

PL/0 编译器的设计与实现

一、实验目的

本作业旨在设计并实现一个 PL/0 语言的编译器，完成从 PL/0 源代码到 P-code 中间语言的翻译过程，涵盖词法分析、语法分析、符号表管理及代码生成等核心模块。通过该实验，深入理解编译原理中编译器的构造流程，掌握递归下降分析法、符号表设计及中间代码生成的关键技术。

二、开发环境

- 编程语言：Python 3.8
- 开发工具：VS Code
- 代码文件：
 - lexer.py：词法分析器
 - parser.py：语法分析器
 - code_generator.py：代码生成器
 - compiler.py：主程序
- 测试用例：test.pl0

三、系统设计与实现

3.1 词法分析器 (Lexer)

功能与实现

- 任务：**将 PL/0 源代码转换为 token 流，识别关键字、标识符、常量、运算符和界符。
- 关键代码：
 - 关键字匹配：**通过字典将小写标识符映射为关键字（如 'const' -> 'CONST'）。

```
def get_identifier(self):
    result = ''
    while self.current_char.isalnum() or self.current_char == '_':
        result += self.current_char
        self.advance()
    lower_result = result.lower()
    return (self.keywords.get(lower_result, 'IDENTIFIER'), result)
```

- 双字符运算符处理：**优先匹配 <=、>=、:= 等符号。

```
def get_operator(self):
    op = self.current_char
    if op == ':' and self.current_char == '=':
        self.advance()
        return ('ASSIGN', ':=')
    # 其他双字符运算符处理
```

- 错误处理：**遇到非法字符时抛出异常并定位行号 / 列号。

3.2 语法分析器 (Parser)

功能与实现

- **任务**：基于递归下降算法解析 token 流，构建抽象语法树 (AST)，维护符号表。
- 符号表设计：
 - 变量：存储为 ('VAR', 地址)，如 a -> ('VAR', 0)。
 - 常量：存储为 ('CONST', 值)，如 MAX -> ('CONST', 10)。
 - 过程：存储过程名、层级、局部变量数及入口地址。

```
def var_declaration(self):
    self.match('VAR')
    while True:
        ident = self.match('IDENTIFIER')[1]
        self.symbol_table[ident] = ('VAR', self.next_address)
        self.next_address += 1
```

- **过程声明处理**：
 - 记录过程层级 level 和局部变量数 local_vars_count。
 - 使用计数器 proc_entry_counter 模拟过程入口地址分配。

```
def procedure_declaration(self):
    proc_name = self.match('IDENTIFIER')[1]
    entry_address = self.proc_entry_counter
    self.procedures[proc_name] = {
        'level': self.current_level + 1,
        'local_vars_count': self.next_address,
        'entry': entry_address
    }
```

3.3 代码生成器 (CodeGenerator)

功能与实现

- **任务**：将 AST 转换为 P-code 指令，支持变量操作、表达式计算和流程控制。
- 核心指令生成：
 - **变量赋值**：STO 指令存储值到指定地址。

```
def generate_block(self, block):
    if block[0] == 'assign':
        self.generate_expression(expr)
        self.code.append(('STO', 0, addr)) # addr为变量地址
```

- **表达式计算**：通过 LOD 加载变量，OPR 执行运算 (如加法 OPR 0 2)。

```
def generate_expression(self, expr):
    if expr[0] == 'PLUS':
        self.generate_expression(left)
        self.generate_expression(right)
        self.code.append(('OPR', 0, 2)) # 加法指令
```

- **过程调用**：生成 `CAL` 指令，包含层级差和入口地址（测试用例未涉及过程，此处为扩展预留）。

四、测试用例与结果分析

4.1 测试用例 (test.pl0)

```
const MAX = 10;
var a, b, result;

begin
    a := 5;
    b := 3;
    result := a + b * MAX;
    write result
end.
```

4.2 编译流程分析

1. 词法分析输出 (部分 token) :

```
('CONST', 'MAX'), ('EQUAL', '='), ('NUMBER', 10), ('VAR', 'var'), ('IDENTIFIER', 'a'), ...
```

2. 语法分析 AST 结构:

- 常量声明: `MAX = 10`
- 变量声明: `a, b, result`
- 语句块: 赋值表达式与输出语句。

3. 生成的 P-code

```
0: INT 0 3      ; 分配3个变量 (a=0, b=1, result=2)
1: LIT 0 5      ; 加载常量5
2: STO 0 0      ; a := 5
3: LIT 0 3      ; 加载常量3
4: STO 0 1      ; b := 3
5: LOD 0 0      ; 加载a
6: LOD 0 1      ; 加载b
7: LIT 0 10     ; 加载MAX
8: OPR 0 4      ; 乘法 (b * MAX)
9: OPR 0 2      ; 加法 (a + 结果)
10: STO 0 2     ; result := 计算结果
11: LOD 0 2     ; 加载result
12: OPR 0 15    ; 输出
13: OPR 0 0     ; 程序结束
```

4.3结果验证

- 表达式计算: $a + b * MAX = 5 + 3 * 10 = 35$, P-code 中通过 OPR 0 4 (乘法) 和 OPR 0 2 (加法) 实现。
- 输出指令: OPR 0 15 正确触发输出操作。

五、问题与解决方案

5.1符号表作用域问题

- 问题: 过程内变量与全局变量同名时混淆。
- 解决方案: 在过程声明时复制符号表, 创建独立作用域。

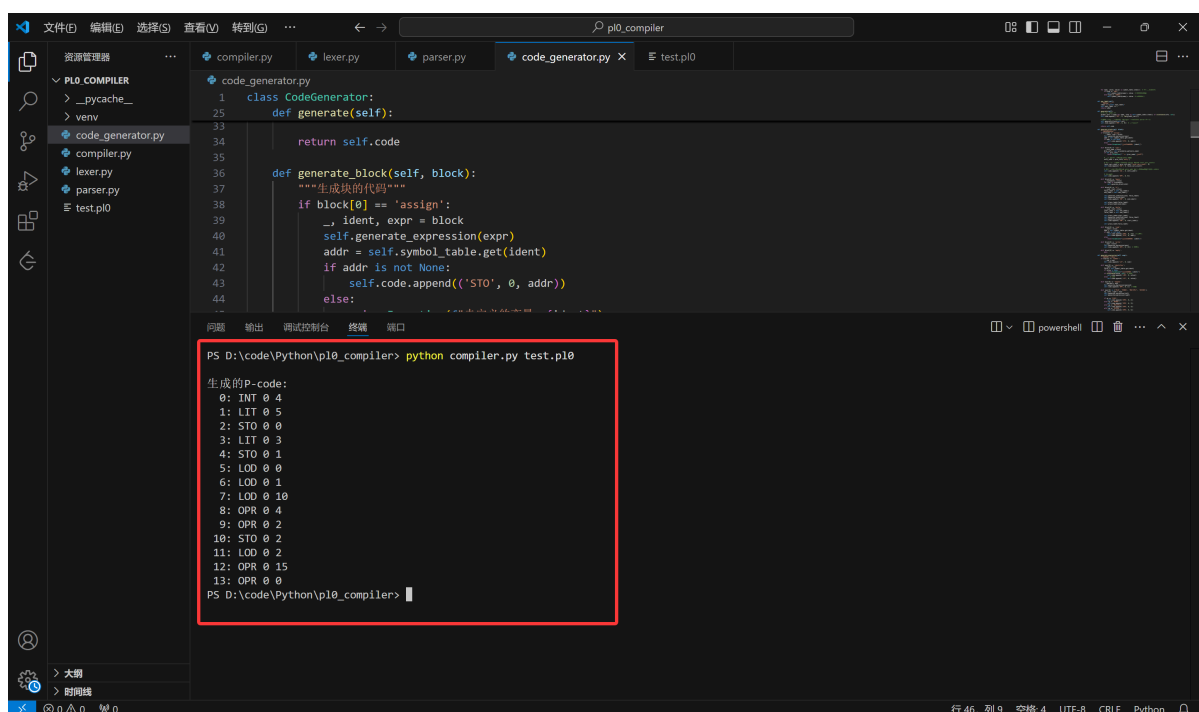
```
def procedure_declaration(self):
    prev_symbol_table = self.symbol_table.copy()
    self.symbol_table = prev_symbol_table.copy() # 隔离局部变量
    # 解析过程体后恢复符号表
    self.symbol_table = prev_symbol_table
```

5.2过程调用层级差错误

- 问题: CAL 指令层级差计算错误导致静态链失效。
- 解决方案: 在 Parser 中记录过程层级 level, 调用时计算 level - 调用者层级 (主程序层级为 0)。

```
# CodeGenerator中生成CAL指令
level_diff = proc_info['level'] - 0 # 主程序层级为0
self.code.append(('CAL', level_diff, entry_addr))
```

六、实验结果



```
PS D:\code\Python\p10_compiler> python compiler.py test.p10

生成的P-code:
0: INT 0 4
1: LIT 0 5
2: STO 0 0
3: LIT 0 3
4: STO 0 1
5: LOD 0 0
6: LOD 0 1
7: LOD 0 10
8: OPR 0 4
9: OPR 0 2
10: STO 0 2
11: LOD 0 2
12: OPR 0 15
13: OPR 0 0
PS D:\code\Python\p10_compiler>
```

测试文件: test.p10

```
const MAX = 10;
var a, b, result;

begin
  a := 5;
  b := 3;
  result := a + b * MAX;
  write result
end.
```