实验一：PR001 CACT

组别：12组

姓名：吴亚晨 祝田田 全荟霖

日期：2025年4月18日

一、实验目标

1. **理解EBNF规范**
   - 掌握扩展巴科斯范式（EBNF）的基本语法和规范。
2. **文法描述能力**
   - 能正确使用EBNF描述计算机编程语言的文法规则。
3. **词法与语法区分**
   - 明确区分词法规则和语法规则，并在实际应用中正确分类。
4. **工具链实践**
   - 学习ANTLR等编译器工具，实现词法和语法分析部分的自动化生成。

二、实验内容

1. ANTLR工具链实践

- 学习ANTLR生成词法分析器（Lexer）和语法分析器（Parser）的流程。
- 通过生成代码（如`*.tokens`和`*.interp`文件）验证工具链的正确性。

2. 环境搭建与Demo验证

- 配置ANTLR开发环境（JDK、CLASSPATH等）。
- 运行课程提供的Demo，确保基础功能正常。

3. CACT文法实现

- 根据CACT语言规范编写`.g4`文法文件。
- 修改`*main.cpp`文件覆盖ANTLR默认错误处理机制，实现自定义词法/语法错误检查。

三、实验过程

1. 词法词法分析器

- **任务**：头部文件设置

- **实现：** 定义语法名称和语言设置

```
grammar Hello;
  options {
      language = Cpp;
  }
  @header {
      #include <vector>
  }
```

- **任务：**Parser 规则
- **实现：** 定义了语言的语法结构,如：程序结构、变量声明等。（由于数量略多，仅展示几个。）

```
//程序的入口，由compUnit组成
program
: compUnit
;

compUnit
    : (funcDef | decl)+
    ;

decl
    : constDecl
    | varDecl
    ;

constDecl
    : CONST bType constDef (COMMA constDef)* SEMI
    ;
```

- **任务：**Lexer 规则
- **实现：** 这部分定义了如何识别单词（Token），例如：关键字、标识符、常量、运算符、分隔符。

```
Ident: [a-zA-Z_] [a-zA-Z0-9_]*;

NewLine: ('\r' '\n'? | '\n') -> skip;

WhiteSpace: [ \t]+ -> skip;

LineComment: '//' ~[\r\n]* -> skip;
BlockComment: '/*' .*? '*/' -> skip;

IntConst: DecimalConst | OctalConst | HexadecConst;
```

```
fragment DecimalConst: NonZeroDigit Digit* | '0';
fragment OctalConst: '0' [0-7]+;
fragment HexadecConst: HexadecimalPrefix HexadecimalDigit+;
```

2. 自定义词法词法错误检查

（1）错误监听器（CustomErrorListener）

- **作用**：捕获词法/语法分析阶段的错误。提供错误统计和消息获取接口并用syntaxError记录错误位置和描述。
- **实现**：

```
class CustomErrorListener : public BaseErrorListener {
    int errorCount;
    std::vector<std::string> errorMessages;
public:
    void syntaxError(...) override; // 收集语法错误
    int getErrorCount() const;
    const std::vector<std::string>& getErrorMessages() const;
};
```

（2）语义分析器（SemanticVisitor）

- **作用**：实现AST遍历和语义规则检查。
- **2.1 符号表管理**：存储变量名、类型和初始化状态，记录函数名检查语义错误。

```
std::unordered_map<std::string, std::pair<std::string, std::string>>
symbolTable;
std::unordered_set<std::string> functionTable;
```

- **2.2 核心检查逻辑-类型校验**：显式初始化时CACT限制初值表达式必须是常数，隐式初始化时未被显式初始化的整型和浮点型变量/常量/数组被默认初始化为0，布尔型被默认初始化为false。

```
// 严格字面量检查（禁止任何运算符）
bool isLiteralInitializer(HelloParser::ConstInitValContext* ctx, const
std::string& expectedType) {
    if (!ctx) return false;

     // 检查是否是数组初始化
```

```
    if (ctx->LBRACE()) {
        // 数组初始化必须使用大括号
        for (auto initVal : ctx->constInitVal()) {
            if (!isLiteralInitializer(initVal, expectedType)) {
                return false;
            }
        }
        return true;
    }

    //直接是字面量
    if (ctx->constExp()) {
        auto addExp = ctx->constExp()->addExp();
        // 禁止加减运算符（AddExp的子节点只能有一个MulExp）
        if (addExp->children.size() > 1) return false;

        auto mulExp = addExp->mulExp();
        // 禁止乘除运算符（MulExp的子节点只能有一个UnaryExp）
        if (mulExp->children.size() > 1) return false;

        auto unaryExp = mulExp->unaryExp();
        // 禁止单目运算符（必须是PrimaryExp）
        if (unaryExp->primaryExp() == nullptr) return false;

        auto primaryExp = unaryExp->primaryExp();
        // 必须是数字字面量
        if (primaryExp->number()) {
            auto numberCtx = primaryExp->number();
            // 类型匹配检查
            if (numberCtx->FloatConst() && expectedType != "float") return
false;
            if (numberCtx->CharConst() && expectedType != "char") return
false;
            if (numberCtx->IntConst() && expectedType != "int") return
false;
            return true;
        }
        return false;
    }
    return false;
}
```

- **2.3 核心检查逻辑-数组检查**：数组格式{}

```
bool checkArrayInitializer(HelloParser::ConstInitValContext* ctx,
                     int expectedDimensions,
                     int* totalElements,
                     bool isTopLevel = true) {
    // 非数组变量禁止用 {}
    if (expectedDimensions == 0 && ctx->LBRACE()) {
        return false;
    }
```

```
    // 数组初始化必须用 {}（顶层）
    if (isTopLevel && expectedDimensions > 0 && !ctx->LBRACE()) {
        return false;
    }

    if (ctx->LBRACE()) {
        // 检查初始化项
        for (auto initVal : ctx->constInitVal()) {
            if (initVal->LBRACE()) {
                // 嵌套 {}：仅当剩余维度 > 1 时允许
                if (expectedDimensions <= 1) {
                    return false; // 单维数组不允许嵌套 {}
                }
                if (!checkArrayInitializer(initVal, expectedDimensions - 1,
                                           totalElements, false)) {
                    return false;
                }
            } else {
                // 扁平化元素：直接计数
                (*totalElements)++;
            }
        }
        return true;
    } else if (ctx->constExp()) {
        // 直接元素（无 {}）
        (*totalElements)++;
        return true;
    }
    return false;
}
```

- **2.4 核心检查逻辑-函数定义检查**：检查main是否重复定义，检查返回类型是否为int，检查参数列表是否为空。

```
void checkMainFunction(HelloParser::FuncDefContext* ctx) {
    if (ctx->Ident()->getText() == "main") {
        if (hasMainFunction) {
            semanticErrors.push_back("Error: 重复的main函数定义");
        }
        hasMainFunction = true;

        if (ctx->funcType()->INT() == nullptr) {
            semanticErrors.push_back("Error: main函数必须返回int类型");
        }

        if (ctx->funcFParams() != nullptr) {
            semanticErrors.push_back("Error: main函数不能有参数");
        }
    }
```

```
}
```

- **2.5 核心检查逻辑-函数定义检查**：检查函数是否已定义，排除内置函数

```
// 函数调用检查
std::any visitUnaryExp(HelloParser::UnaryExpContext* ctx) override {
    // 检查函数调用（如：func2(a,b)）
    if (ctx->Ident() && ctx->LPAREN()) {
        std::string funcName = ctx->Ident()->getText();

        // 排除内置函数
        if (builtinFunctions.find(funcName) == builtinFunctions.end()) {
            // 检查函数是否已定义
            if (functionTable.find(funcName) == functionTable.end()) {
                semanticErrors.push_back("Error: 函数'" + funcName + "'未声
明");
            }
        }
    }
    return visitChildren(ctx);
}
```

- **2.5 核心检查逻辑-变量声明检查**

```
std::any visitVarDecl(HelloParser::VarDeclContext* ctx) override {
    if (!isValidType(ctx->bType())) {
        semanticErrors.push_back("Error: 无效的变量类型");
    }

    std::string type = getTypeText(ctx->bType());

    for (auto varDef : ctx->varDef()) {
        std::string varName = varDef->Ident()->getText();
        if (symbolTable.count(varName)) {
            semanticErrors.push_back("Error: 变量'" + varName + "'重复声明");
        }

        for (size_t i = 0; i < varDef->LBRACK().size(); i++) {
            if (!varDef->IntConst(i)) {
                semanticErrors.push_back("Error: 数组'" + varName + "'的维度必
须为常量");
            }
        }

        if (varDef->constInitVal()) {
```

```
            if (!isLiteralInitializer(varDef->constInitVal(), type)) {
                semanticErrors.push_back("Error: 变量'" + varName + "'必须使用
字面量初始化");
            }
            symbolTable[varName] = {type, "显式初始化"};
        } else {
            symbolTable[varName] = {type, "隐式初始化"};
        }
    }
    return visitChildren(ctx);
}
```

（3）主函数流程

- **3.1 文件处理** 将输入文件转为ANTLR流。 词法分析生成Token流。 语法分析构建ParseTree。

```
ANTLRInputStream input(stream);
HelloLexer lexer(&input);
CommonTokenStream tokens(&lexer);
HelloParser parser(&tokens);
```

- **3.2 错误处理集成** 词法/语法错误通过监听器收集。 语义错误通过Visitor检测。

```
lexer.addErrorListener(&lexerErrorListener);
parser.addErrorListener(&parserErrorListener);
```

- **3.3 结果输出** 输出数字满足要求输出。

```
if (totalErrors == 0) {
    cout << "0" << endl;  // 成功标记
}
```

四、改错的一些过程和思考

本次实验不仅要进行语法分析、还要进行语义分析，在运行测试文件的过程中也遇到了一些遗漏或比较困难的地方。

- CACT要求初值表达式必须是常数，而不能是常数式。最初认为最初认为用一个定义名来赋值另一个定义数是可以的，比如说int a=b;如果提前定义过b的话是可以的，但是后来发现这种应该是错误的，于是进行了改正。

- 还考虑了定义时是否要考虑负数，因为我们的代码逻辑是定义时不允许出现加减乘除符号来保证是常数，如果负数是允许的话则需要更改逻辑，但是考虑到.g4文件中定义的数字只有0~9，测试用例也没有出现负数，经过一些思考我们就认为可以不考虑负数。
- 定义数组时也是比较复杂的检查，因为数组必须包含{}，且要不然全部嵌套，要不然只有一个{}，混合嵌套是不允许的，最后我们采用了根据嵌套维度来判断是否正确。
- 因为我们既要检查语义，而cact中是存在内置函数的，所以我们检查函数时会优先检查是否为内置函数，不过不是的话才会再检查是否已经定义过这个函数。

五、感悟

本次实验让我们更理解了编译器的工作流程，尤其是Lexer、Parser、和语义分析处理机制。通过定义.g4文件语法规则，ANTLR4能自动生成词法分析器和语法分析器，将形式语言理论转化为可以运行的代码。这让我直观感受到理论如何变成实际。

虽然ANTLR4能处理语法结构，但我们只检查语法正确不能满足实际上的的需求。语义规则（如变量作用域、类型检查）还需要手动实现。这让我意识到，语法正确不代表程序合法，逻辑正确的检查不能只是语法的检查。

在实现数组初始化检查时，多维数组的嵌套{}格式、初始值数量计算等细节极易出错。数组定义的检查需要小心检查才能符合要求，否则就难以达到预期。