Parser Stage

沙之洲 2020012408

实验内容

这一阶段我们主要实现的是代码的解析部分，也即使 AST 树的构建部分。具体来说，我们要根据给定的产生式解析代码，并构建 AST 树。

我们以 p_relational 为例进行详细分析

```
@first(*p_additive.first)
def p_relational(self: Parser) -> Expression:
    """
    relational : relational '<' additive
        | relational '>' additive
        | relational '<=' additive
        | relational '>=' additive
        | additive
    """

    """ TODO
    1. Refer to the implementation of 'p_equality'.
    """
    lookahead = self.lookahead
    node = p_additive(self)
    while self.next in ("Less", "Greater", "LessEqual", "GreaterEqual"):
        op = BinaryOp.backward_search(lookahead())
        rhs = p_additive(self)
        node = Binary(op, node, rhs)
    return node
```

这里 lookahead 函数是读取并消耗下一个字符，而 self.next 只是读取下一个字符，并不消耗它。

我们先假定 node 是要走 additive 产生式，因此在 while 之前有 `node = p_additive(self)` 的语句。

接下来，我们通过 self.next 判断下一个字符是否属于我们合法的产生式当中，如果是，则按照产生式访问，并把之前定义的 node 覆盖掉。

最后，我们把最终的 node 返回回去。

其他的 parser 函数与上述函数类似，只需要通过代码实现对应的产生式即可。

思考题

第一题

我们先消除左递归，通过引入一个新的非终结符 `additive_temp` 的方式，产生式变为

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

第二题

考虑如下错误代码

```
int main () {
    int a = 1;
    int b = 1;
    a = a { b + a;
    return 0;
}
```

可以注意到在 return 的上一行，a 和 b 之间的运算出现了错误。

而在我们的框架的解析中，实际上是进行到了非终结符 additive 的解析步骤中。

对于这个错误处理，我们只需要考虑 $S = BeginSym \cup EndSym$ 即可。可以推测，S 集合中会包含 ! , ~ , Identifier 和 ; 等符号。对于我们的错误代码而言，当 parser 遇到 { 之后，会跳过不属于 S 集合的 { ，并从后边的 Identifier b 开始继续完成后续分析。