实验报告：语义分析器的设计与实现

王贤义

计算机基地班

320210931221

实验目的

理解语义分析器的设计与实现： 掌握语义分析器的设计和实现原理。

理解类型检查的分析和实现： 在语义分析的过程中，对应每一个产生式，根据综合属性判断是否发生类型错误的问题。

语法规则

```
PROG     -> {    DECLS    STMTS    }
DECLS    -> DECLS DECL | $
DECL     -> int  NAMES  ;  |  bool  NAMES  ;
NAMES    -> NAMES ,  NAME  |  NAME
NAME     -> id
STMTS    -> STMTS  STMT  |   STMT
STMT     -> id  =  EXPR ;    |   id := BOOL ;
STMT     -> if  id   then  STMT
STMT     -> if  id   then  STMT  else STMT
STMT     -> while   id  do  STMT
STMT     -> {  STMTS   STMT  }
STMT     -> read  id  ;
STMT     -> write  id  ;
EXPR     -> EXPR  ADD  TERM  |  TERM
ADD      -> + | -
TERM     -> TERM  MUL NEGA  |  NEGA
MUL      -> * | /
NEGA     -> FACTOR  |  - FACTOR
FACTOR   -> (  EXPR ) |  id  |  number
BOOL     -> BOOL  ||  JOIN    |    JOIN
JOIN     -> JOIN   &&   NOT  |   NOT
NOT      -> REL   |  ! REL
REL      -> EXPR   ROP  EXPR
ROP      -> >  |  >=  |  <  |  <=  |  ==  |   !=
```

通过分析显然此语法存在不可避免的移进规约冲突（if else），本语法不属于SLR语法，但通过强制规定else与最近的if连接，当if id then STMT • 遇见else时，强制移进>规约。

设计约束

（1）从技术路线上说，语义分析应遵循语法制导翻译方法，在语法分析的基础之上进行。

（2）从处理过程上来说，整个分析过程对源程序处理两遍或三遍。例如，第一遍是词法分析和语法分析，以语法分析为主导，工作成果是语法错误报告或语法树；第二遍是类型检查和代码翻译，工作成果是类型错误报告或中间代码。在第一遍工作中报告错误后就退出分析，不再进入第二遍处理。第二遍类似，只有在没有任何错误的情况下才进入代码翻译工作。也可以把类型检查做成第二遍，把代码翻译做成第三遍。

（3）从程序结构上来说，语义分析要调用前面词法和语法分析的程序。所以建议将该程序设计为多个模块，至少要明确区分出驱动模块和工作模块。驱动模块包含了程序的入口和出口，主要负责输入、输出处理并按照设定的工作逻辑调用各个工作模块来完成工作；工作模块可以是：词法分析器、语法分析器、语义分析器的分解或组合，还可以专门设计一些子模块来完成特定工作，如：遍历树、查填符号表、报告错误等。

（4）从数据结构上说，语法树和符号表都是非常重要的全局数据结构，应该做好相应的设计。

程序设计与实现

工作流程

1. **移进**：移进过程中将文法符号信息压入
2. **规约**：规约过程中执行类型检查和代码翻译

关键算法描述

- 移进
    - 在移进过程中通过检测if，while，else关键词来记录其对应的入口地址

```
if (
    self.action_goto_tables_[(current_state, word_string)].op
    == SLR_OPERATIONS.MOVE
):  # 移进操作
    if get_word.word_string == 'while':
        self.sementic_analyzer_.backpatching_level_ += 1
        self.sementic_analyzer_.temp[self.sementic_analyzer_.backpatching_level_]=self.sementic_analyzer_.PeekNextStateNum()
        self.sementic_analyzer_.GetNextStateNum()
        self.sementic_analyzer_.GetNextStateNum()

    if get_word.word_string == 'if':
        self.sementic_analyzer_.backpatching_level_ += 1
        self.sementic_analyzer_.temp[self.sementic_analyzer_.backpatching_level_]=self.sementic_analyzer_.PeekNextStateNum()
        self.sementic_analyzer_.GetNextStateNum()
        self.sementic_analyzer_.GetNextStateNum()

    if get_word.word_string == 'else':
        self.sementic_analyzer_.temp_else[self.sementic_analyzer_.backpatching_level_] = self.sementic_analyzer_.PeekNextStateNum()
        self.sementic_analyzer_.GetNextStateNum()
    self.state_sequence_stack_.append(
        self.action_goto_tables_[(current_state, word_string)].state
    )
    self.move_conclude_string_stack_.append(word_string)
    self.PrintAnalysisProcess(
        sytactic_step,
        self.action_goto_tables_[(current_state, word_string)],
        get_word
    )
    sytactic_step += 1
    self.grammar_symbol_info_stack_.append(
        # {get_word.word_string, get_word.value}
        GrammarSymbolInfo(symbol_name=get_word.word_string, txt_value=get_word.value)
    )  # 文法符号信息压入
    break
```

- 规约

```
if not self.sementic_analyzer_.ExecuteSemanticCheck(
    self.grammar_symbol_info_stack_, self.productions_[conclude_production_number]
):  # 语义分析错误，则退出
    return False
```

- 语义分析的关键代码在ExecuteSemanticCheck()中，包含类型检查和代码翻译

最后规约到PROG表示语义分析结束。

```
        if "PROG" == production.left:
            # PROG -> { DECLS STMTS }
            print("中间代码生成完毕！")
            conclude_symbol_info =
GrammarSymbolInfo(symbol_name=production.left)
            self.PopAndAppend(symbol_info_stack, conclude_symbol_info,
production)
```

这段代码是针对特定语法产生式 `DECL -> int NAMES;` 的处理逻辑。当语义分析器遇到符合这一规则的语法结构时，它会执行以下步骤：

1. 获取标识符信息，存储在$NAMES$中，该信息在语法分析栈中的倒数第二个位置。
2. 遍历标识符列表，并为每个标识符创建一个符号表条目，表示一个整数类型的变量。
3. 记录已完成的符号信息，包括产生式的左部和标识符的文本值。
4. 更新语法分析栈，将处理过的符号信息出栈，并将新的符号信息入栈。

```
        elif "DECL" == production.left and production.right[0] == "int":
            # DECL -> int NAMES;
            symbol_identifier = symbol_info_stack[-2]
            for identifier_txt_value in symbol_identifier.txt_value:
                variable_symbol = Symbol()
                variable_symbol.mode = CATEGORY.VARIABLE  # 类型
                variable_symbol.name = identifier_txt_value  # 名称
                variable_symbol.type = VARIABLE_TYPE.INT
                variable_symbol.value = ""  # 字面值还没有被赋值，此处为空
                symbol_pos = self.symbol_tables_[0].AddSymbol(
                    variable_symbol
                )  # 加入符号表中
            conclude_symbol_info = GrammarSymbolInfo(
                symbol_name=production.left,
txt_value=symbol_identifier.txt_value
            )
            self.PopAndAppend(symbol_info_stack, conclude_symbol_info,
production)
```

这段代码处理形如 `NAMES -> NAMES , NAME` 的语法规则。具体步骤如下：

1. 获取符号栈中倒数第三个元素的文本值，并将其复制到新的符号信息对象中。
2. 将符号栈中倒数第一个元素的文本值添加到新符号信息对象的文本值列表中。
3. 更新语法分析栈，将处理过的符号信息出栈，并将新的符号信息入栈。

```
        elif "NAMES" == production.left and production.right[0] == "NAMES":
            # NAMES -> NAMES , NAME
            # 由规约后的产生式左部构造一个文法符号属性
            conclude_symbol_info = GrammarSymbolInfo(
                symbol_name=production.left,
                txt_value=symbol_info_stack[-3].txt_value.copy(),
            )

 conclude_symbol_info.txt_value.append(symbol_info_stack[-1].txt_value)
            self.PopAndAppend(symbol_info_stack, conclude_symbol_info,
production)
```

这段代码处理形如 `STMT -> id = EXPR;` 的语法规则。其主要逻辑包括：

1. 获取符号栈中的表达式和标识符的符号信息。

2. 在符号表栈中逆序查找标识符是否已被定义，如果找不到则输出语义错误信息。
3. 根据标识符的位置生成中间代码，表示将表达式的值赋给标识符。
4. 更新语法分析栈，将处理过的符号信息出栈，并将新的符号信息入栈。

```
        elif (
            "STMT" == production.left
            and production.right[0] == "id"
            and production.right[1] == "="
        ):
            # STMT -> id = EXPR;
            # 获取identifier与Expression的文法符号属性
            symbol_expression = symbol_info_stack[-2]
            symbol_identifier = symbol_info_stack[-4]
            # 在符号表栈中一级级逆序寻找 identifier 是否被定义过
            identifier_pos = -1
            identifier_layer = len(self.current_symbol_table_stack_) - 1
            while identifier_layer >= 0:
                search_table = self.symbol_tables_[0]
                identifier_pos =
search_table.FindSymbol(symbol_identifier.txt_value)
                if identifier_pos != -1:  # 表示在某一级中找到了
                    break
                identifier_layer -= 1

            if identifier_pos == -1:  # 没找到，表示不存在，说明没定义就使用，语义错误
                print("语义错误！！！", symbol_identifier.txt_value, "没有定义！")
                return False

            # 输出赋值的中间代码
            sp = SymbolPos()
            sp.table_pos = self.current_symbol_table_stack_[identifier_layer]
            sp.symbol_pos = identifier_pos
            result_name = self.GetArgName(sp)
            arg1_name = self.GetArgName(symbol_expression.pos)
            self.PrintQuadruples(
                Quadruples(self.GetNextStateNum(), ":=", arg1_name, "-",
result_name)
            )
            conclude_symbol_info =
GrammarSymbolInfo(symbol_name=production.left)
            self.PopAndAppend(symbol_info_stack, conclude_symbol_info,
production)
```

这段代码处理形如 `STMT -> if id then STMT else STMT` 的语法规则。其主要逻辑包括：

1. 获取符号栈中的标识符信息，表示条件表达式。
2. 在符号表中查找该标识符的位置，用于生成中间代码。
3. 根据条件表达式生成跳转指令，控制程序流程，生成跳转形式如下：

```
1 j, id, -, 3
2 j, -, -, 6
3 --- #then
4 ---
5 j, -, -, 8
6 --- #else
7 ---
8
```

4. 更新语法分析栈，将处理过的符号信息出栈，并将新的符号信息入栈。

```
        elif "else" in production.right:
            # STMT -> if id then STMT else STMT
            symbol_id = symbol_info_stack[-5]
            symbol_pos = self.symbol_tables_[0].FindSymbol(symbol_id.txt_value)
            symbol_name = self.GetArgName(SymbolPos(0, symbol_pos))
            backpatching_level_temp = self.backpatching_level_
            backpatching_level_temp = self.backpatching_level_
            self.PrintQuadruples(
                Quadruples(
                    self.temp[backpatching_level_temp],
                    "j",
                    symbol_name,
                    "_",
                    self.temp[backpatching_level_temp] + 2,
                )
            )
            self.PrintQuadruples(
                Quadruples(
                    self.temp[backpatching_level_temp] + 1,
                    "j",
                    "_",
                    "_",
                    self.temp_else[backpatching_level_temp] + 1,
                )
            )
            self.PrintQuadruples(
                Quadruples(
                    self.temp_else[backpatching_level_temp],
                    "j",
                    "_",
                    "_",
                    self.PeekNextStateNum(),
                )
            )

            self.backpatching_level_ -= 1
            if self.backpatching_level_ == 0:
                for Qua in self.quadruples_stack_:
                    self.PrintQuadruples(Qua)
                self.quadruples_stack_ = []
            conclude_symbol_info =
GrammarSymbolInfo(symbol_name=production.left)
            self.PopAndAppend(symbol_info_stack, conclude_symbol_info,
production)
```

这段代码处理形如 `STMT -> if id then STMT` 的语法规则。其主要逻辑包括：

1. 获取符号栈中的标识符信息，表示条件表达式。
2. 在符号表中查找该标识符的位置，用于生成中间代码。
3. 根据条件表达式生成跳转指令，控制程序流程，生成跳转形式如下：

```
1 j, id, -, 3
2 j, -, -, 5
3 --- #then
4 ---
5
```

4. 更新语法分析栈，将处理过的符号信息出栈，并将新的符号信息入栈。

```
        elif "if" == production.right[0]:
            # STMT -> if id then STMT
            symbol_id = symbol_info_stack[-3]
            symbol_pos = self.symbol_tables_[0].FindSymbol(symbol_id.txt_value)
            symbol_name = self.GetArgName(SymbolPos(0, symbol_pos))
            backpatching_level_temp = self.backpatching_level_
            self.backpatching_level_ -= 1
            self.PrintQuadruples(
                Quadruples(
                    self.temp[backpatching_level_temp],
                    "j",
                    symbol_name,
                    "_",
                    self.temp[backpatching_level_temp] + 2,
                )
            )
            self.PrintQuadruples(
                Quadruples(
                    self.temp[backpatching_level_temp] + 1,
                    "j",
                    "_",
                    "_",
                    self.temp_else[backpatching_level_temp] + 1,
                )
            )
            # then
            if self.backpatching_level_ == 0:
                for Qua in self.quadruples_stack_:
                    self.PrintQuadruples(Qua)
                self.quadruples_stack_ = []
            conclude_symbol_info = GrammarSymbolInfo(
                symbol_name=production.left,
            )
            self.PopAndAppend(symbol_info_stack, conclude_symbol_info,
production)
```

这段代码处理形如 `STMT -> while id do STMT` 的语法规则。其主要逻辑包括：

1. 获取符号栈中的标识符信息，表示条件表达式。
2. 在符号表中查找该标识符的位置，用于生成中间代码。
3. 根据条件表达式生成跳转指令，控制程序流程，生成跳转形式如下：

```
1 j, id, -, 3
2 j, -, -, 6
3 --- # do
4 ---
5 j, -, -, 1
6
```

4. 更新语法分析栈，将处理过的符号信息出栈，并将新的符号信息入栈。

```
        elif "while" == production.right[0]:
            # STMT -> while id do STMT
            symbol_id = symbol_info_stack[-3]
            backpatching_level_temp = self.backpatching_level_
            self.backpatching_level_ -= 1
            symbol_pos = self.symbol_tables_[0].FindSymbol(symbol_id.txt_value)
            symbol_name = self.GetArgName(SymbolPos(0, symbol_pos))
            self.PrintQuadruples(
                Quadruples(
                    self.temp[backpatching_level_temp],
                    "j",
                    symbol_name,
                    "-",
                    self.temp[backpatching_level_temp] + 2,
                )
            )
            self.PrintQuadruples(
                Quadruples(
                    self.temp[backpatching_level_temp] + 1,
                    "j",
                    "-",
                    "-",
                    self.PeekNextStateNum() + 1,
                )
            )
            # do
            if self.backpatching_level_ == 0:
                for Qua in self.quadruples_stack_:
                    self.PrintQuadruples(Qua)
                self.quadruples_stack_ = []
            self.PrintQuadruples(
                Quadruples(
                    self.PeekNextStateNum(),
                    "j",
                    "-",
                    "-",
                    self.temp[backpatching_level_temp],
                )
            )
            self.GetNextStateNum()
            conclude_symbol_info =
GrammarSymbolInfo(symbol_name=production.left)
            self.PopAndAppend(symbol_info_stack, conclude_symbol_info,
production)
```

这段代码处理形如 `STMT -> read id ;` 的语法规则。其主要逻辑包括：

1. 获取符号栈中的标识符信息，表示要读取值的变量。
2. 在符号表中查找该标识符的位置，用于生成中间代码。
3. 生成读取输入值的中间代码，并将结果存储到变量中。
4. 更新语法分析栈，将处理过的符号信息出栈，并将新的符号信息入栈。

```
        elif "read" == production.right[0]:
            # STMT -> read id ;
            symbol_info = symbol_info_stack[-2]
            symbol_pos =
self.symbol_tables_[0].FindSymbol(symbol_info.txt_value)
            symbol_name = self.GetArgName(SymbolPos(0, symbol_pos))
            self.PrintQuadruples(
                Quadruples(self.GetNextStateNum(), "read", "-", "-",
symbol_name)
            )
            conclude_symbol_info =
GrammarSymbolInfo(symbol_name=production.left)
            self.PopAndAppend(symbol_info_stack, conclude_symbol_info,
production)
```

这段代码处理形如 `EXPR -> EXPR ADD TERM` 的语法规则。其主要逻辑包括：

1. 获取符号栈中的两个表达式（EXPR）和一个加法操作符（ADD）的符号信息。
2. 为新的表达式结果创建一个符号表条目，并生成一个新的符号名。
3. 生成中间代码，表示对两个表达式进行加法运算，并将结果存储到新的符号名中。
4. 更新语法分析栈，将处理过的符号信息出栈，并将新的符号信息入栈。

```
        elif "EXPR" == production.left and production.right[0] == "EXPR":
            # EXPR -> EXPR ADD TERM
            symbol_expression2 = symbol_info_stack[-1]
            symbol_add = symbol_info_stack[-2]
            symbol_expression1 = symbol_info_stack[-3]
            symbol_pos = self.symbol_tables_[1].AddSymbol()
            symbol_name = self.symbol_tables_[1].GetSymbolName(symbol_pos)

            arg1_name = self.GetArgName(symbol_expression1.pos)
            arg2_name = self.GetArgName(symbol_expression2.pos)
            self.PrintQuadruples(
                Quadruples(
                    self.GetNextStateNum(),
                    symbol_add.txt_value,
                    arg1_name,
                    arg2_name,
                    symbol_name,
                )
            )

            conclude_symbol_info = GrammarSymbolInfo(
                symbol_name=production.left,
                txt_value=symbol_name,
                pos=SymbolPos(1, symbol_pos),
            )
            self.PopAndAppend(symbol_info_stack, conclude_symbol_info,
production)
```

这段代码处理形如 `ADD -> +` 或 `ADD -> -` 的语法规则。其主要逻辑包括：

1. 创建一个符号信息对象，用于表示加法操作符。
2. 将加法操作符的文本值添加到符号信息对象中。
3. 更新语法分析栈，将处理过的符号信息出栈，并将新的符号信息入栈。

```
        elif "ADD" == production.left:
            # ADD -> + || -
            conclude_symbol_info = GrammarSymbolInfo(
                symbol_name=production.left, txt_value=production.right[0]
            )
            self.PopAndAppend(symbol_info_stack, conclude_symbol_info,
production)
```

这段代码处理形如 `NEGA -> - FACTOR` 的语法规则。其主要逻辑包括：

1. 获取符号栈中的因子（FACTOR）的符号信息。
2. 为负数因子创建一个符号表条目，并生成一个新的符号名。
3. 生成中间代码，表示对因子进行取负操作，并将结果存储到新的符号名中。
4. 更新语法分析栈，将处理过的符号信息出栈，并将新的符号信息入栈。

```
        elif "NEGA" == production.left and production.right[0] == "-":
            # NEGA -> - FACTOR
            symbol_info = symbol_info_stack[-1]
            symbol_pos = self.symbol_tables_[1].AddSymbol(symbol_info.txt_value)
            symbol_name = self.symbol_tables_[1].GetSymbolName(symbol_pos)
            conclude_symbol_info = GrammarSymbolInfo(
                symbol_name=production.left,
                txt_value=symbol_name,
                pos=SymbolPos(1, symbol_pos),
            )

            self.PrintQuadruples(
                Quadruples(
                    self.GetNextStateNum(),
                    "nec",
                    symbol_info.txt_value,
                    "_",
                    symbol_name,
                )
            )
            self.PopAndAppend(symbol_info_stack, conclude_symbol_info,
production)
```

这段代码处理形如 `FACTOR -> id` 的语法规则。其主要逻辑包括：

1. 获取符号栈中的标识符（id）的符号信息。
2. 在符号表中查找该标识符的位置，并将位置信息添加到符号信息对象中。
3. 更新语法分析栈，将处理过的符号信息出栈，并将新的符号信息入栈。

```
        elif "FACTOR" == production.left and production.right[0] == "id":
            # FACTOR -> id
            symbol_info = symbol_info_stack[-1]
            symbol_pos =
self.symbol_tables_[0].FindSymbol(symbol_info.txt_value)
            conclude_symbol_info = GrammarSymbolInfo(
                symbol_name=production.left,
                txt_value=symbol_info.txt_value,
                pos=SymbolPos(0, symbol_pos),
            )
            self.PopAndAppend(symbol_info_stack, conclude_symbol_info,
production)
```

这段代码处理形如 `FACTOR -> number` 的语法规则。其主要逻辑包括：

1. 获取符号栈中的数字（number）的符号信息。
2. 为数字创建一个符号表条目，并生成一个新的符号名。
3. 生成中间代码，表示将数字赋值给新的符号名。
4. 更新语法分析栈，将处理过的符号信息出栈，并将新的符号信息入栈。

```
        elif "FACTOR" == production.left and production.right[0] == "number":
            # FACTOR -> number
            symbol_info = symbol_info_stack[-1]
            symbol_pos = self.symbol_tables_[1].AddSymbol(symbol_info.txt_value)
            symbol_name = self.symbol_tables_[1].GetSymbolName(symbol_pos)
            # 由规约后的产生式左部构造一个文法符号属性
            conclude_symbol_info = GrammarSymbolInfo(
                symbol_name=production.left,
                txt_value=symbol_name,
                pos=SymbolPos(1, symbol_pos),
            )

            self.PrintQuadruples(
                Quadruples(
                    self.GetNextStateNum(),
                    ":=",
                    symbol_info.txt_value,
                    "_",
                    symbol_name,
                )
            )
            self.PopAndAppend(symbol_info_stack, conclude_symbol_info,
production)
```

这段代码处理形如 `BOOL -> BOOL || JOIN` 的语法规则。其主要逻辑包括：

1. 获取符号栈中的两个布尔表达式（BOOL）和一个逻辑或操作符（||）的符号信息。
2. 为新的布尔表达式结果创建一个符号表条目，并生成一个新的符号名。
3. 生成中间代码，表示对两个布尔表达式进行逻辑或运算，并将结果存储到新的符号名中。
4. 更新语法分析栈，将处理过的符号信息出栈，并将新的符号信息入栈。

```
        elif "BOOL" == production.left and production.right[0] == "BOOL":
            # BOOL -> BOOL || JOIN
            symbol_expression2 = symbol_info_stack[-1]
            symbol_or = symbol_info_stack[-2]
            symbol_expression1 = symbol_info_stack[-3]
```

```
            symbol_pos = self.symbol_tables_[1].AddSymbol()
            symbol_name = self.symbol_tables_[1].GetSymbolName(symbol_pos)

            arg1_name = self.GetArgName(symbol_expression1.pos)
            arg2_name = self.GetArgName(symbol_expression2.pos)
            self.PrintQuadruples(
                Quadruples(
                    self.GetNextStateNum(),
                    symbol_or.txt_value,
                    arg1_name,
                    arg2_name,
                    symbol_name,
                )
            )

            conclude_symbol_info = GrammarSymbolInfo(
                symbol_name=production.left,
                txt_value=symbol_name,
                pos=SymbolPos(1, symbol_pos),
            )
            self.PopAndAppend(symbol_info_stack, conclude_symbol_info,
production)
```

这段代码处理形如 `NOT -> ! REL` 的语法规则。其主要逻辑包括：

1. 获取符号栈中的关系表达式（REL）的符号信息。
2. 为新的逻辑非表达式结果创建一个符号表条目，并生成一个新的符号名。
3. 生成中间代码，表示对关系表达式进行逻辑非运算，并将结果存储到新的符号名中。
4. 更新语法分析栈，将处理过的符号信息出栈，并将新的符号信息入栈。

```
        elif "NOT" == production.left and production.right[0] == "!":
            # NOT -> ! REL
            symbol_info = symbol_info_stack[-1]
            symbol_pos = self.symbol_tables_[1].AddSymbol()
            symbol_name = self.symbol_tables_[1].GetSymbolName(symbol_pos)
            arg1_name = self.GetArgName(symbol_expression1.pos)
            self.PrintQuadruples(
                Quadruples(self.GetNextStateNum(), "nec", arg1_name, "-",
symbol_name)
            )
            conclude_symbol_info = GrammarSymbolInfo(
                symbol_name=production.left,
                txt_value=symbol_name,
                pos=SymbolPos(1, symbol_pos),
            )
            self.PopAndAppend(symbol_info_stack, conclude_symbol_info,
production)
```

这段代码处理形如 `REL -> EXPR ROP EXPR` 的语法规则。其主要逻辑包括：

1. 获取符号栈中的两个表达式（EXPR）和一个关系操作符（ROP）的符号信息。
2. 为新的关系表达式结果创建一个符号表条目，并生成一个新的符号名。
3. 生成中间代码，表示对两个表达式进行关系运算，并将结果存储到新的符号名中。
4. 更新语法分析栈，将处理过的符号信息出栈，并将新的符号信息入栈。

```
        elif "REL" == production.left and production.right[0] == "EXPR":
```

```
            # REL -> EXPR ROP EXPR
            symbol_expression2 = symbol_info_stack[-1]
            symbol_ROP = symbol_info_stack[-2]
            symbol_expression1 = symbol_info_stack[-3]
            symbol_pos = self.symbol_tables_[1].AddSymbol()
            symbol_name = self.symbol_tables_[1].GetSymbolName(symbol_pos)

            arg1_name = self.GetArgName(symbol_expression1.pos)
            arg2_name = self.GetArgName(symbol_expression2.pos)
            self.PrintQuadruples(
                Quadruples(
                    self.GetNextStateNum(),
                    symbol_ROP.txt_value,
                    arg1_name,
                    arg2_name,
                    symbol_name,
                )
            )

            conclude_symbol_info = GrammarSymbolInfo(
                symbol_name=production.left,
                txt_value=symbol_name,
                pos=SymbolPos(1, symbol_pos),
            )
            self.PopAndAppend(symbol_info_stack, conclude_symbol_info,
production)
```

数据结构设计

- **SLR_OPERATIONS**：枚举，用于存储移进，规约和接受等操作类型。
- **TreeNode**：类，用于存储树结点的值和孩子结点。
- **LrItem**：类，存储LR项目，包括涉及的产生式和其中•的位置信息。
- **first_map, *follow_map***：字典，用于存储非终结符的first集和follow集。
- **state_sequence_stack_**：列表，状态栈。
- **move_conclude_string_stack_**：列表，符号栈。
- **tree_node_stack_**：列表，树结点栈。
- **normal_family_**：列表，项目集规范族。
- **action_goto_tables_**：集合，存储（状态，符号）对应的操作。

- **symbol_tables_**: 列表，用于存储符号表
- **quadruples_stack_**：列表，用于在if，while时临时存储将产生的中间代码
- **temp**：列表，存储if，while入口的位置
- **temp_else**：列表，存储else入口的位置

程序结构

```
class SLR_OPERATIONS(Enum): ...
class TreeNode: ...
class LrItem: ...
class SlrOperation: ...
class SymbolTable: ...
class Symbol: ...
class LexicalAnalyzer:
```

```
    def __init__(self):

class SyntacticAnalyzer:
    def __init__(self, show_detail=True):
    def IsNonTerminalSymbol(self, symbol):
    def GetProductionFirstSet(self, symbol_string):
    def GenProduction(self):
    def GenAugmentedGrammar(self):
    def GenFirstSet(self):
    def GenFollowSet(self):
    def GenGrammarSymbolSet(self):
    def GenLrItems(self):
    def GenItemClosureSet(self, input_item):
    def GenItemsClosureSet(self, items):
    def GenNormalFamilySet(self):
    def build_grammar(self):
    def PrintAnalysisProcess(self, step, sl_op ,get_word):
    def StartAnalize(self, code_filename):


class SemanticAnalyzer
    def __init__(self):
    def PopAndAppend(self, symbol_info_stack, conclude_symbol_info, production):
    def GetNextStateNum(self):
    def PeekNextStateNum(self):
    def GetArgName(self, sp, is_return=False):
    def PrintQuadruples(self, quadruples=None):
    def CreateSymbolTable(self, table_type, table_name):
    def ExecuteSemanticCheck(self, symbol_info_stack, production):

if __name__ == "__main__":
    num = int(input("请输入要分析的源程序编号[1-10]:"))
    file_name =f"./sourceProgram/sourceProgram{num}.txt"
    sa = SyntacticAnalyzer()
    if sa.StartAnalize(file_name):
        sa.VisualizeTree(
            sa.tree_node_stack_[0],
            f"./treeOutput/treeOutput{num}.txt"
        )
```

测试用例及结果

程序经过10组测试用例进行了充分测试，测试结果显示当程序符合语法规则时会输出完整的竖向语法分析树，当程序不符合语法规则时会抛出导致报错的位置和对应单词（9，10）。当程序不符合语义规则时会抛出出错原因（6）。

测试1

输入 sourceProgram1.txt：

```
//program 1: add two numbers.
{
    int a, b, c  ;
    a = 1;
    b = 2;
    c = a + b ;
}
```

输出 intermediate_code.txt：

```
1 (:=, 1, -, T0)
2 (:=, T0, -, a-Var)
3 (:=, 2, -, T1)
4 (:=, T1, -, b-Var)
5 (+, a-Var, b-Var, T2)
6 (:=, T2, -, c-Var)
```

测试2

输入 sourceProgram2.txt:

```
//program 2: do some calculation.
{
    int a, b, c  ;
    a = 5;      //positive number
    b = -3;     //negative number
    c = (a+b)*(a-b);//calculation
    write c  ;  //output:16.
}
```

输出 intermediate_code.txt：

```
1 (:=, 5, -, T0)
2 (:=, T0, -, a-Var)
3 (:=, 3, -, T1)
4 (nec, T1, -, T2)
5 (:=, T2, -, b-Var)
6 (+, a-Var, b-Var, T3)
7 (-, a-Var, b-Var, T4)
8 (*, T3, T4, T5)
9 (:=, T5, -, c-Var)
10 (write, -, -, c-Var)
```

测试3

输入 sourceProgram3.txt:

```
/*program 3: add numbers from 1 to 100
 *and print the result.
 */
{
    int a , sum ;
    bool  b ;
    a = 1 ;
    sum = 0 ;
    b := a <= 100 ;
    while b do
    {
        sum = sum + a ;
```

```
        a = a + 1 ;
        b := a <= 100 ;
    }
    write  sum  ;
}
```

输出 intermediate_code.txt：

```
1 (:=, 1, -, T0)
2 (:=, T0, -, a-Var)
3 (:=, 0, -, T1)
4 (:=, T1, -, sum-Var)
5 (:=, 100, -, T2)
6 (<=, a-Var, T2, T3)
7 (:=, T3, -, b-Var)
8 (j, b-Var, -, 10)
9 (j, -, -, 19)
10 (+, sum-Var, a-Var, T4)
11 (:=, T4, -, sum-Var)
12 (:=, 1, -, T5)
13 (+, a-Var, T5, T6)
14 (:=, T6, -, a-Var)
15 (:=, 100, -, T7)
16 (<=, a-Var, T7, T8)
17 (:=, T8, -, b-Var)
18 (j, -, -, 8)
19 (write, -, -, sum-Var)
```

测试4

输入 sourceProgram4.txt:

```
//program 4: input 3 numbers, find the largest
//one, and output it .
{
    int a,b,c;
    int lg;
    bool cond;

    read  a;    read  b;    read  c;

    cond := a > b  ;
    if  cond  then  lg = a ;
    else  lg = b ;

    cond := lg < c ;
    if  cond  then  lg = c ;

    write lg  ;
}
```

输出 intermediate_code.txt：

```
1 (read, -, -, a-Var)
2 (read, -, -, b-Var)
3 (read, -, -, c-Var)
4 (>, a-Var, b-Var, T0)
5 (:=, T0, -, cond-Var)
6 (j, cond-Var, -, 8)
7 (j, -, -, 10)
8 (:=, a-Var, -, lg-Var)
9 (j, -, -, 11)
10 (:=, b-Var, -, lg-Var)
11 (<, lg-Var, c-Var, T1)
12 (:=, T1, -, cond-Var)
13 (j, cond-Var, -, 15)
14 (j, -, -, 10)
15 (:=, c-Var, -, lg-Var)
16 (write, -, -, lg-Var)
```

测试5

输入 sourceProgram5.txt:

```
/* program 5: find all numbers which is
  divisible by 3  between 1 and 12 .
 */
{
    int  number, res ;
    bool  cond1, cond2  ;

    number = 1 ;
    cond1 := number <= 12 ;
    while cond1 do
    {
        res = number - ( number / 3 ) * 3 ;
        cond2 := res == 0  ;
        if cond2 then write number ;
        number = number + 1 ;
        cond1 := number <= 12 ;
    }
}
```

输出 intermediate_code.txt：

```
1 (:=, 1, -, T0)
2 (:=, T0, -, number-Var)
3 (:=, 12, -, T1)
4 (<=, number-Var, T1, T2)
5 (:=, T2, -, cond1-Var)
6 (j, cond1-Var, -, 8)
7 (j, -, -, 27)
8 (:=, 3, -, T3)
9 (/, number-Var, T3, T4)
10 (:=, 3, -, T5)
11 (*, T4, T5, T6)
12 (-, number-Var, T6, T7)
13 (:=, T7, -, res-Var)
14 (:=, 0, -, T8)
15 (==, res-Var, T8, T9)
16 (:=, T9, -, cond2-Var)
17 (j, cond2-Var, -, 19)
18 (j, -, -, 1)
19 (write, -, -, number-Var)
20 (:=, 1, -, T10)
21 (+, number-Var, T10, T11)
22 (:=, T11, -, number-Var)
23 (:=, 12, -, T12)
24 (<=, number-Var, T12, T13)
25 (:=, T13, -, cond1-Var)
26 (j, -, -, 6)
```

测试6

语义错误，标识符未声明

输入 sourceProgram6.txt:

```
//program 6: no decls in program .
//NO errors in Syntax Checking.
//Undeclared id errors should be reported in Symantic Checking.
{
    //int a; bool b;
    a = -1 ;
    b := a <= 0;
    write a;
}
```

输出报错：

```
请输入要分析的源程序编号[1-10]:6
语义错误！！！ a 没有定义！
```

测试7

输入 sourceProgram7.txt:

```
//program 7: read  in two numbers and do some calculations.
{
    int a, b, c  ;
    read a ;
    read b ;

    //Test Expr().
    a = a + 1 + 2 * 3 * 4 ;
    b = b * 8 / 2 / 4 ;
    c = ( a + b ) * ( a - b ) ;

    write c  ;
}
```

输出 intermediate_code.txt：

```
1 (read, -, -, a-Var)
2 (read, -, -, b-Var)
3 (:=, 1, -, T0)
4 (+, a-Var, T0, T1)
5 (:=, 2, -, T2)
6 (:=, 3, -, T3)
7 (*, T2, T3, T4)
8 (:=, 4, -, T5)
9 (*, T4, T5, T6)
10 (+, T1, T6, T7)
11 (:=, T7, -, a-Var)
12 (:=, 8, -, T8)
13 (*, b-Var, T8, T9)
14 (:=, 2, -, T10)
15 (/, T9, T10, T11)
16 (:=, 4, -, T12)
17 (/, T11, T12, T13)
18 (:=, T13, -, b-Var)
19 (+, a-Var, b-Var, T14)
20 (-, a-Var, b-Var, T15)
21 (*, T14, T15, T16)
22 (:=, T16, -, c-Var)
23 (write, -, -, c-Var)
```

测试8

输入 sourceProgram8.txt:

```
/* program8: input three numbers ,output the largest one.
 * Test Statement : If_then and If_Then_Else .
 */
{
    int a,b,c ;
    bool cond1,cond2,cond3;

    read  a;  read  b;  read  c;

    cond1 := a >= b ;
    cond2 := a >= c ;
    cond3 := b >= c ;

    if cond1 then
        if cond2 then write a ;
        else write c ;
```

```
    cond1 := a < b ;
    if cond1 then
        if cond3 then write b ;
        else write c ;
}
```

输出 intermediate_code.txt：

```
1 (read, -, -, a-Var)
2 (read, -, -, b-Var)
3 (read, -, -, c-Var)
4 (>=, a-Var, b-Var, T0)
5 (:=, T0, -, cond1-Var)
6 (>=, a-Var, c-Var, T1)
7 (:=, T1, -, cond2-Var)
8 (>=, b-Var, c-Var, T2)
9 (:=, T2, -, cond3-Var)
10 (j, cond1-Var, -, 12)
11 (j, -, -, 1)
12 (j, cond2-Var, -, 14)
13 (j, -, -, 16)
14 (write, -, -, a-Var)
15 (j, -, -, 17)
16 (write, -, -, c-Var)
17 (<, a-Var, b-Var, T3)
18 (:=, T3, -, cond1-Var)
19 (j, cond1-Var, -, 21)
20 (j, -, -, 1)
21 (j, cond3-Var, -, 23)
22 (j, -, -, 25)
23 (write, -, -, b-Var)
24 (j, -, -, 26)
25 (write, -, -, c-Var)
```

测试9

错误测试，空程序不符合语法规则，语法分析器会抛出报错单词。

输入 sourceProgram9.txt:

```
//program 9: empty program.
{

}
```

输出报错：

```
语法分析器过程中，发生错误!
}       行号:_ 4
state:  1  与  }  在action_goto_table 中不含对应操作!
```

测试10

错误测试，根据语法规则有"&&"的左右两边都必须接表达式非单独的id，例如本例中第九行的flag。

输入 sourceProgram10.txt:

```
//program10: test prime numbers.
{
    int number,i,j;
    bool cond,flag;
```

```
    read number;

    i = 2;
    flag = true;
    cond := i<number && flag;

    while cond do
    {
        j=number-(number/i)*i;
        flag:= j!=0;
        i=i+1;
        cond := i<number && flag;
    }
    if flag then write number;
}
```

输出报错：

```
语法分析器过程中，发生错误!
;       行号:  9
state:  36  与  ;  在action_goto_table 中不含对应操作!
```