_value), fourth));
    }
    else {

AnalysisStack.push(AnalysisStackElement(stoi(new_status),
          new_symbol, to_string(top_value), new_quad.result));
        quadrupleQueue.push(new_quad);
    }
}
else {
    cout << "Syntax ERROR!" << endl;
    return false;
}
```

在每次执行归约时，函数首先从分析栈中弹出若干个元素，这些元素对应的是当前被规约的产生式右部的各个符号。每个元素都包含了状态、符号、语义值以及可能的操作数名称。函数将这些操作数取出后，根据当前的语法规则

进行相应的运算处理，例如加法、减法、乘法、除法等，同时也会处理括号表达式或单目负号等特殊结构。

在完成运算逻辑的同时，函数会创建一个新的四元式对象，并将其运算符字段设置为当前操作的符号，例如“+”、“-”、“*”或“/”，并将两个操作数字段分别设置为参与运算的两个操作数的标识符或临时变量名。结果字段则通常是一个新生成的临时变量名，用于保存本次运算的结果，以便后续的归约步骤能够继续使用这个值。这个四元式对象随后被压入四元式队列中，作为整个程序中间代码生成的一部分。

四元式的生成并非仅仅依赖于语义值的计算，而是更多地基于对符号的引用和临时变量的管理。在遇到标识符时，函数不会立即进行数值计算，而是保留其原始名称；而在遇到常量时，则直接参与运算并生成新的临时变量名。这种设计使得程序在面对包含变量的表达式时仍能正确生成中间代码，而不仅仅局限于可以立即求值的常量表达式。

此外，四元式的生成流程在整个函数中保持高度统一：无论处理哪种类型的二元运算或一元运算，函数都会按照“取操作数—执行运算—生成结果—构造四元式”的顺序依次完成。这种一致性不仅增强了代码的可读性和可维护性，也使得新增语法规则时能够较为容易地扩展相应的语义动作。

## 三、 实验数据测试

由于实验五的主要任务是实现四元式的生成，因此在测试时，我们主要关注四元式的正确性和完整性。即不考虑语法分析和语义分析出错的情况，因为这些情况下无法生成四元式。

### 3.1. 测试 1

输入如下，中间添加了标识符和常量运算。

```
1.  +8/9*(5+bc)-8*((-ident)*8-(-2+abc123))
```

输出如下，可以发现四元式的生成是正确的。

```
( =, 8, _, t1 )
( =, 9, _, t2 )
( /, t1, t2, t3 )
( =, 5, _, t4 )
( +, t4, bc, t5 )
( *, t3, t5, t6 )
( =, 8, _, t7 )
( -, 0, ident, t8 )
( =, 8, _, t9 )
( *, t8, t9, t10 )
( =, 2, _, t11 )
( -, 0, t11, t12 )
( +, t12, abc123, t13 )
( -, t10, t13, t14 )
( *, t7, t14, t15 )
( -, t6, t15, t16 )
```

### 3.2. 测试 2

输入如下，添加了单个标识符被括号包围的情况。

```
-5*(+c)/9-4*(5*a*2)/(8/(-c))
```

输出如下，可以发现四元式的生成是正确的。

```
( =, 5, _, t1 )
( *, t1, c, t2 )
( =, 9, _, t3 )
( /, t2, t3, t4 )
( -, 0, t4, t5 )
( =, 4, _, t6 )
( =, 5, _, t7 )
( *, t7, a, t8 )
( =, 2, _, t9 )
( *, t8, t9, t10 )
( *, t6, t10, t11 )
( =, 8, _, t12 )
( -, 0, c, t13 )
( /, t12, t13, t14 )
( /, t11, t14, t15 )
( -, t5, t15, t16 )
```

### 3.3. 测试 3

输入如下，主要测试多复杂运算符组成的表达式。

```
1/2/3*(-2*(-3)*(-4))+5/6-1*2
```

输出如下，可以发现四元式的生成是正确的。

```
( =, 1, _, t1 )
( =, 2, _, t2 )
( /, t1, t2, t3 )
( =, 3, _, t4 )
( /, t3, t4, t5 )
( =, 2, _, t6 )
( =, 3, _, t7 )
( -, 0, t7, t8 )
( *, t6, t8, t9 )
( =, 4, _, t10 )
( -, 0, t10, t11 )
( *, t9, t11, t12 )
( -, 0, t12, t13 )
( *, t5, t13, t14 )
( =, 5, _, t15 )
( =, 6, _, t16 )
( /, t15, t16, t17 )
( +, t14, t17, t18 )
( =, 1, _, t19 )
( =, 2, _, t20 )
( *, t19, t20, t21 )
( -, t18, t21, t22 )
```

### 3.4. 测试 4

输入如下，输入多括号复杂表达式，无标识符。

```
1/2*3-(-5*(-8+5)/(+5))
```

输出如下，可以发现四元式的生成是正确的。

```
( =, 1, _, t1 )
( =, 2, _, t2 )
( /, t1, t2, t3 )
( =, 3, _, t4 )
( *, t3, t4, t5 )
( =, 5, _, t6 )
( =, 8, _, t7 )
( -, 0, t7, t8 )
( =, 5, _, t9 )
( +, t8, t9, t10 )
( *, t6, t10, t11 )
( =, 5, _, t12 )
( /, t11, t12, t13 )
( -, 0, t13, t14 )
( -, t5, t14, t15 )
```

### 3.5. 测试 5

输入如下，出现单括号表达式。

```
+8/2/3-4-2*(-2)/(+2)+4/9
```

输出如下，可以发现四元式的生成是正确的。

```
( =, 8, _, t1 )
( =, 2, _, t2 )
( /, t1, t2, t3 )
( =, 3, _, t4 )
( /, t3, t4, t5 )
( =, 4, _, t6 )
( -, t5, t6, t7 )
( =, 2, _, t8 )
( =, 2, _, t9 )
( -, 0, t9, t10 )
( *, t8, t10, t11 )
( =, 2, _, t12 )
( /, t11, t12, t13 )
( -, t7, t13, t14 )
( =, 4, _, t15 )
( =, 9, _, t16 )
( /, t15, t16, t17 )
( +, t14, t17, t18 )

```

## 四、 实验总结

本次实验的主要任务是实现四元式的生成。通过对实验四的语义值计算程序进行扩展，我们成功地在每次归约时生成了相应的四元式，并将其存储在四元式队列中。 在实现过程中，我们深入理解了四元式的结构和生成逻辑，并通过对语法规则的分析，设计了合理的四元式生成策略。实验结果表明，我们的实现是正确的，能够处理多种复杂的表达式运算。 在实验总结中，我们认识到四元式的生成不仅仅是一个简单的过程，而是一个涉及到语法分析、语义分析和代码生成等多个方面的复杂任务。通过本次实验，我们对编译原理中的四元式生成有了更深入的理解，也为后续的编译器设计和实现打下了良好的基础。