Parser-Stage Report

曹伦郗 *2020011020*

实验内容

实现了*p_relational*，与*p_equality*以及*p_additive*类似。

```
    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
```

其等效的*EBNF*为：

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

实现了*p_logical_and*，与*p_logical_or*类似。

```
    lookahead = self.lookahead
    node = p_equality(self)
    while self.next in ('And',):
        op = BinaryOp.backward_search(lookahead())
        rhs = p_equality(self)
        node = Binary(op, node, rhs)
    return node
```

其等效的*EBNF*为：

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

实现了*p_assignment*，即消耗赋值号并分析右值表达式，将返回值用于构造赋值语句的*AST*节点。

```
    if self.next == "Assign":
        if node_type != "Identifier":
            raise DecafSyntaxError(current_tok)
        """ TODO
        1. Match token 'Assign'.
        2. Parse expression to get rhs.
        3. Build an `Assignment` node with node (as lhs) and rhs
        4. Return the node.
        """
        lookahead()
        rhs = p_expression(self)
        return Assignment(node, rhs)
```

实现了*p_expression*。

```
    """ TODO
    1. Parse assignment and return it.
    """
    return p_assignment(self)
```

实现了*p_statement*。

```
    elif self.next == 'If':
        return p_if(self)
    elif self.next == 'Return':
        return p_return(self)
```

实现了*p_declaration*。

```
    if self.next == "Assign":
        """TODO
        1. Match token 'Assign'.
        2. Parse expression to get the initial value.
        3. Set the child `init_expr` of `decl`.
        """
        lookahead()
        decl.init_expr = p_expression(self)
```

实现了*p_block*，值得注意的是，声明语句被分析后，还需匹配一个分号。

```
        lookahead = self.lookahead
        if self.next in p_statement.first:
            # TODO: Complete the action if the next is a statement.
            return p_statement(self)
        elif self.next in p_declaration.first:
            # TODO: Complete the action if the next is a declaration.
            node = p_declaration(self)
            lookahead('Semi')
            return node
```

实现了*p_if*。

```
    lookahead = self.lookahead
    lookahead('If')
    lookahead('LParen')
    cond = p_expression(self)
    lookahead('RParen')
    then = p_statement(self)
    node = If(cond, then)
    if self.next == 'Else':
        lookahead()
        node.otherwise = p_statement(self)
    return node
```

实现了*p_return*。

```
    lookahead = self.lookahead
    lookahead('Return')
    expr = p_expression(self)
    lookahead('Semi')
    return Return(expr)
```

实现了*p_type*，由于只有*int*类型，故返回一个*Tint*类节点。

```
    self.lookahead('Int')
    return TInt()
```

思考题

1.
   ```
   additive : multiplicative T
   T : '+' multiplicative T | '-' multiplicative T | ϵ
   ```

2. 出错程序如

   ```
   int n = return 0;
   ```

   这是一条带有初值的声明语句。从*p_declaration*调用*p_expression*开始，上层语法分析函数调用下层，直到*p_unary*被调用，发现*First(unary)*不含下一终结符*Return*，故抛出语法错误异常。

   在我设想的错误恢复机制下，每个语法分析函数在其开始正常语法分析流程前，都应当检查下一终结符是否在该函数所分析的非终结符的*First*集合中出现。若不存在，则应打印显示该终结符为语法错误，并用*lookahead*消耗掉它，但不抛出异常，以便程序继续运行。按照这样的方法连续越过出错位置，直到遇到合法的下一终结符，即在对应的*First*集合中能找到该终结符，则开始正常的语法分析流程。

   具体到这个例子来说，在*p_unary*中，将会发现*First(unary)*不含下一终结符*Return*，于是打印显示这个终结符为一个语法错误，同时用*lookahead*将*Return*消耗掉。而下一个终结符*Integer*∈*First(unary)*，故*Integer*不是语法错误，可以开始进行语法分析。