实验报告:

任务说明

- 熟悉antlr的安装过程和使用;
- 根据给出的CACT文法规范，编写.g4文件，并通过Antlr生成CACT源码的词法-语法分析;
- 覆写Antlr中默认的文法错误处理机制，能检查出源码中的词法语法错误;
- 实现一个简单的编译器前端可以处理输入的cact源码;

成员组成

艾华春, 李霄宇, 王淼

实验设计 (语法规范设计)

设计思路

Cact 语言的语法规范旨在定义一种类 C 的编程语言结构。设计遵循以下思路：

1. **模块化定义**：将语法规则分解为程序结构、声明、函数定义、语句和表达式等几个主要部分。
2. **清晰的层级**：通过产生式规则清晰地展示不同语法结构之间的层级关系和依赖。
3. **ANTLR 兼容性**：所有规则均按照 ANTLR .g4 文件的语法进行编写，以便工具能够正确解析和生成代码。
4. **消除歧义与左递归**：在设计产生式时，注意避免文法歧义，并消除直接或间接左递归，以确保 ANTLR 能够正确处理。

设计编译器的目录结构

```
cc/
├── build/              # 编译后的输出文件
├── deps/               # ANTLR工具安装
├── grammar/            # 文法文件、语法分析器和词法分析器
├── src/                # 编译器主体代码
└── test/               # 测试样例
```

实验实现 (Cact.g4 文件重点规则解析)

Cact 语言的完整 .g4 规范定义了其词法和语法结构。以下将重点解析其中的核心规则。

Parser 规则 (语法分析器)

Parser 规则定义了 Cact 语言的句法结构，即如何将词法单元（Tokens）组合成有意义的程序构造。

1. **顶层结构 (Program Structure)**

```
program     : compUnit;
compUnit    : (decl | funcDef)+ EOF;
```

- `program`: 定义了一个 Cact 程序的基本入口点，它由一个 `compUnit` 组成。
- `compUnit`: 这是编译单元的核心，规定了一个 Cact 程序由一个或多个声明 (`decl`) 或函数定义 (`funcDef`) 序列构成，并以文件结束符 `EOF` 结尾。`+` 表示至少出现一次，体现了程序不能为空。
- 添加 `EOF` 是为了确保整个输入被完全匹配，没有剩余的未解析字符, 提高语法完整性检测能力

2. **声明 (Declarations)**

```
decl        : constDecl | varDecl;
bType       : INT_KW | DOUBLE_KW | CHAR_KW | FLOAT_KW;

constDecl   : CONST_KW bType constDef (COMMA constDef)* SEMICOLON;
constDef    : IDENT (L_BRACKET intConst R_BRACKET)* ASSIGN constInitVal;

varDecl     : bType varDef (COMMA varDef)* SEMICOLON;
varDef      : IDENT (L_BRACKET intConst R_BRACKET)* (ASSIGN
constInitVal)?;
```

- `decl`: 声明可以是常量声明 (`constDecl`) 或变量声明 (`varDecl`)。
- `bType`: 定义了基础数据类型，包括整型 (`INT_KW`)、双精度浮点型 (`DOUBLE_KW`)、字符型 (`CHAR_KW`) 和单精度浮点型 (`FLOAT_KW`)。
- `constDecl`: 常量声明以 `const` 关键字开头，后跟类型和至少一个 `constDef`。常量定义 `constDef` 必须在声明时通过 `ASSIGN` 赋予初始值 `constInitVal`。支持数组形式，数组维度必须是整型常量 `intConst`。
- `varDecl`: 变量声明与常量声明类似，但不使用 `const` 关键字。变量定义 `varDef` 可以在声明时选择性地赋予初始值 (`(ASSIGN constInitVal)?`)。

3. **函数定义 (Function Definitions)**

```
funcDef     : funcType IDENT L_PAREN (funcFParams)? R_PAREN block;
funcType    : bType | VOID_KW;
funcFParams : funcFParam (COMMA funcFParam)*;
funcFParam  : bType IDENT (L_BRACKET intConst? R_BRACKET (L_BRACKET
intConst R_BRACKET)*)?;
```

- `funcDef`: 定义了一个函数结构，包括返回类型 (`funcType`)、函数名 (`IDENT`)、圆括号包裹的参数列表 (`funcFParams`) 以及函数体 (`block`)。
- `funcType`: 函数返回类型可以是基础数据类型 `bType` 或 `void` (`VOID_KW`)。

- `funcFParam`: 函数参数定义了参数的类型和名称。参数可以是普通变量，也可以是数组。对于数组参数，第一个方括号内的维度 `intConst?` 是可选的（表示可以不指定第一维大小），后续维度必须指定。

4. **语句 (Statements)**

```
stmt        : lVal ASSIGN exp SEMICOLON   // 赋值语句
            | block                       // 代码块
            | IF_KW L_PAREN cond R_PAREN stmt (ELSE_KW stmt)?  // if-else 语句
            | WHILE_KW L_PAREN cond R_PAREN stmt // while 循环语句
            | RETURN_KW exp? SEMICOLON          // return 语句
            | exp? SEMICOLON;                   // 表达式语句（如函数调用或空语句）
```

- `stmt` 规则是 Cact 语言中执行单元的核心。它通过 `|` (或)操作符定义了多种语句类型。
- **赋值语句**: `lVal ASSIGN exp SEMICOLON`，将表达式 `exp` 的值赋给左值 `lVal` (通常是变量或数组元素)。
- **条件语句**: `IF_KW L_PAREN cond R_PAREN stmt (ELSE_KW stmt)?`，实现了标准的 `if-else` 逻辑，`else` 部分可选。
- **循环语句**: `WHILE_KW L_PAREN cond R_PAREN stmt`，实现了当条件 `cond` 满足时重复执行语句 `stmt` 的 `while` 循环。
- **返回语句**: `RETURN_KW exp? SEMICOLON`，用于从函数返回，可以带有可选的返回值表达式 `exp`。
- **表达式语句**: `exp? SEMICOLON`，允许一个表达式（如函数调用）自成一条语句，或者一个单独的分号表示空语句。

5. **表达式的优先级与结合性 (Expression Hierarchy)**

Cact 语言通过一系列级联的规则来定义表达式的优先级和结合性，这是确保表达式能被正确解析的关键。规则从低优先级到高优先级排列，高优先级的运算会先被组合。

```
exp         : addExp;
cond        : lOrExp; // 条件表达式本质是逻辑或表达式

lOrExp      : lAndExp (OR lAndExp)*;        // 逻辑或（最低优先级）
lAndExp     : eqExp (AND eqExp)*;           // 逻辑与
eqExp       : relExp (eqOp relExp)*;        // 等于/不等于
relExp      : addExp (relOp addExp)*;       // 关系运算（>, <, >=, <=）
addExp      : mulExp (addOp mulExp)*;       // 加减运算
mulExp      : unaryExp (mulOp unaryExp)*;   // 乘除模运算
unaryExp    : primaryExp                    // 一元运算（包括正负号，逻辑非）
            | unaryOp unaryExp
            | IDENT L_PAREN funcRParams? R_PAREN; // 函数调用
primaryExp  : L_PAREN exp R_PAREN           // 括号表达式
            | lVal                         // 左值（变量或数组元素）
            | number;                      // 字面量
```

- `exp`: 是表达式的总入口，通常直接指向加法/减法表达式层级 `addExp`。
- `cond`: 条件表达式，其顶层是逻辑或表达式 `lOrExp`。

- **级联结构**: 每个表达式规则 (如 `lOrExp`) 通常由更高优先级的表达式规则 (如 `lAndExp`) 和相应的操作符组成。例如 `addExp : mulExp (addOp mulExp)*;` 表示一个加法/减法表达式由一个或多个乘法/除法表达式通过加法或减法操作符 (`addOp`) 连接而成。`*` 表示操作符和后续的 `mulExp` 可以出现零次或多次，这自然地处理了左结合性。
- `unaryExp`: 处理一元运算符 (`+`, `-`, `!`) 和函数调用。函数调用 `IDENT L_PAREN funcRParams? R_PAREN` 被放在一元表达式层级，具有较高优先级。
- `primaryExp`: 是最高优先级的表达式，包括括号包裹的表达式、左值（变量、数组元素）和数字字面量。

Lexer 规则 (词法分析器)

Lexer 规则定义了如何将输入的字符流分割成一个个有意义的词法单元 (Tokens)。

1. **关键字 (Keywords)**

```
CONST_KW    : 'const';
INT_KW      : 'int';
// ... 其他关键字 ...
RETURN_KW   : 'return';
```

   - 关键字通过直接匹配字符串字面量来定义。例如，当词法分析器遇到字符序列 `const` 时，会将其识别为 `CONST_KW` 这个 Token。这些 Token 在 Parser 规则中用于识别特定的语言结构。

2. **标识符 (Identifiers)**

```
IDENT       : [a-zA-Z_][a-zA-Z_0-9]*;
```

   - `IDENT`: 定义了标识符的构成规则。它必须以字母 (大小写) 或下划线 `_` 开头，后续可以跟任意数量的字母、数字或下划线。这是变量名、函数名等的通用规则。

3. **常量 (Constants)**

```
intConst    : DECIMAL_CONST | OCTAL_CONST | HEXADECIMAL_CONST;
DECIMAL_CONST: [1-9][0-9]*;
OCTAL_CONST : '0' [0-7]*;
HEXADECIMAL_CONST: ('0x' | '0X') [0-9a-fA-F]+;

FloatConst  : [0-9]* '.' [0-9]+ [fF]? | [0-9]+ '.' [0-9]* [Ff]?;
EXPONENT    : (FloatConst|DECIMAL_CONST) [eE] [+\-]? [0-9]+ [fF]?;
CharConst   : '\'' REGULAR_CHAR '\'';
```

   - ```
     intConst
     ```

     : 整数常量可以是十进制 (

     ```
     DECIMAL_CONST
     ```
     ```

)、八进制 (

```
OCTAL_CONST
```

) 或十六进制 (

```
HEXADECIMAL_CONST
```

)。

- `DECIMAL_CONST`: 十进制数，由非零数字开头后跟任意数字. (纯0 的时候认为是八进制数)
- `OCTAL_CONST`: 八进制数以 `0` 开头，后跟任意位 0-7 的数字。
- `HEXADECIMAL_CONST`: 十六进制数以 `0x` 或 `0X` 开头，后跟一位或多位 0-9, a-f, A-F 的数字。

- `FloatConst`: 定义了浮点数的格式，包括小数点前后的数字以及可选的 `f` 或 `F` 后缀。
- `EXPONENT`: 定义了科学计数法表示的浮点数，可以基于 `FloatConst` 或 `DECIMAL_CONST` 加上指数部分。
- `CharConst`: 字符常量由单引号 `'` 包围的单个 `REGULAR_CHAR` 组成。`REGULAR_CHAR` 和 `ESC` fragment 定义了可接受的字符和转义序列。

4. **操作符与分隔符 (Operators and Punctuation)**

```
PLUS        : '+';
ASSIGN      : '=';
SEMICOLON   : ';';
// ... 其他操作符和分隔符 ...
```

- 这些规则简单地将特定的符号映射为对应的 Token，如 `+` 变为 `PLUS`，`=` 变为 `ASSIGN` 等。

5. **空白与注释 (Whitespace and Comments)**

```
WS          : [ \t\r\n]+ -> skip ;
LineComment : '//' ~[\r\n]* -> skip;
BlockComment: '/*' .*? '*/' -> skip;
```

- `WS`: 匹配一个或多个空格、制表符、回车或换行符。`-> skip` 指令告诉词法分析器忽略这些空白字符，不将它们传递给语法分析器。
- `LineComment`: 匹配以 `//` 开头的行注释，直到行尾。同样使用 `-> skip` 忽略。
- `BlockComment`: 匹配以 `/*` 开始并以 `*/` 结束的块注释。`.*?` 是一个非贪婪匹配，确保它在遇到第一个 `*/` 时结束。同样被忽略。

语法和词法错误处理

替换默认错误监听器, 使得的只要语法有错，立即报错退出。

```cpp
class BailErrorListener : public antlr4::BaseErrorListener {
public:
    void syntaxError(antlr4::Recognizer *recognizer,
                     antlr4::Token *offendingSymbol,
                     size_t line,
                     size_t charPositionInLine,
                     const std::string &msg,
                     std::exception_ptr e) override {

        std::cerr << "Syntax Error at line " << line << ":" << charPositionInLine
                  << " - " << msg << std::endl;

        exit(EXIT_FAILURE);  // 错误发生时直接终止程序
    }
};


int main(){
    // ...

    antlr4::Parser parser(tokens);
    parser.removeErrorListeners();  // 去掉默认监听器
    parser.addErrorListener(new BailErrorListener());
    // ...
}
```

总结

实验结果总结

- **程序基本骨架**: 定义了编译单元由声明和函数定义组成。
- **数据类型与声明**: 支持 `int`, `double`, `char`, `float` 等基本类型，以及常量和变量的声明，包括数组和初始化。
- **函数机制**: 详细规定了函数的定义、参数（包括数组参数）和返回类型。
- **控制流程**: 实现了赋值、条件（`if-else`）、循环（`while`）、跳转（`break`, `continue`, `return`）等核心语句。
- **表达式解析**: 通过明确的级联规则有效地处理了各种运算符的优先级和结合性，包括算术、关系、逻辑运算以及函数调用。
- **词法元素**: 精确定义了关键字、标识符、各种类型的数字和字符常量、操作符、分隔符，并合理处理了空白和注释。

此 .g4 文件是构建 Cact 编译器前端的关键，为后续的语义分析和代码生成阶段提供了的句法基础。