parser-stage：自顶向下语法分析器

陈炫中　2019011236

实验内容

结合课堂上学习的 LL(1) 分析方法，完成一个手工实现的递归下降语法分析器，支持 Step1-6 的语法，具体需要完成 `frontend/parser/my_parser.py` 中的以下函数：

```
p_relational p_logical_and p_assignment p_expression p_statement
p_declaration p_block p_if p_return p_type
```

在 `frontend/parser/my_parser.py` 中：

通过 `lookahead(type: Optional[str] = None)` 来匹配并消耗当前的 `next_token`。

在具体的实现过程中：

`p_relational` 参照了已有的 `p_equality` 进行实现；

`p_logical_and` 参照了已有的 `p_logical_or` 进行实现；

`p_assignment` 匹配并消耗 `Assign` 后，通过 `p_expression(self)` 得到 `rhs`，建立 `Assignment` 节点并返回；

`p_expression` 直接 `return p_assignment(self)`；

`p_statement` 通过对 `self.next` 进行判断，在 `If/Return` 直接返回其对应的函数，否则报错；

`p_declaration` 首先匹配并消耗 `Assign`，然后通过 `p_expression(self)` 来得到表达式初值，最后将其赋给 `decl.init_expr`；

`p_block` 中的 `p_block_item`，当 `self.next` 分别在 `p_statement/p_declaration` 的 `First` 集合中时，返回对应的函数即可；

`p_if` 中也是逐步匹配并消耗相应的 `token`，通过 `expression/statement` 得到 `condition/body` 后返回 `If` 节点，返回前对 `next_token` 中有 `Else` 的情况要通过 `statement` 获得 `otherwise`；

`p_return` 匹配并消耗 `Return`，通过 `p_expression` 获得要返回的表达式，最后匹配并消耗 `Semi` 后将其返回即可；

`p_type` 匹配并消耗 `Int` 后返回一个 `TInt()` 节点。

思考题

1.

消除直接左递归后得到不含左递归的 LL(1) 文法：

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

2.

一个出错程序的例子：

```
int main() {
    return 0
}
```

该例子中在语法上 `return 0` 后 缺少 `;`，对于这个例子，我们的程序框架对该语句的解析过程为：

```
def p_return(self: Parser) -> Return:
    "return : 'return' expression ';'"

    """ TODO
    1. Match token 'Return'.
    2. Parse expression.
    3. Match token 'Semi'.
    4. Build a `Return` node and return it.
    """
    lookahead = self.lookahead
    lookahead("Return")
    expr = p_expression(self)
    lookahead("Semi")
    return Return(expr)
```

我心目中的错误恢复机制是解析器在解析到 `return 0` 时，先对下一个输入符进行判断，如果是 `;` 则同之前一样 `lookahead("Semi")`；反之则直接 `return Return(expr)`。
