



南京理工大学  
NANJING UNIVERSITY OF SCIENCE & TECHNOLOGY

## 南京理工大学计算机科学与工程学院

### 软件课程设计（Ⅱ）报告

班    级\_\_\_\_\_9191069501\_\_\_\_\_

学生姓名\_\_\_\_\_王子辉\_\_\_\_\_

学    号\_\_\_\_\_9191160D0236\_\_\_\_\_

指导教师\_\_\_\_\_王永利\_\_\_\_\_

# 一、程序及系统环境介绍

## 1、系统环境与编程语言

系统环境：Windows10

编程语言：Python 3.8.3

IDE：PyCharm Community Edition 2021.2.3

## 2、选择 python 的原因

由于本人就读于智能专业，平时很多代码基本上都是用 python 来写，对使用 python 的 IDE 十分的熟悉，以后的研究生生活也基本上是和 python 打交道，再加上 python 语言本身就易于读写，并且也提供了面向对象的方法，相较于程序语言、语法，python 更侧重于解决问题本身，因此这次选择利用 python 完成此次的软件课程设计。

## 3、程序介绍

《编译原理》是计算机专业的一门重要的专业课程，其中包含大量软件设计细想。通过课程设计，实现一些重要的算法，或设计一个完整的编译程序，能够进一步加深和掌握所学知识，提高自己的代码工程能力，对个人以后的发展具有重要而深远的意义。

# 二、词法分析

## 1、词法分析概述

词法分析的任务是利用输入的词法配置文件，将输入代码进行分词，并对分好的 token 作行号标记和类型判别。

## 2、词法分析步骤

### （1）读入配置文件并进行预处理

利用 json 库将.json 格式的配置文件读入，调用 load()函数，返回一个字典。配置文件内容如下图：



## (2) 代码文件预处理

在代码文件中，需要先把注释去掉再进行后续分析，需要去掉注释；但是注释也会占据代码行，因此需要记录代码所在行数。注释分为单行注释和多行注释。对于单行注释，只需利用单行注释的正则表达式去匹配即可消除；对于多行注释，需要多行读入拼接，并用注释的正则表达式去匹配即可消除。具体实现如下图：

```
# 去掉单行注释
if note_pattern.match(line.strip(" ")):
    continue

# 去掉多行注释
for i in range(len(code_list0)):
    if code_list0[i]['code_val'] == "/**":
        temp = code_list0[i]['code_val']
        code_list0[i]['code_line'] = 0
        for j in range(i + 1, len(code_list0)):
            if note_pattern.match(temp):
                code_list0[j]['code_line'] = 0
                break
            temp += code_list0[j]['code_val']
            code_list0[j]['code_line'] = 0
```

## (3) 分词产生 token 列表

分词工作只需要确定代码段中的停顿标志和停用词即可。有下述若干种停顿标志与停用词：

### ① 空格：

在代码段中（字符串常量内除外），空格是唯一的停顿标志；词法分析过程中，当读入空格时，需要将前面的 token 装入 token 列表内，并开始读取新的 token。具体实现如下：

```
# 遇到空格
if c == " ":
    if token != '':
        token_list.append(token)
        line_list.append(code['code_line'])
    token = ""
```

### ② 符号

配置文件中的 SYMBOL 列表内，除了小数点“.”以外，其余符号在所有情况下都是停用词（字符串常量内除外）。词法分析过程中，当读入这些符号时，将前面的 token 装入 token 列表中，并将该符号单独装入 token 列表中，然后再开始下个 token 的读取。具体实现如下：

```
# 停用词表
stop_list = self.symbol_list
stop_list.remove(".")
```

```
# 确定停顿词
elif c in stop_list:
    if token != '':
        token_list.append(token)
        line_list.append(code['code_line'])
    token_list.append(c)
    line_list.append(code['code_line'])
    token = ""
```

### ③运算符

对于配置文件中的运算符而言，需要分三种情况处理（字符串常量内除外）。其一，单独出现一个单独的运算符，左边无“E”（科学计数法的标志），右边无运算符“此为双目运算符”，则作为停用词进行分析；其二：运算符左边有“E”，则继续读取 token；其三：运算符右边有运算符，则将前面的 token 放入 token 列表中，此运算符与后运算符两者拼接作为一个新的 token 放入 token 列表中，然后再开始下个 token 的读取。具体实现如下：

```
# 处理双目运算符
elif c in self.op_list and code['code_val'][i + 1] in self.op_list:
    cc = c + code['code_val'][i + 1]
    if token != '':
        token_list.append(token)
        line_list.append(code['code_line'])
    token_list.append(cc)
    line_list.append(code['code_line'])
    token = ""
    i += 1

# 处理科学计数法的数据表示
elif c in self.op_list and code['code_val'][i - 1] == 'E':
    token += c
    i += 1
    continue

# 仅仅只有一个运算符却没有被空格分开的公式的情况
elif c in self.op_list:
    if token != '':
        token_list.append(token)
        line_list.append(code['code_line'])
    token_list.append(c)
    line_list.append(code['code_line'])
    token = ""
```

#### ④字符串常量

以上三种情况的处理均为非字符串常量状态下的分词处理。当遇见字符串常量的标志“`"`”时，需要对该行进行一个字符串常量状态的标注，即该行目前已进入字符串常量的读取状态，除非遇见与该“`"`”对应的另一个“`"`”，否则不作任何处理，继续 token 读入。如果一行结束也未遇见下一个“`"`”，则将前面的 token 放入 token 列表并开始下个 token 的读入，出错分析由上述 `get_token_type()`方法实现。具体实现如下：

```
# 根据双引号判断是否为字符串常量
is_string = False
i = 0
while i < len(code['code_val']):
    c = code['code_val'][i]
    # 判断是否进入字符串常量状态
    if c == '"':
        if is_string == False:
            # 进入字符串常量状态
            is_string = True
        else:
            # 退出字符串常量状态
            is_string = False
```

```
# 如果此时是字符串常量状态，则一直向token中加入新字符
else:
    token += c
    # 如果一行结束还未遇到'"', 则将识别出的token加入token列表，其类型为ERROR
    if i == len(code['code_val']) - 1:
        token_list.append(token)
        line_list.append(code['code_line'])
```

#### (4) 标注类型和行号

根据前面记录下的各 token 所在的行数以及类型判断函数，进行词法分析的最后一步处理，生成 token 分析表。一旦检测出 ERROR 类型，即标注相应 token 及对应行号，并终止词法分析。具体实现如下：

```
token_dict_list = []
for i in range(len(token_list)):
    token_dict_list.append(
        {'val': token_list[i], 'type': self.get_token_type(token_list[i]), 'line': line_list[i]})
    if self.get_token_type(token_list[i]) == 'ERROR':
        self.is_lexer_error = True
        self.error_line = line_list[i]
        self.error_token = token_list[i]
        break
return token_dict_list
```

### 3、词法分析结果

词法分析部分结果如下图：

1	{'val': 'function', 'type': 'KEYWORD', 'line': 1}
2	{'val': 'void', 'type': 'KEYWORD', 'line': 1}
3	{'val': 'main', 'type': 'ID', 'line': 1}
4	{'val': '(', 'type': 'SYMBOL', 'line': 1}
5	{'val': ')', 'type': 'SYMBOL', 'line': 1}
6	{'val': '{', 'type': 'SYMBOL', 'line': 1}
7	{'val': 'int', 'type': 'KEYWORD', 'line': 4}
8	{'val': 'a', 'type': 'ID', 'line': 4}
9	{'val': ',', 'type': 'SYMBOL', 'line': 4}
10	{'val': 'dd', 'type': 'ID', 'line': 4}
11	{'val': '=', 'type': 'OP', 'line': 4}
12	{'val': '12', 'type': 'CONSTANT', 'line': 4}
13	{'val': ';', 'type': 'SYMBOL', 'line': 4}
14	{'val': 'int', 'type': 'KEYWORD', 'line': 5}
15	{'val': 'b', 'type': 'ID', 'line': 5}
16	{'val': '=', 'type': 'OP', 'line': 5}
17	{'val': '14', 'type': 'CONSTANT', 'line': 5}
18	{'val': ';', 'type': 'SYMBOL', 'line': 5}
19	{'val': 'double', 'type': 'KEYWORD', 'line': 6}
20	{'val': 'c', 'type': 'ID', 'line': 6}
21	{'val': '=', 'type': 'OP', 'line': 6}
22	{'val': 'a', 'type': 'ID', 'line': 6}
23	{'val': '--', 'type': 'OP', 'line': 6}
24	{'val': '+', 'type': 'OP', 'line': 6}
25	{'val': 'b', 'type': 'ID', 'line': 6}

## 三、语法分析

### 1、语法分析概述

语法分析的任务是根据输入的配置文件（已知配置文件符合 LR（1）文法规则），构造 LR（1）分析表，并利用该分析表对输入待分析的代码段进行语法分析，从而判断该代码段是否能被接受，输出 “Yes!” 或者 “No!”。

### 2、语法分析步骤

（1）生成 LR（1）分析表

生成 LR（1）分析表需要经过以下若干个子步骤进行：

### ①生成非终结符的 FIRST 集

首先确定配置文件中的所有非终结符。观察发现，给定文法产生式的所有左部即为该文法中的所有非终结符。将所有左部装进非终结符列表中，那么其他符号便都是终结符。已知文法产生式中有很多非终结符能推导出空字符，因此将这些可以推导出空字符的非终结符标记为“space\_nonTerminator”，具体做法是将这些非终结符装进 space\_nonTerminator\_list 中。

已知所有终结符的 FIRST 集中仅有其本身一个元素，若要求非终结符的 FIRST 集，则需根据文法产生式进行深度优先搜索，直到检索到终结符。由此可以利用递归策略求出所有非终结符的 FIRST 集，其中若检索到非终结符，则判断该非终结符是否在上述可推导至空字符的非终结符列表中，若不在则进行下一层搜索，并在递归返回该地址时，返回先前产生的 first\_list；若在其中则依旧进行下一层搜索，但在递归返回该地址时继续检索该非终结符的下个字符。值得一提的是，每次检索到一个非终结符时，都需要将其标记为“visited”，即将其装进 visited\_nonTerminator\_list 中，以此避免死递归。具体实现如下：

```
def get_first(self, s):
    # 避免死递归
    if s in self.visited_nonTerminator_list:
        return []
    if s in self.a.nonTerminator_list:
        self.visited_nonTerminator_list.append(s)
    # 递归边界
    if s in self.a.terminator_list:
        return [s]
    first_list = []
    for pro_right in self.a.production_map[s]:
        for word in pro_right:
            if word not in self.a.space_nonTerminator:
                first_list += self.get_first(word)
                break
            first_list += self.get_first(word)
    return first_list
```



求出的 FIRST 集部分结果如下图：

```
START : {'public', 'function', 'protected', 'class', 'private'}
S : {'public', 'function', 'protected', 'class', 'private'}
CLASS_S : {'public', 'class', 'protected', 'private'}
FUNC_S : {'function'}
FIELD_TYPE : {'public', 'protected', 'private'}
OPTIONAL_FIELD_TYPE : {'final', 'static'}
TYPEDEF : {'float', 'short', 'int', 'void', 'double', 'long', 'var', 'bool', 'ID', 'char'}
CONST_TYPE : {'float', 'short', 'int', 'void', 'long', 'var', 'bool', 'double', 'char'}
DECLARE_INTER : {'float', 'short', 'int', 'void', 'double', 'long', 'var', 'bool', 'ID', 'char'}
DECLARE_CLASS : {'static', 'float', 'short', 'char', 'int', 'void', 'double', 'public', 'long', 'var'}
METHOD_CLASS : {'static', 'float', 'short', 'char', 'int', 'void', 'double', 'public', 'long', 'var'}
DECLARE_INIT : {'='}
DECLARE_VARS : {'', ''}
ARGS : {'float', 'short', 'int', 'void', 'double', 'long', 'var', 'bool', 'ID', 'char'}
ARG : {'', ''}
EXPRESSION : {'ID', '--', 'CONSTANT', '(', '++'}
VALUE : {'--', 'ID', '++', 'CONSTANT'}
OPERATION_SPECIAL : {'--', '++'}
OPERATION : {'+', '=', '+=', '^', '>', '~', '*', '/', '+', '%=', '&', '|', '!', '-', '<=', '<', '>='}
OPERATION_CAL : {'-', '!', '~', '*', '/', '^', '&', '+', '%', '|'}
OPERATION_EQ : {'=', '*=', '+=', '-=', '%=', '/='}
OPERATION_CMP : {'==', '<=', '<', '>=', '!=', '>'}
OPERATION_LOG : {'!', '||', '&&'}
IF_STMT : {'if'}
ELSE_IF_STMT : {'else'}
FOR_STMT : {'for'}
```

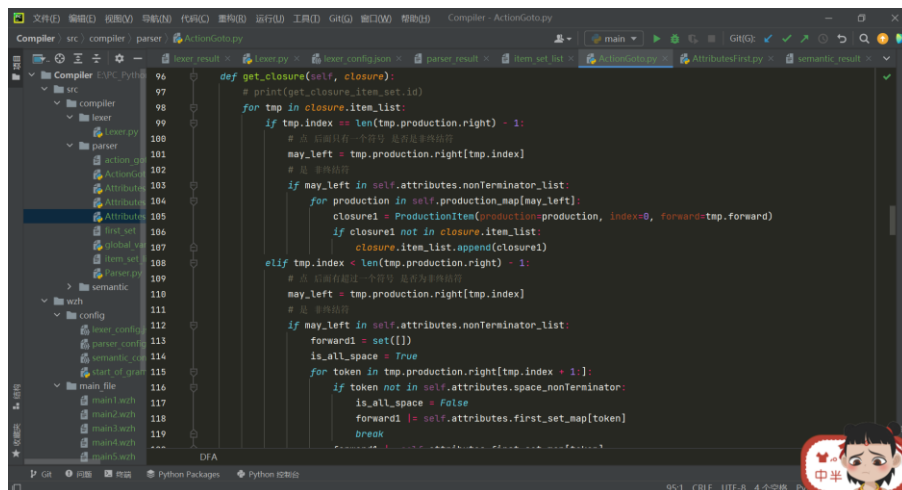
## ②生成 LR (1) 项目集族和 LR (1) 分析表

在生成 LR (1) 这一步中，需要封装较多的数据结构，比如 Production（产生式）、ProductionItem（产生式项目）、Closure（项目集）以及 DFA（有限自动机，即项目集族）。由产生式和 FIRST 集生成产生式项目，由一个个产生式项目封装成一个个项目集，再根据移进-规约的规则利用广度优先搜索生成一个个新的项目集，直到项目集数量不再变化便生成一个完整的项目集族。其中产生式、产生式项目以及项目集均需要重写对象的 `__eq()` 方法，以此判断上述三种数据结果在生成项目集族的过程中是否出现重复。

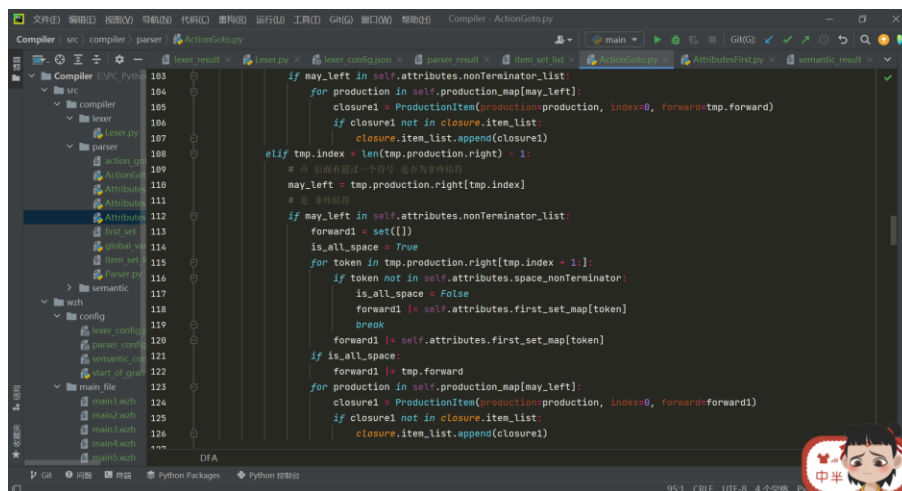
为了产生完整的项目集族，需要定义两种方法：`get_closure()` 和 `search_forward()`，前者为根据初始产生式项目生成一个完整的项目集，后者为依照给定的项目集向前搜索，从而建立起项目集之间的联系。

生成项目集的方法实现如下（其中会用到先前定义的 FIRST 集）：首先，我们读取该项目里面的产生式，对于每一个产生式，从头到尾处理（项目数量会发生动态变化）。如果点后（用 `index` 标记点下标）仅一个符号，则判断其是否为非终结符，如果是非终结符，就把左部为该非终结符的产生式列表加入项目，并加入上一个的 FIRST 集合，如果该项目中无这一条，则添加；如果点后超过一个符号，则同样判断其是否为非终结符，如果是非终结符，就把左部为该非终结符的产生式列表加入项目，并查找该非终结符后符号的 FIRST 集合并累加直至后符号不为前述的“`space_nonTerminator`”，如果项目中没这一条，则添加。

具体代码实现如下图：



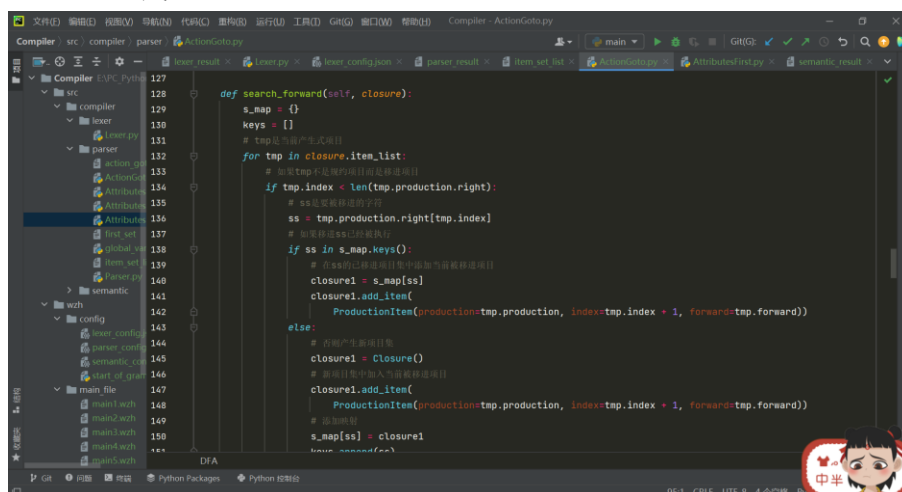
```
def get_closure(self, closure):
    # print(get_closure_item_set_id)
    for tmp in closure.item_list:
        if tmp.index == len(tmp.production.right) - 1:
            # 点 后面只有一个符号 是否为非终结符
            may_left = tmp.production.right[tmp.index]
            # 是非终结符
            if may_left in self.attributes.nonTerminator_list:
                for production in self.production_map[may_left]:
                    closure1 = ProductionItem(production=production, index=0, forward=tmp.forward)
                    if closure1 not in closure.item_list:
                        closure.item_list.append(closure1)
        elif tmp.index < len(tmp.production.right) - 1:
            # 点 后面有超过一个符号 是否为非终结符
            may_left = tmp.production.right[tmp.index]
            # 是非终结符
            if may_left in self.attributes.nonTerminator_list:
                forward = set({})
                is_all_space = True
                for token in tmp.production.right[tmp.index + 1:]:
                    if token not in self.attributes.space_nonTerminator:
                        is_all_space = False
                        forward1 = self.attributes.first_set_map[token]
                        break
                forward1 = self.attributes.first_set_map[token]
                if is_all_space:
                    forward1 = tmp.forward
                for production in self.production_map[may_left]:
                    closure1 = ProductionItem(production=production, index=0, forward=forward1)
                    if closure1 not in closure.item_list:
                        closure.item_list.append(closure1)
```



```
if may_left in self.attributes.nonTerminator_list:
    for production in self.production_map[may_left]:
        closure1 = ProductionItem(production=production, index=0, forward=tmp.forward)
        if closure1 not in closure.item_list:
            closure.item_list.append(closure1)
elif tmp.index < len(tmp.production.right) - 1:
    # 点 后面有超过一个符号 是否为非终结符
    may_left = tmp.production.right[tmp.index]
    # 是非终结符
    if may_left in self.attributes.nonTerminator_list:
        forward1 = set({})
        is_all_space = True
        for token in tmp.production.right[tmp.index + 1:]:
            if token not in self.attributes.space_nonTerminator:
                is_all_space = False
                forward1 = self.attributes.first_set_map[token]
                break
        forward1 = self.attributes.first_set_map[token]
        if is_all_space:
            forward1 = tmp.forward
        for production in self.production_map[may_left]:
            closure1 = ProductionItem(production=production, index=0, forward=forward1)
            if closure1 not in closure.item_list:
                closure.item_list.append(closure1)
```

向前搜索的方法实现如下：首先我们取出项目中的产生式，对每一个产生式执行如下操作：若点后符号则查看是否已创建新状态，若已创建则添加该项目，若没有则新建一个状态再进行添加；结束上述步骤后，还需查找是否有一样的项目集（具体实现为重写\_\_eq\_\_方法），然后去掉重复的项目。在去重之前，还需计算一次闭包。

具体代码实现如下图：




```
def search_forward(self, closure):
    s_map = {}
    keys = {}
    # tmp 是当产生式项目
    for tmp in closure.item_list:
        # 如果tmp不是规约项目则是移进项目
        if tmp.index < len(tmp.production.right):
            ss = tmp.production.right[tmp.index]
            # ss是要被移进的字串
            if ss in s_map.keys():
                # 在ss的已移进项目和中添加当前被移进项目
                closure1 = s_map[ss]
                closure1.add_item(
                    ProductionItem(production=tmp.production, index=tmp.index + 1, forward=tmp.forward))
            else:
                # 否则产生新项目
                closure1 = Closure()
                # 新项目是中添加当前被移进项目
                closure1.add_item(
                    ProductionItem(production=tmp.production, index=tmp.index + 1, forward=tmp.forward))
                # 添加映射
                s_map[ss] = closure1
    return s_map[ss]
```



### ③根据项目集族生成 LR（1）分析表

对于项目集族中的每个项目（即每个状态），生成其对应的 Action 和 Goto 映射，即遇见何种终结符执行何种 Action，遇见何种非终结符执行何种 Goto，从而生成 LR（1）分析表，用来进行对输入代码段的匹配。求得的 LR（1）分析表（ActionGoto 表）部分结果如下图：

```
0:Action: {'function': 5, 'public': 6, 'private': 7, 'protected': 8, '#': 'S -> []', 'class': 'FIE ✓
1:Action: {'#': 'acc'}, Goto: {}
2:Action: {'function': 5, 'public': 6, 'private': 7, 'protected': 8, '#': 'S -> []', 'class': 'FIELD_
3:Action: {'function': 5, 'public': 6, 'private': 7, 'protected': 8, '#': 'S -> []', 'class': 'FIELD_
4:Action: {'class': 11}, Goto: {}
5:Action: {'ID': 13, 'void': 15, 'char': 16, 'bool': 17, 'short': 18, 'int': 19, 'long': 20, 'double'
6:Action: {'class': "FIELD_TYPE -> ['public']"}, Goto: {}
7:Action: {'class': "FIELD_TYPE -> ['private']"}, Goto: {}
8:Action: {'class': "FIELD_TYPE -> ['protected']"}, Goto: {}
9:Action: {'#': "S -> ['CLASS_S', 'S']"}, Goto: {}
10:Action: {'#': "S -> ['FUNC_S', 'S']"}, Goto: {}
11:Action: {'ID': 24}, Goto: {}
12:Action: {'ID': 25}, Goto: {}
13:Action: {'ID': "TYPEDEF -> ['ID']"}, Goto: {}
14:Action: {'ID': "TYPEDEF -> ['CONST_TYPE']"}, Goto: {}
15:Action: {'ID': "CONST_TYPE -> ['void']"}, Goto: {}
16:Action: {'ID': "CONST_TYPE -> ['char']"}, Goto: {}
17:Action: {'ID': "CONST_TYPE -> ['bool']"}, Goto: {}
18:Action: {'ID': "CONST_TYPE -> ['short']"}, Goto: {}
19:Action: {'ID': "CONST_TYPE -> ['int']"}, Goto: {}
20:Action: {'ID': "CONST_TYPE -> ['long']"}, Goto: {}
21:Action: {'ID': "CONST_TYPE -> ['double']"}, Goto: {}
22:Action: {'ID': "CONST_TYPE -> ['float']"}, Goto: {}
23:Action: {'ID': "CONST_TYPE -> ['var']"}, Goto: {}
24:Action: {'ID': 26}, Goto: {}
25:Action: {'ID': 27}, Goto: {}
```



### ④执行 LR（1）分析

由上得到的 LR（1）分析表，我们可以对输入代码段执行 LR（1）分析（前提是不出现任何词法错误）。首先初始化状态栈为 0，随后看一下目前的状态和面临的符号，进行移进或规约操作。若为移进操作，则状态栈移入状态，符号栈移入符号；若为规约操作，则弹出状态并用产生式进行规约。以上分析持续进行，直至进入“acc”（接受）状态。如果找不到面临符号对应的操作，则终止分析，并输出发生语法错误的代码行号以及需要改动的地方。具体实现代码如下图：

```
# 判断是否出错
def is_error(self, state, input_token):
    return input_token not in self.ag.action_list[state].keys()

# 接受
elif action == 'acc':
    return True
```

```

# S{action}. 移进
if type(action) == int:
    state_stack.append(action)
    token_stack.append(input_token)
    self.input_token_list.pop(0)

# 出错词
self.error_token = self.input_val_line_list[0]['val']
# 出错行
self.error_line = self.input_val_line_list[0]['line']

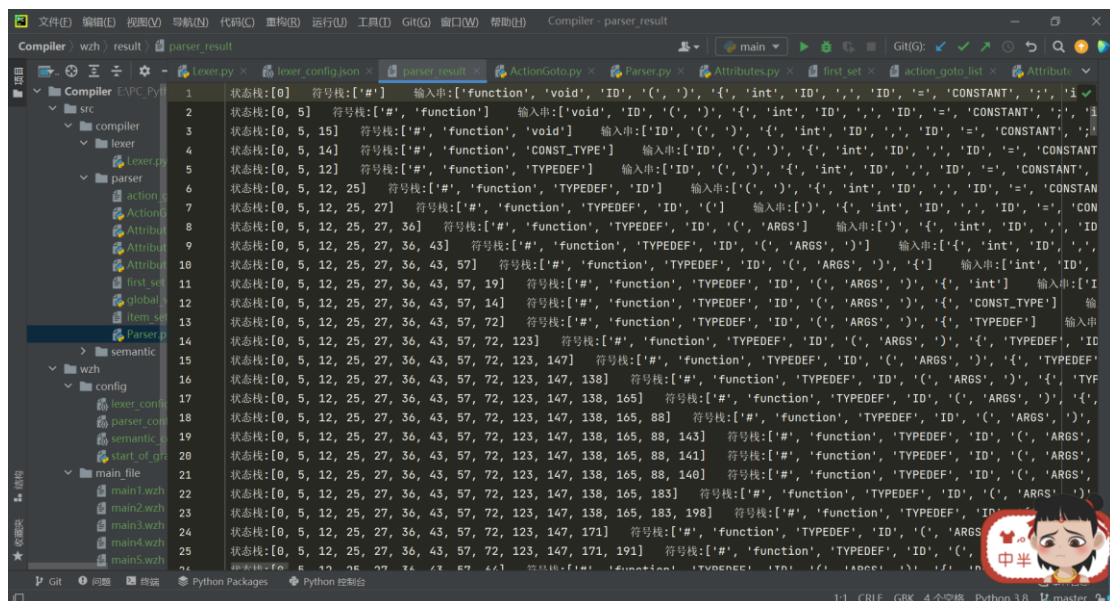
self.input_val_line_list.pop(0)

# r{x}. 规约
elif type(action) == Production:
    len_pop = len(action.right)
    for i in range(len_pop):
        state_stack.pop(-1)
        token_stack.pop(-1)
    token_stack.append(action.left)
    state_stack.append(self.ag.goto_list[state_stack[-1]][action.left])

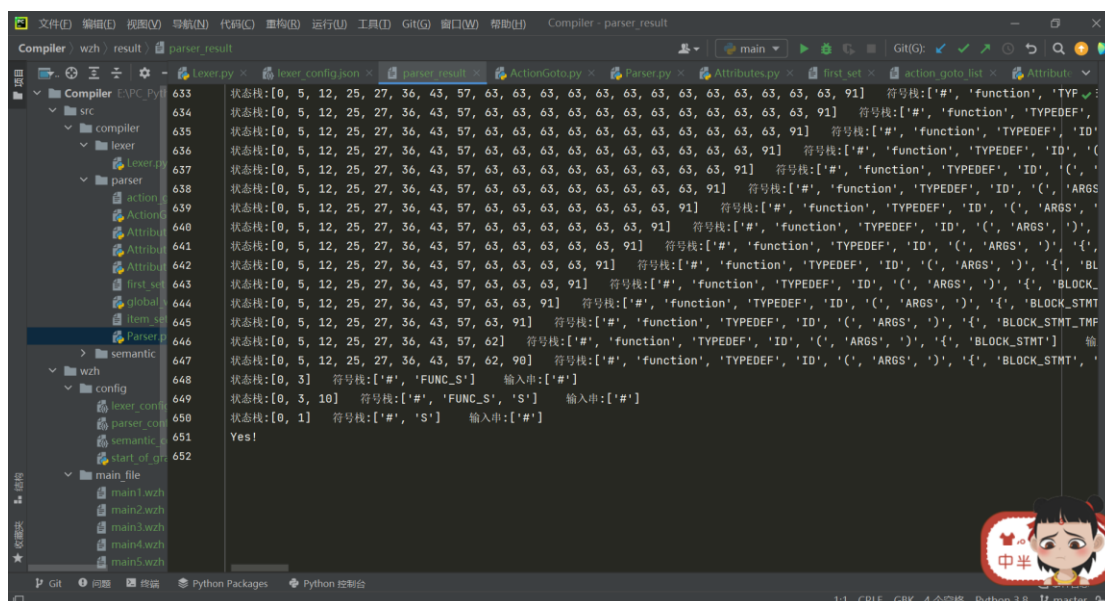
```

### 3、语法分析结果

语法分析部分结果如下图：







#### 四、语义分析

语法分析即在语法分析的配置文件上稍作改动，增加多个增广文法，利用该配置文件对输入代码段进行语义分析。本次课程设计仅局限于可以输出代码段的四元式（TAC 语句）。

## 2、语义分析步骤

### (1) 分析运算式、赋值语句及变量声明语句

具体做法是根据先前定义好的算符优先级，将运算式转成后缀表达式，算符优先级如下图：

```
# 运算优先级映射：“非关与或赋”
op_map = {'!': 0, '||': 1, '&&': 2, '==': 2.5, '!=': 2.5, '<=': 2.5, '>=': 2.5, '>': 2.5, '<': 2.5, '+': 3, '-': 3,
          '*': 4, '/': 4, '%': 4}
```

有了优先级，我们便可将原运算式（中缀表达式）转换为后缀表达式。首先，我们设立一个操作符栈，用以临时存放操作符；设立一个列表，用以存放后缀表达式。随后，我们从左至右扫描原运算式，如果碰到操作数（ID、CONSTANT），就把操作数加入后缀表达式中。如果碰到操作符 **op**，就将其优先级与操作符栈的栈顶操作符优先级比较：若 **op** 的优先级高于栈顶操作符的优先级，则压入操作符栈；若 **op** 的优先级低于或等于栈顶操作符的优先级，则将操作符栈的操作符不断弹出到后缀表达式中，直到 **op** 的优先级高于栈顶操作符的优先级。重复以上操作，直到原运算式扫描完毕，之后若操作符栈中仍有元素，则将它们依次弹出至后缀表达式中。值得一提的是，这里引入了括号的操作，具体处理为：如果遇见左括号“（”，就压入操作符栈；如果是右括号“）”，就把操作符栈里的元素不断弹出到后缀表达式中直到碰到左括号“（”。具体实现如下：

```

for token in token_list:
    if token['type'] == 'ID' or token['type'] == 'CONSTANT':
        LRD_stack.append(token)
    elif token['val'] in op_map.keys():
        op = token
        while 1:
            if len(op_stack) == 0 or op_stack[-1]['val'] == '(' or op_map[op['val']] > op_map[
                op_stack[-1]['val']]:
                break
            LRD_stack.append(op_stack[-1])
            op_stack.pop(-1)
        op_stack.append(op)
    elif token['val'] == '(':
        op_stack.append(token)
    elif token['val'] == ')':
        while 1:
            if op_stack[-1]['val'] != '(':
                LRD_stack.append(op_stack[-1])
                op_stack.pop(-1)
            else:
                op_stack.pop(-1)
                break

```

有了后缀表达式，我们便可进行运算式的语义分析。具体做法为：从左到右扫描后缀表达式，如果是操作数（ID、CONSTANT），就压入栈；如果是操作符，就连续弹出两个操作数（后弹出的是第一操作数，先弹出的是第二操作数），然后进行操作符的操作，生成的中间结果压入栈中。如此反复直到后缀表达式扫描完毕，这时栈中只会存在一个数，记为 result。具体实现如下：

```

for token in LRD_stack:
    # print(token)
    if token['type'] == 'ID' or token['type'] == 'CONSTANT':
        var_stack.append(token['val'])

    elif token['type'] == 'OP':
        var2 = var_stack[-1]
        var_stack.pop(-1)
        var1 = var_stack[-1]
        var_stack.pop(-1)
        process_var = "T" + str(gl.global_var_cnt)
        if token['val'] == '=':
            tac = TAC((token['val'], var2, '-', var1))
        else:
            tac = TAC((token['val'], var1, var2, process_var))
            gl.global_var_cnt += 1
        result.append(tac)
        # print(tac)
        var_stack.append(process_var)

```

前述运算式的 result 在形如 “a=<运算式>” 的赋值语句中可生成(‘=’,result, ‘-’,a)这样的 TAC 语句。若是变量声明语句，即在之前加上(<变量类型>,<变量>, ‘-’, ‘-’)即可。

关于双目运算符的处理如下：

① ‘++’ 或 ‘--’：

i) 单独出现：形如 “a++”，则作赋值语句 “T=a+1” 和 “a=T” 的操作；形如 “++a” 与前述完全一致。 ‘--’同理。

ii) 出现在运算式中：形如 “a+++b”，则运算式作 “a+b” 处理，待运算式处理完毕后加入赋值语句 “T=a+1” 和 “a=T” 的操作；形如 “++a+b”，则先做 “T=a+1” 和 “a=T” 的操作，再对运算式作 “a+b” 的操作。 ‘--’同理。

② ‘+=’ 或 ‘-=’：

该类双目运算符仅表示赋值语句。形如 “a+=3”，则作赋值语句 “T=a+3” 和 “a=T” 的操作。 ‘-=’ 同理。

具体实现如下：

```
if marked_op == '+= ' or marked_op == '-=':
    result.append(TAC((marked_op[0], var0, token_list[2]['val'], process_var)))
    result.append(TAC(('=', process_var, '-', var0)))
    gl.global_var_cnt += 1
    # result.append((marked_op[0], var0, token_list[2]['val'], var0))
elif marked_op == '++ ' or marked_op == '-- ':
    # result.append((marked_op[0], var0, 1, var0))
    result.append(TAC((marked_op[0], var0, 1, process_var)))
    result.append(TAC(('=', process_var, '-', var0)))
    gl.global_var_cnt += 1
```

## (2) print、return 语句的语义分析

① print 语句：

形如 “print (“王子辉”)", TAC 语句为(‘print’, “王子辉”, ‘-’, ‘-’)。

② return 语句：

形如 “return 0”，TAC 语句为(‘return’,0, ‘-’, ‘-’)。

需要注意的是，如果 print 或 return 后接的是运算式，则相应的 TAC 语句中第二项应为前述运算式的 result。如果 print 或 return 后没有内容，则第二项为 ‘-’。

具体实现如下：

```
def gen_PRINT_STMT_TAC(self, token_list):
    result = []
    if len(token_list[2:]) == 1:
        result.append(TAC(('print', token_list[2]['val'], '-', '-')))
    else:
        tac_list, res = self.gen_OP_TAC(token_list[2:-2])
        for tac in tac_list:
            result.append(tac)
        result.append(TAC(('print', res, '-', '-')))
    return result
```



```

def gen_RETURN_STMT_TAC(self, token_list):
    result = []
    # 返回一个表达式的值
    if len(token_list[1:]) > 1:
        tac_list, res = self.gen_OP_TAC(token_list[1:])
        for tac in tac_list:
            result.append(tac)
        result.append(TAC(('return', res, '-', '-')))
    # 返回仅仅一个值
    elif len(token_list[1:]) == 1:
        result.append(TAC(('return', token_list[1]['val'], '-', '-')))
    # 返回空值
    else:
        result.append(TAC(('return', '-', '-', '-')))
    return result

```

### (3) 控制语句的语义分析

此次语义分析完成了对 if 语句、while 语句和 for 语句的语义分析，其内容大致相同，都是利用了代码回填技术，在遇到控制语句时先占 TAC 语句的坑，在执行到跳转目标的语句段末尾时，将当前 TAC 语句的标号回填至前述控制语句的 TAC 语句当中。该算法看起来比较容易，实际实现起来非常困难，因为在 if 语句中还要考虑 else if、else 语句段，以及各个控制语句嵌套等。实现过程非常复杂，且作为选做任务，在此不过多赘述，仅陈述思路：定义全局状态栈，在读取到 if、else if、else、while、for 时，加入 TAC 语句占坑（占坑的具体做法是记录下当前需要回填标签的 TAC 语句标号，将其置入相应栈的栈顶中，此时并不知道将要跳转至何处）并将其（if、else if、else、while、for）置入状态栈栈顶作为后续代码段的状态；在读取到右大括号“}”时，根据状态栈栈顶元素执行相应跳转标签回填操作，随后状态栈栈顶元素弹出。嵌套的情况也很好处理，即定义“栈中栈”，将需要回填跳转标签的 TAC 语句根据嵌套情况执行回填标签栈的嵌套。具体实现如下：

```

if line[0]['val'] in branch_list:
    if line[0]['val'] == 'if':
        branch_state_stack.append('if')
        gl.global_if_goto_label_stack.append(gl.global_label_cnt - 1)
    elif line[0]['val'] == 'else':
        if line[1]['val'] == 'if':
            branch_state_stack.append('elif')
            gl.global_if_goto_label_stack.append(gl.global_label_cnt - 1)
        else:
            branch_state_stack.append('else')
    elif line[0]['val'] == 'for':
        branch_state_stack.append('for')

    elif line[0]['val'] == 'while':
        branch_state_stack.append('while')

```

```

elif line[0]['val'] == '}':
    if branch_state_stack[-1] == 'if':
        branch_state_stack.pop(-1)
        tac_list[gl.global_if_goto_label_stack[-1]].tac_tuple[3] = gl.global_label_cnt + 1
        gl.global_if_goto_label_stack.pop(-1)
        gl.global_goto_label_stack.append([gl.global_label_cnt])
        tac_list.append(TAC(('goto', '-', '-', gl.global_label_cnt + 1))) # 若没有else语句则默认跳转下一条语句

    elif branch_state_stack[-1] == 'elif':
        branch_state_stack.pop(-1)
        tac_list[gl.global_if_goto_label_stack[-1]].tac_tuple[3] = gl.global_label_cnt + 1
        gl.global_if_goto_label_stack.pop(-1)
        gl.global_goto_label_stack[-1].append(gl.global_label_cnt)
        tac_list.append(TAC(('goto', '-', '-', gl.global_label_cnt + 1))) # 若没有else语句则默认跳转下一条语句

```

```

elif branch_state_stack[-1] == 'else':
    branch_state_stack.pop(-1)
    for label in gl.global_goto_label_stack[-1]:
        tac_list[label].tac_tuple[3] = gl.global_label_cnt
    gl.global_goto_label_stack.pop(-1)

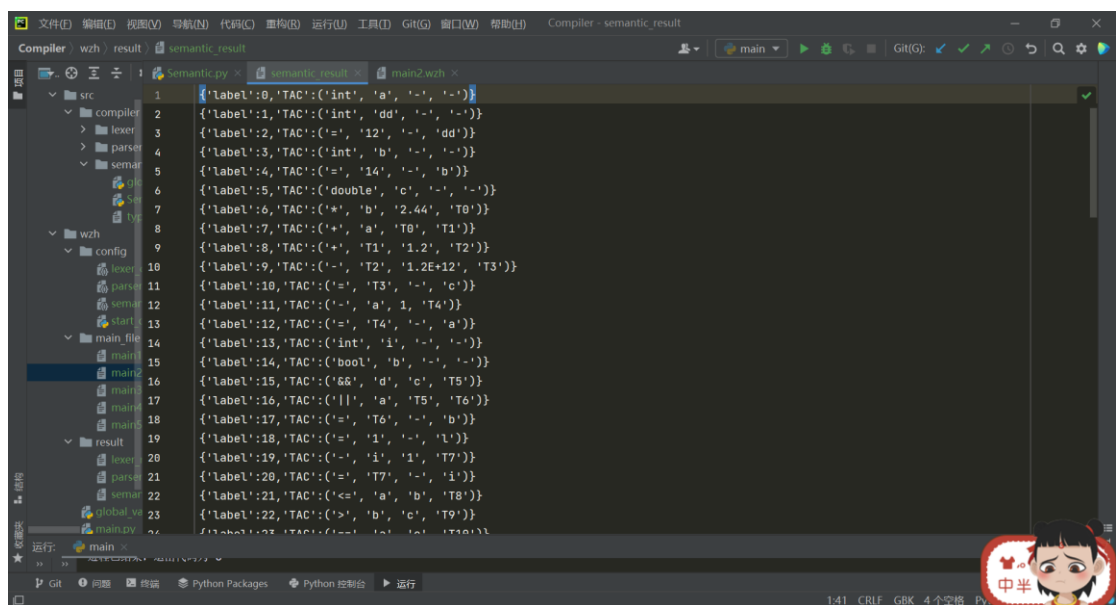
elif branch_state_stack[-1] == 'while':
    branch_state_stack.pop(-1)
    tac_list[gl.global_while_label_stack[-1] + 1].tac_tuple[3] = gl.global_label_cnt + 1
    tac_list.append(TAC(('goto', '-', '-', gl.global_while_label_stack[-1])))
    gl.global_while_label_stack.pop(-1)

elif branch_state_stack[-1] == 'for':
    branch_state_stack.pop(-1)
    # tac_list += gl.global_for_operator_stack[-1]
    tac_list += gen_tac.gen_BLOCK_STMT_EXPRESSION_TAC(gl.global_for_operator_stack[-1])
    gl.global_for_operator_stack.pop(-1)
    tac_list[gl.global_for_label_stack[-1] + 1].tac_tuple[3] = gl.global_label_cnt + 1
    tac_list.append(TAC(('goto', '-', '-', gl.global_for_label_stack[-1])))
    gl.global_for_label_stack.pop(-1)

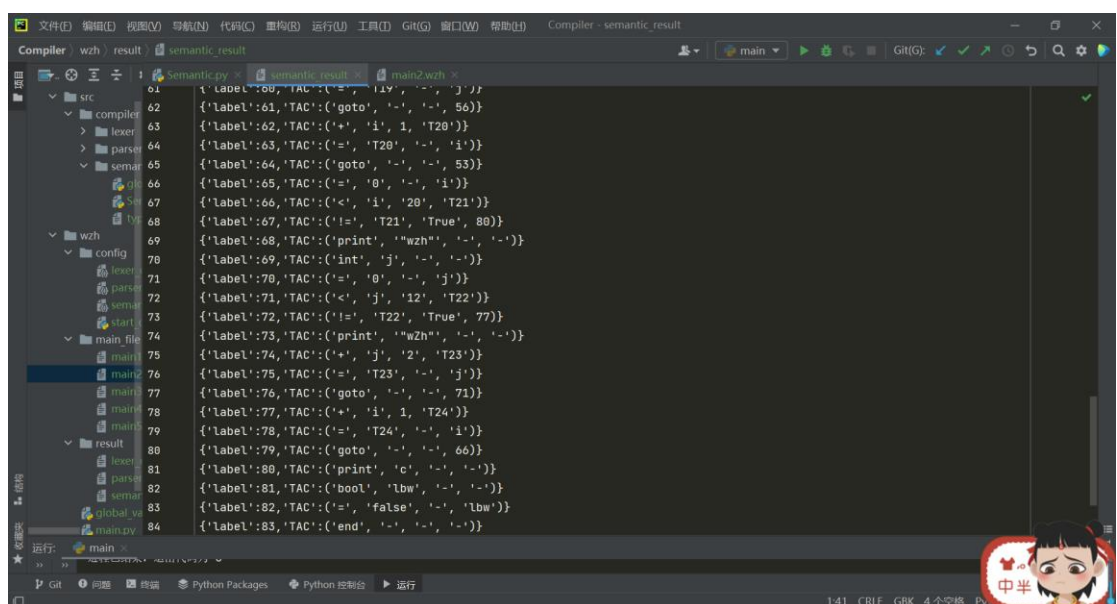
```

### 3、语义分析结果

语义分析部分结果如下图：



```
1 {'label':0,'TAC':('int', 'a', '-', '-')}
2 {'label':1,'TAC':('int', 'dd', '-', '-')}
3 {'label':2,'TAC':('=', '12', '-', 'dd')}
4 {'label':3,'TAC':('int', 'b', '-', '-')}
5 {'label':4,'TAC':('=', '14', '-', 'b')}
6 {'label':5,'TAC':('double', 'c', '-', '-')}
7 {'label':6,'TAC':('*', 'b', '2.44', 'T0')}
8 {'label':7,'TAC':('+', 'a', 'T0', 'T1')}
9 {'label':8,'TAC':('T1', '1.2', 'T2')}
10 {'label':9,'TAC':('T2', '1.2E+12', 'T3')}
11 {'label':10,'TAC':('c', 'T3', '-', 'c')}
12 {'label':11,'TAC':('=', 'a', '1', 'T4')}
13 {'label':12,'TAC':('=', 'T4', '-', 'a')}
14 {'label':13,'TAC':('int', 'i', '-', '-')}
15 {'label':14,'TAC':('bool', 'b', '-', '-')}
16 {'label':15,'TAC':('&&', 'd', 'c', 'T5')}
17 {'label':16,'TAC':('||', 'a', 'T5', 'T6')}
18 {'label':17,'TAC':('=', 'T6', '-', 'b')}
19 {'label':18,'TAC':('=', '1', '-', '1')}
20 {'label':19,'TAC':('=', 'i', '1', 'T7')}
21 {'label':20,'TAC':('=', 'T7', '-', 'i')}
22 {'label':21,'TAC':('<=', 'a', 'b', 'T8')}
23 {'label':22,'TAC':('>', 'b', 'c', 'T9')}
```



```
60 {'label':60,'TAC':('int', 'j', '-', '-')}
61 {'label':61,'TAC':('goto', '-', '-', '56')}
62 {'label':62,'TAC':('+', 'i', '1', 'T20')}
63 {'label':63,'TAC':('=', 'T20', '-', 'i')}
64 {'label':64,'TAC':('goto', '-', '-', '53')}
65 {'label':65,'TAC':('=', '0', '-', 'i')}
66 {'label':66,'TAC':('<', 'i', '20', 'T21')}
67 {'label':67,'TAC':('!=', 'T21', 'True', 80)}
68 {'label':68,'TAC':('print', '"wzh"', '-', '-')}
69 {'label':69,'TAC':('int', 'j', '-', '-')}
70 {'label':70,'TAC':('=', '0', '-', 'j')}
71 {'label':71,'TAC':('<', 'j', '12', 'T22')}
72 {'label':72,'TAC':('!=', 'T22', 'True', 77)}
73 {'label':73,'TAC':('print', '"wzh"', '-', '-')}
74 {'label':74,'TAC':('+', 'j', '2', 'T23')}
75 {'label':75,'TAC':('=', 'T23', '-', 'j')}
76 {'label':76,'TAC':('goto', '-', '-', '71')}
77 {'label':77,'TAC':('+', 'i', '1', 'T24')}
78 {'label':78,'TAC':('=', 'T24', '-', 'i')}
79 {'label':79,'TAC':('goto', '-', '-', '66')}
80 {'label':80,'TAC':('print', 'c', '-', '-')}
81 {'label':81,'TAC':('bool', 'lbw', '-', '-')}
82 {'label':82,'TAC':('=', 'false', '-', 'lbw')}
83 {'label':83,'TAC':('end', '-', '-', '-')}
```

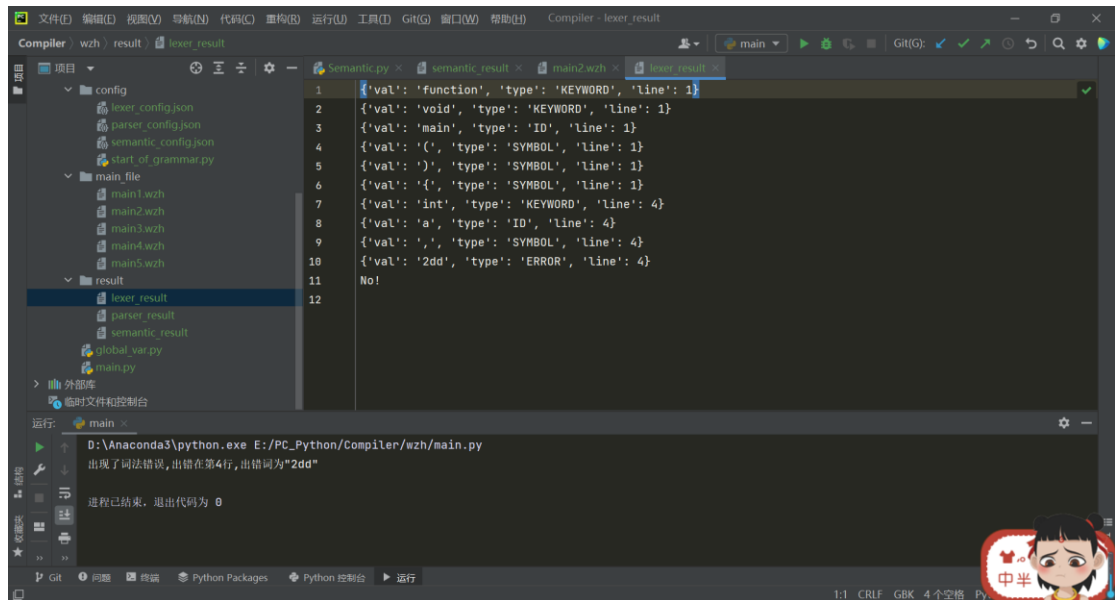
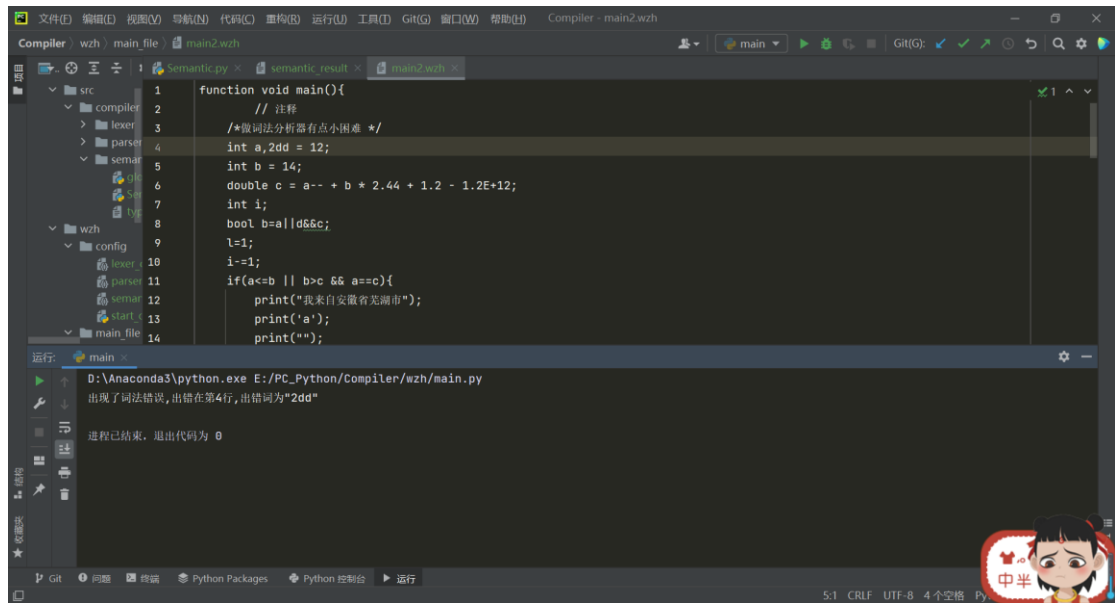
## 五、出错处理

对于输入代码段，我们首先进行词法分析，一旦词法分析不通过，将不再进行后续分析；通过词法分析后，我们进行语法分析，一旦语法分析不通过，将不再进行后续分析；只有同时通过词法分析和语法分析的代码段，编译器才会对其进行语义分析。

关于出错处理，由于语义分析仅局限于生成输入代码段的 TAC 语句序列，因此在本次课程设计中，只有词法和语法的出错处理，均实现输出错误代码行号以及错误原因。

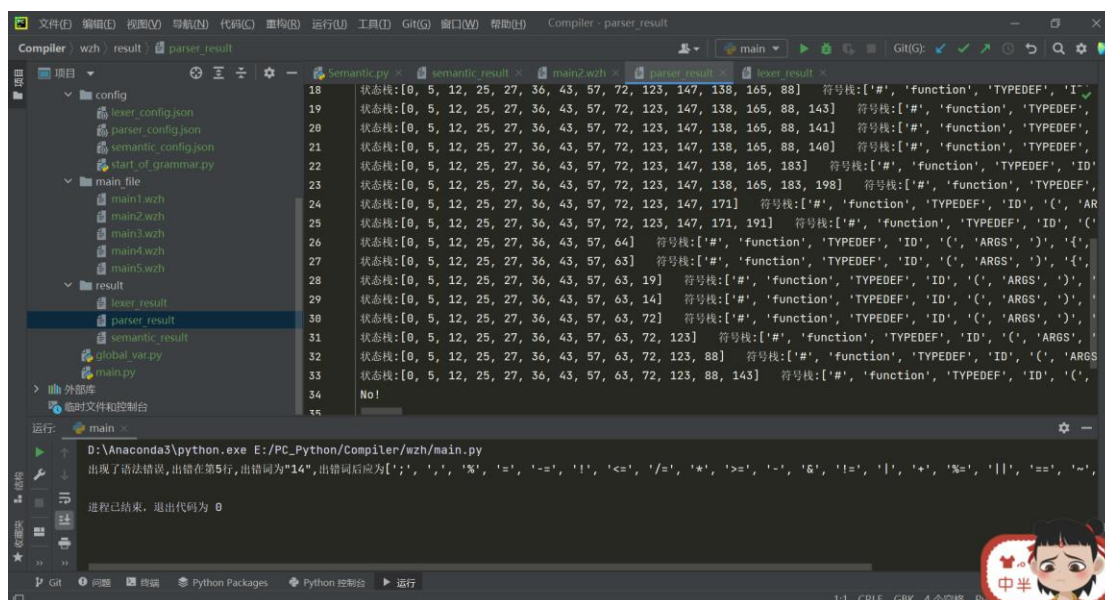
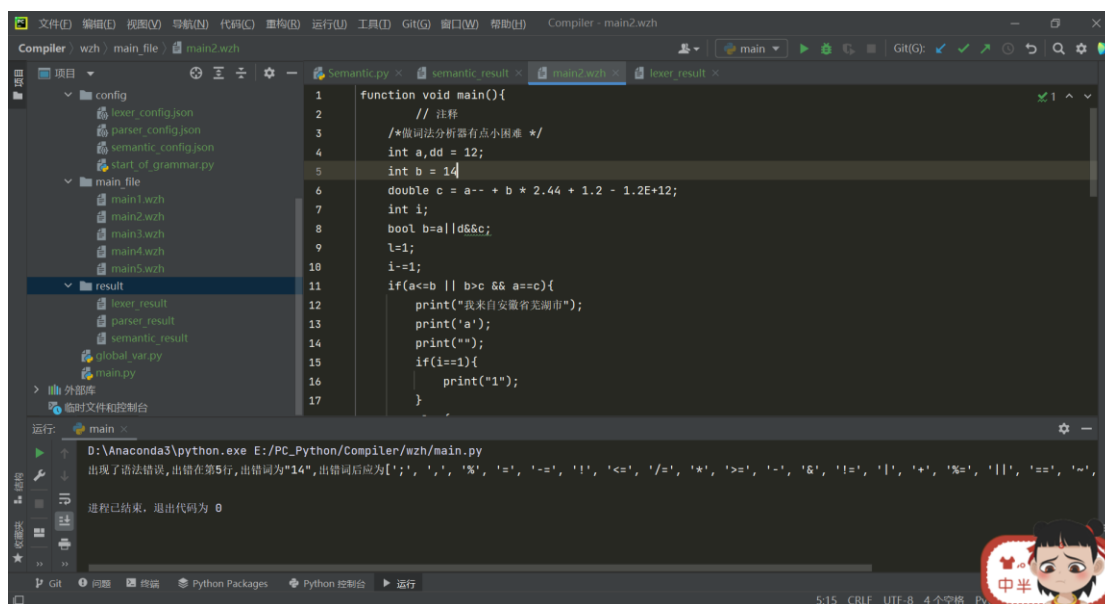
### 1、词法分析出错

下图为错误命名的变量导致词法分析出错：



## 2、语法分析出错

下图为缺少分号导致语法分析出错：



## 六、心得体会

这次的课程设计，一个字总结：**难**！两个字总结：**痛快**！

除去某一门老师划考试重点的专选课，《编译原理》大概是我大学三年里学的最好的一门科目了，真的没有之一。考试的时候轻敌了，以为轻车熟路的题目写得飞快，结果送分题干错了，甚至没拿满绩，心里很不爽，于是想要在此次课程设计里扳回一城。

在同学们看来,我写这次课设的必做部分,确确实实只花了五天时间去完成,但是实际上远不止于此。我在寒假便拉长战线,找到学长曾经做过的课程设计,大概了解这次要做什么东西、生成些什么,并做适当的知识储备。有了思路,我就打开熟悉的 **Pycharm** 开始编写程序。老师说,这种任务,一天闲闲地写个两小时,写个五天就能写完。但实际上,我这五天里每天都在电脑旁坐了六七个小时,白天睡个懒觉起来,下午边听课边写程序就能写到晚上十点十一分,有时候

不想终止思路还要加班加点，甚至拿个本子记着自己的思路和进展。夜里睡不着的时候也在想：到底是哪步出了问题？明天要不输出看看是什么样的？但总是忍不住披上外套打开电脑，看看到底哪里出了问题。好在前述必做任务实在没什么太难的地方，五天做完，我就打算不做了。

大概在课设必做任务完成后的一个月左右，一位成绩非常优秀的同学来和我交流代码问题，那阵子，一股莫名的力量涌上心头，想着要把语义分析给办了。说干就干，我打开熟悉的 Pycharm，构建语义分析的大局。语义分析的思维量比词法和语法要大得多，我选择用我自己的方式生成 TAC 语句序列。前述的控制语句 TAC 语句处理，寥寥数行字，却倾注了我四五天的心血。在白天苦思冥想也想不出的嵌套、回填逻辑，竟然是在睡梦中解决的。我闭上眼睛，满脑子的入栈出栈、栈入栈栈出栈，一下子就有了灵感。第二天起了个大早，把睡梦中所想转化为五彩缤纷的代码，运行成功，顺便吃了个早饭。

由于自己身处智能专业，以后大概感兴趣的方向也是模式识别这方面，与编译器的缘分可能就告一段落了。但总结一下的话，这次的课程设计很考验个人的工程能力和代码能力，无论是人工智能领域，还是计算机应用领域，这两项能力都不可或缺；既要会分析需要解决的问题，又要会把思路转换为代码，实属不易。不管这次课设的结果如何，我对自己都挺满意的。