

Stage-2 实验报告

范道宇 2019013273

一、工作概述。

- p_Type 函数：

匹配 `int`，生成 `IntType` 节点作为返回值。

```
static ast::Type* p_Type(){
    /* type : Int */
    Location* location=lookahead(TokenType::INT).loc;
    return new ast::IntType(location);
}
```

- p_StmtList 函数：

检测下一个 token 是不是 statement 或者 declaration 的第一个 token，如果是的话就递归地解析 statement 或者 declaration。

```
static ast::StmtList* p_StmtList(){
    // StmtList : (Statement | VarDecl ';')*
    ast::StmtList* statements = new ast::StmtList();
    if (!isFirst[SymbolType::Statement][next_token.type] &&
        !isFirst[SymbolType::VarDecl][next_token.type] &&
        !isFollow[SymbolType::StmtList][next_token.type]) {
        mind::err::issue(next_token.loc, new mind::err::SyntaxError("unexpected " +
            TokenName[next_token.type]));
        lookahead();
    }
    while(!isFollow[SymbolType::StmtList][next_token.type]){
        if(isFirst[SymbolType::Statement][next_token.type]){
            // TODO: Complete the action if the next is a statement.
            // Using statements->append(...)
            statements->append(p_Statement());
        }else if(isFirst[SymbolType::VarDecl][next_token.type]){
            // TODO: Complete the action if the next is a declaration. Remeber to consume
            ';'.
            statements->append(p_VarDecl());
            lookahead(TokenType::SEMICOLON);
        }else{
            mind::err::issue(next_token.loc, new mind::err::SyntaxError("expect statement
            or vardecl, get" + TokenName[next_token.type]));
        }
    }
    return statements;
}
```

- p_Statement 函数：

通过判断 `next_token.type` 来区分下一条该 statement 是 if/block/empty，分别递归的解析相应 statement，另外，如果 `next_token.type` 不是上述的情况，说明发生了语法错误，应该报错。

```
static ast::Statement* p_Statement(){
    /* statement : if | return | ( expression )? ';' | '{' StmtList '}' */
    if(isFirst[SymbolType::Expression][next_token.type]){
        ast::Expr* stmt_as_expr = p_Expression();
        lookahead(TokenType::SEMICOLON);
    }
```

```

        return new ast::ExprStmt(stmt_as_expr, stmt_as_expr->getLocation());
    }else{
        if(next_token.type == TokenType::RETURN){
            return p_Return();
        }
        // TODO: Complete the action to do if the next token is
        `If`/`LBRACE`/`SEMICOLON`
        else if(next_token.type == TokenType::IF){
            return p_If();
        }
        else if(next_token.type == TokenType::LBRACE){
            return p_Block();
        }
        else if(next_token.type == TokenType::SEMICOLON){
            Location* location = lookahead(TokenType::SEMICOLON).loc;
            return new ast::EmptyStmt(location);
        }
        else{
            // next token is not allowed
            mind::err::issue(next_token.loc, new mind::err::SyntaxError("expect
            if/LBRACE/SEMICOLON, get" + TokenName[next_token.type]));
            return nullptr;
        }
    }
    mind::err::issue(next_token.loc, new mind::err::SyntaxError("expect statement,
    get" + TokenName[next_token.type]));
    return NULL;
}

```

- p_VarDecl 函数:

识别出 Identifier 作为 VarDecl 的 name，如果有 initExpression，就继续解析出 expression 作为 VarDecl 的 init，再将生成的 VarDecl 返回。

```

static ast::VarDecl* p_VarDecl(){
    /* declaration : type Identifier ('=' expression)? */
    ast::Type* type = p_Type();
    Token identifier = lookahead(TokenType::IDENTIFIER);
    if(next_token.type == TokenType::ASSIGN){
        lookahead(TokenType::ASSIGN);
        ast::Expr* expression = p_Expression();
        return new ast::VarDecl(identifier.id, type, expression, type->getLocation());
    }
    return new ast::VarDecl(identifier.id, type, type->getLocation());
}

```

- p_Return 函数:

匹配 return 后递归解析 expression，用返回值构造 ReturnStmt。

```

static ast::ReturnStmt* p_Return(){
    /* return : 'return' expression ';' */
    lookahead(TokenType::RETURN);
    ast::Expr* expression = p_Expression();
    lookahead(TokenType::SEMICOLON);
    return new ast::ReturnStmt(expression, expression->getLocation());
}

```

- p_If 函数:

匹配一系列终结符，递归解析 expression，trueStatement，falseStatement，用返回值构造 IfStmt。

```

static ast::IfStmt* p_If(){
    /* if : 'if' '(' expression ')' statement ( 'else' statement )? */

    // TODO:
    /**
     * 1. Match token 'If' and 'LParen'.
     * 2. Parse expression to get condition.
     * 3. Match token 'RParen'.
     * 4. Parse statement to get body.
     * 5. Build a 'If' node with condition and body.
     * 6. If the next token is 'Else', match token 'else' and parse statement to get
    otherwise of the node.
     * 7. Return the 'If' node.
     *
     * hint: use an `EmptyStmt` for an empty else branch instead of NULL
     * hint: to check the type of next_token without consuming it, access
    next_token.type directly
    */
    Location* location = lookahead(TokenType::IF).loc;
    lookahead(TokenType::LPAREN);
    ast::Expr* condition = p_Expression();
    lookahead(TokenType::RPAREN);
    ast::Statement* trueStatement = p_Statement();
    if(next_token.type==TokenType::ELSE){
        lookahead(TokenType::ELSE);
        ast::Statement* falseStatement = p_Statement();
        return new ast::IfStmt(condition, trueStatement, falseStatement, location);
    }
    else{
        ast::EmptyStmt* emptyStatement = new ast::EmptyStmt(next_token.loc);
        return new ast::IfStmt(condition, trueStatement, emptyStatement, location);
    }
}

```

- p_Expression 函数：

无需额外操作，只需递归解析 assignment。

```

static ast::Expr* p_Expression(){
    /* expression : assignment */
    return p_Assignment();
}

```

- p_Assignment 函数：

递归解析 conditional，其返回值作为 Identifier（注意这里应该做强制类型转换）。递归解析 expression，其返回值作为要赋的值，用这两个返回值构造 AssignExpr。

```

static ast::Expr* p_Assignment(){
    /*
        assignment : Identifier '=' expression
                    | conditional                */
    ast::Expr* node = p_Conditional();
    if(next_token.type == TokenType::ASSIGN){
        lookahead(TokenType::ASSIGN);
        ast::Expr* expression = p_Expression();
        return new ast::AssignExpr(dynamic_cast<ast::LvalueExpr*>(node)->lvalue,
        expression, node->getLocation());
    }
    return node;
}

```

- p_LogicalAnd 函数：

仿照 p_LogicalOr 函数，根据扩展巴克斯范式递归解析 equality 。

```
static ast::Expr* p_LogicalAnd(){
    /* logical_and : logical_and '&&' equality
                       | equality                               */

    // equivalent EBNF:

    /* logical_and : equality { '&&' equality } */
    ast::Expr* node = p_Equality();
    while(next_token.type == TokenType::AND){
        Token And = lookahead();
        ast::Expr* operand2 = p_Equality();
        node = new ast::AndExpr(node,operand2,And.loc);
    }

    return node;
}
```

- p_Relational 函数：

仿照 p_Additive ，根据给出的扩展巴克斯范式进行匹配，递归解析 additive 。

```
static ast::Expr* p_Relational(){
    /* relational : relational '<' additive
                       | relational '>' additive
                       | relational '<=' additive
                       | relational '>=' additive
                       | additive                               */

    // equivalent EBNF:
    /* relational : additive { '<' additive | '>' additive | '<=' additive | '>='
additive} */
    ast::Expr* node = p_Additive();
    while(next_token.type == TokenType::LT || next_token.type == TokenType::GT ||
next_token.type == TokenType::LEQ || next_token.type == TokenType::GEQ){

        Token operation = lookahead();
        ast::Expr* operand2 = p_Additive();

        switch(operation.type){
            case TokenType::LT:
                node = new ast::LesExpr(node,operand2,operation.loc);
                break;
            case TokenType::GT:
                node = new ast::GrtExpr(node,operand2,operation.loc);
                break;
            case TokenType::LEQ:
                node = new ast::LeqExpr(node,operand2,operation.loc);
                break;
            case TokenType::GEQ:
                node = new ast::GeqExpr(node,operand2,operation.loc);
                break;
            default:
                break;
        }
    }
    return node;
}
```

二、思考题。

1.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
```

答：

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

其中 `epsilon` 表示空串。

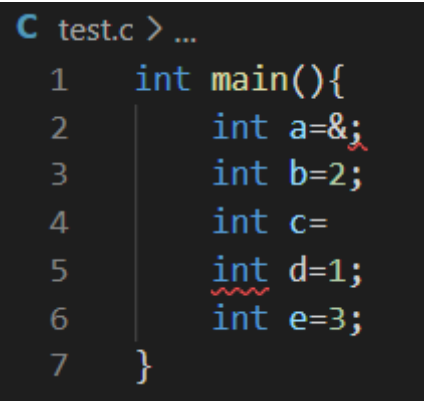
2.对于我们的程序框架，在自顶向下语法分析的过程中，如果出现一个语法错误，可以进行错误恢复以继续解析，从而继续解析程序中后续的语法单元。请尝试举出一个出错程序的例子，结合我们的程序框架，描述你心目中的错误恢复机制对这个例子，怎样越过出错的位置继续解析。（注意目前框架里是没有错误恢复机制的。）

答：

如果在解析一个 `Statement` (或 `Declaration`) 时遇到错误，即下一个 `Token` 的类型不是预期的类型，可以认为发生语法错误，转而将该 `Statement` 的结尾的 `Token` 类型作为目标匹配类型，当匹配成功时，认为该 `Statement` 匹配结束，可以去正常匹配下一条 `Statement`。如果没有匹配成功，就消耗掉一个 `Token` 继续进行匹配，直到匹配成功为止。

例如对于 `int a=&;` 这样的错误，在匹配到 `=` 时发现 `NextToken.type` 不是预期的 `ExpressionType`，于是认为该处出现语法错误，转而去匹配该条 `Statement` 的末尾的 `Token`，即 `;`，发现仍然匹配不成功（因为 `=` 后面是 `&`），于是消耗掉一个 `Token`，即 `&`，继续匹配，发现下一个 `Token` 为 `;`，匹配成功，于是认为该条 `Statement` 匹配结束，可以去正常匹配下一条 `Statement`。

最终实现大致如下的效果：



```
C test.c > ...
1  int main(){
2  int a=&;
3  int b=2;
4  int c=
5  int d=1;
6  int e=3;
7  }
```

可以看到第二行因为末尾有 `;`，所以不影响第三行的正确识别。而第四行没有 `;`，导致第五行的 `d` 是灰色的，没有被识别。