parser-stage：自顶向下语法分析器

计04 何秉翔 2020010944

1. 实验内容

在 parser-stage 中，我们主要是在提供的 `parser` 框架上实现语法分析，不涉及中端和后端。

1.1 p_relational

仿照 `p_equality` 的实现，对于产生式：

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

我们将其转化为等价的 `EBNF` 文法：

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

先解析 `additive`，再根据下一个 `token` 是否是上述关系运算符，往下扫描并建立 `Binary` 节点。

```
lookahead = self.lookahead
node = p_additive(self)
while self.next in ("Less", "LessEqual", "Greater", "GreaterEqual"):
    op = BinaryOp.backward_search(lookahead())
    rhs = p_additive(self)
    node = Binary(op, node, rhs)
return node
```

1.2 p_logical_and

仿照 `p_logical_or` 的实现，对于产生式：

```
logical_and : logical_and '&&' equality
        | equality
```

我们将其转化为等价的 `EBNF` 文法：

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

先解析 `equality`，再根据下一个 `token` 是否是 `And`，来进行随后的解析。

```
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
```

1.3 p_assignment

产生式如下：

```
assignment : Identifier '=' expression
           | conditional
```

解析时，先按照 `conditional` 解析，再查看下一个 `token`，若是 `Assign`，则说明为普通的赋值表达式，匹配 `Assign` 并一步步解析即可。

```
lookahead("Assign")
rhs = p_expression(self)
return Assignment(node, rhs)
```

1.4 p_expression

产生式如下：

```
expression : assignment
```

则直接解析返回即可：

```
return p_assignment(self)
```

1.5 p_statement

产生式如下：

```
statement : if | return | ( expression )? ';'
```

`first` 列表为：

```
@first("If", "Return", "Semi", *p_expression.first)
```

框架中已经给出了 `Semi` 和 `p_expression.first` 的实现，接下来只需分别实现 `If` 和 `Return` 的情况即可。

```
elif self.next == "Return":
    return p_return(self)
elif self.next == "If":
    return p_if(self)
else:
    raise DecafSyntaxError(self.next_token)
```

1.6 p_declaration

产生式如下（框架没有带 `;`，与“规范”好像不太符）：

```
declaration : type Identifier ('=' expression)? ';'
```

框架代码已经给出了除了 `Assign` 部分的实现，因此在 `Assign` 下，需要去解析初值表达式并赋值给 `Declaration` 节点上。最后解析 `;`。

```
if self.next == "Assign":
    lookahead("Assign")
    decl.init_expr = p_expression(self)
lookahead("Semi")
```

1.7 p_block

产生式如下：

```
block : (statement | declaration)*
```

我们需要对 `statement` 和 `declaration` 分别解析。

```
if self.next in p_statement.first:
    return p_statement(self)
elif self.next in p_declaration.first:
    return p_declaration(self)
```

1.8 p_if

产生式如下：

```
if : 'if' '(' expression ')' statement ( 'else' statement )?
```

按照产生式一个个解析，根据条件和 `then` 分支构建 `If` 节点，最后判断是否有 `else` 分支，如果有，则解析并赋值到 `If` 节点的 `otherwise` 属性上。

```
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("Else")
    node.otherwise = p_statement(self)
return node
```

1.9 p_return

产生式如下：

```
return : 'return' expression ';'
```

按照产生式解析并返回 `Return` 节点即可。

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

1.10 p_type

产生式如下：

```
type : 'int'
```

直接 `lookahead` 返回 `TInt` 节点即可。

```
lookahead = self.lookahead
lookahead("Int")
return TInt()
```

2. 思考题

2.1 parser-stage

1. **问题**：在框架里我们使用 EBNF 处理了 `additive` 的产生式。请使用课上学习的消除左递归、消除左公因子的方法，将其转换为不含左递归的 LL(1) 文法。（不考虑后续 multiplicative 的产生式）

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

   解答：

   ```
   additive : multiplicative A
   A : '+' multiplicative A | '-' multiplicative A | \epsilon
   ```

2. **问题**：对于我们的程序框架，在自顶向下语法分析的过程中，如果出现一个语法错误，可以进行**错误恢复**以继续解析，从而继续解析程序中后续的语法单元。请尝试举出一个出错程序的例子，结合我们的程序框架，描述你心目中的错误恢复机制对这个例子，怎样越过出错的位置继续解析。（注意目前框架里是没有错误恢复机制的。）

   解答：

   ```
   int main() {
       int x = 0;
       int y = 1;
       if (x) x = 1 else y = 1;
       return 0;
   }
   ```

   语法错误在于 `if` 的 `then` 分支没有 `;`，在解析到 `then` 分支的 `statement` 时，解析了 `expression` 后又 `lookahead("Semi")`，在这里产生报错。我们希望的是如果在这里发生语法错误，可以通过异常处理，先记录错误的行列以及上下文块信息，回溯到解析 `If` 的地方继续往下解析。

3. **问题**：（选做，不计分）指出你认为的本阶段的实验框架/实验设计的可取之处、不足之处、或可改进的地方。

   解答：

   - 可取之处：实验文档的组织很好，比较清晰，分为“任务描述”、“框架介绍”和“规范”等，一目了然，方便查阅。
   - 可改进的地方：虽然需要补全的代码，以及可以参考的已有实现都比较清楚，但是具体在写的时候可能写完了也不知道整个框架在干啥，如果能对整体框架有个更清晰的介绍就好了。