parser-stage report

> 姓名：郑友捷　学号：2021010771　班级：计14

工作内容

依据注释对`my_parser.py`进行了补全。

- `p_relational`函数

  该函数处理比较语句。

  1. 将所示文法消去左递归，转化为`EBNF`范式为

     ```
     relational: additive { '<' additive | '>' additive | '<=' additive |
     '>=' additive }
     ```

  2. 依据文法补全函数：

     1. 首先按照`p_additive`文法进行遍历，得到一个树上节点`node`，并更新next_token。
     2. 若下一个token被包含在`relational`的`first`集合`Less/Greater/LessEqual/GreaterEqual`中，说明存在比较语句，则读取该字符并转化为操作符
     3. 若存在操作符，则按照`p_additive`文法读取得到比较语句的另一边的节点`rhs`，并把`node`与`rhs`结合更新为`Binary`节点`node`，完成比较
     4. 返回`node`。

- `p_logical_and`函数

  该函数处理逻辑与语句。

  1. 将所示文法转化为`EBNF`范式为

     ```
     logical_and: equality { '&&' equality }
     ```

  2. 依据文法补全函数

     1. 首先按照`equality`的文法遍历，得到一个树上节点`node`，并更新next_token
     2. 若下一个字符是`logical_and`文法的`first`集合中的元素（即`And`），则说明存在逻辑与语句，读取下一个token并转化为操作符
     3. 若存在操作符，则按照`p_equality`读取，得到逻辑与语句的另一边的节点`rhs`，并把`node`与`rhs`结合更新为`Binary`节点`node`，完成比较
     4. 返回`node`。

- `p_assignment`函数

  函数在已经给出的代码中新建了一个节点，若下一个token为`Assign`，说明该节点不是条件表达式节点，而是`Assign`赋值表达式节点。则操作如下

  1. 通过`lookahead`消耗下一个token`Assign`。
  2. 按照`expression`文法读取，获得`assign`的`expression`部分，并把一开始得到的`node`与`expression`组合成为一个完整的`Assignment`节点
  3. 返回新的`Assignment`节点

- `p_expression`函数

依据文法为`expression:assignment`，在按照`p_expression`文法遍历时只需要转化为`p_assignment`文法即可，故只需要添加

```
return p_assignment(self)
```

- `p_statement`函数

  补全了`p_statement`文法中当下一个字符为`if`或者`return`的情况

  1. 当下一个token为`if`，则用`p_if`文法处理并得到一个节点，将该节点返回
  2. 当下一个token为`return`，则用`p_return`文法处理并得到一个节点，将该节点返回。
- `p_declaration`函数

  函数在已给出的代码中得到一个`Declaration`节点`decl`，当下一个token为`Assign`，则需要为其添加初始化表达式，操作如下

  1. 通过lookahead消耗下一个token`Assign`。
  2. 按照`p_expression`文法处理，作为`decl`的初始化表达式
- `p_block`函数：

  在给出的部分中，需要判断下一个token属于哪种文法的first集合，从而确定下一个token的类型。

  1. 若下一个token属于`statement`文法的`first`集合，则按照`statement`的文法处理下一个token并返回类型为`statement`的节点。
  2. 若下一个token属于`declartaion`文法的`first`集合，则按照`declartion`的文法处理下一个token，同时判断下一个token是否为`Semi`（即分号），若是则返回类型为`declaration`节点，否则报错。
- `p_if`函数

  1. 先判断下一个token是否为if，若是则读取并消耗这个token。
  2. 再判断下一个token是否为LParen（左括号），若是则读取并消耗这个token。
  3. 以`expression`文法读取，得到类型为`expression`的`cond`节点
  4. 再判断下一个token是否为RParen（右括号），若是则读取并消耗这个token。
  5. 以`statement`文法读取，得到类型为`statement`的`then`节点
  6. 若下一个token为`Else`，则消耗该token并用`statement`文法读取，得到类型为`statement`的分支节点`otherwise`。
  7. 将`cond`/`then`/`otherwise`（若有）组合成一个完整的`If`类型节点并返回。
- `p_return`函数

  该函数用于处理`return`语句

  1. 先判断下一个token是否为return，若是则读取并消耗这个token。
  2. 以`expression`文法读取，得到类型为`expression`的`expr`节点代表返回语句的表达式
  3. 再判断下一个token是否为Semi（分号），若是则读取并消耗这个token。
  4. 将`expr`节点转为`Return`类型的节点并返回。
- `p_type`函数

  该函数用于生成`int`类型的节点。

  判断下一个token是否为`int`，若是则返回`int`类型的节点，否则报错。

思考题

1. 文法为

```
additive: multiplicative Q

Q:  '+' multiplicative | '-' multiplicative
```

2. 出错程序的例子：

```
int main() {
    return 100 100;
}
```

出错处理方法：

在上文中，读取完第一个100后期望读取分号作为return语句的结尾，但此时读取到的是一个int类型的数字，从而导致报错。

预想的错误恢复机制为：

若当前的下一个token并不是预期想要读取的token，则使用`lookahead()`消耗掉当前token，转而读取下一个token，直到token符合预期。

对应代码为

```
def p_return(self: Parser) -> Return:
    "return : 'return' expression ';'"
    try:
        lookahead = self.lookahead
        lookahead("Return")
        expr = p_expression(self)
        while(self.next_token.type != 'Semi'):
            # 进行错误处理
            print(self.next_token)
            lookahead()

        lookahead("Semi")
        return Return(expr)
    except:
        raise DecafSyntaxError(self.next_token)
```

该机制可以成功处理上述错误程序。