中国科学院大学《编译原理》
CACT 词法语法分析实验报告

小组编号 15 小组成员姓名 李紫嫣 梁婧琪 邹为 实验编号 1

实验名称 PR001-CACT 日期 2025.4.17

1 实验要求

熟悉 ANTLR 的安装和使用

了解 ANTLR 工具生成词法-语法源码的能力, 掌握 ANTLR 生成 lexer 和 parser 的流程。搭建 ANTLR 环境并正确运行课程提供的 demo。

完成词法和语法分析

根据 CACT 文法规范编写 ANTLR 文法文件 (g4), 并通过 Antlr 生成 CACT 源码的词法-语法分析。修改 ANTLR 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, 并返回分析结果(0 与非 0 值)。

2 实验设计与思路

2.1 g4 文件设计

按照给定的 cact-spec 中的规则介绍, 使用 ANTLR 语法定义 C 风格语言 CACT 的语法规则, 包括编译单元、声明、函数定义、表达式、语句、条件语句等成分, 以及词法规则如标识符、常量等。

2.1.1 语法规则分析

- **编译单元**

```
compUnit : (decl | funcDef)(compUnit)? EOF ;
```

表示编译单元由若干声明(`decl`)或函数定义(`funcDef`)组成, 递归形式支持多个声明/函数定义。

- **声明部分**

即既有常量声明, 又有变量声明。

```
decl : constDecl | varDecl ;

constDecl : 'const' bType constDef (',' constDef)* ';' ;
```

常量声明以 `const` 开头, 指定基本类型, 后接一个或多个常量定义, 使用逗号分隔。

```
varDecl : bType varDef (',' varDef)* ';' ;
```

变量声明类似常量声明, 但不以 `const` 开头。并支持初始化或数组形式。

- **函数定义**

```
funcDef : funcType Ident '(' (funcParams)? ')' block ;
```

描述函数的返回类型、名称、形参以及函数体。

- **语句结构**

包括赋值、表达式语句、块语句、控制流语句等。

```
stmt
    : lVal '=' exp ';'
    | (exp)? ';'
    | block
    | 'return' (exp)? ';'
    | 'if' '(' cond ')' stmt ('else' stmt)?
    | 'while' '(' cond ')' stmt
    | 'break' ';'
    | 'continue' ';' ;
```

- **表达式分析**

  采用递归下降结构,使用消除左递归的技巧:

  - addExp、mulExp 等运算表达式支持多级二元运算;
  - relExp、eqExp、lAndExp、lOrExp 构成条件表达式的优先级结构;
  - unaryExp 同时支持一元运算符和函数调用;
  - primaryExp 是表达式最基本的单元;

- **左值**

```
lVal : Ident ( '[' exp ']' )* ;
```

支持变量名和多维数组访问形式。

- **语句块与函数体**

```
block : '{' (blockItem)* '}' ;
blockItem : decl | stmt ;
```

这样一个块可以包含多个声明或语句。

2.1.2 词法规则分析

spec 中写明了大部分的词法规则,只需要按照 antlr4 的语法转化实现,另外需要自行补充的是关于浮点数的规则。为了减少生成太多不必要的 token,使用了 fragment 来辅助规则定义。

- **词法汇总**

  - 跳过空白符(空格、Tab)、换行符、注释;
  - Ident 定义了标识符规则;
  - IntConst 支持十进制、八进制、十六进制;
  - FloatConst 支持 IEEE 浮点格式,分为十进制和十六进制形式;
  - CharConst 使用单引号括住一个字符;

- **浮点数的设计思路补充**

  十进制浮点数支持两种形式:

  如果是 FractionalConstant(小数部分)开头,后接可选的指数部分(ExponentPart)以及必要的浮点后缀(FloatingSuffix);如果是整数部分(DigitSequence)开头,后接必须存在的指数部分和浮点后缀。

  小数部分的设计支持:

可省略整数部分，如 .123；

可省略小数部分，如 123.；

同时具备整数和小数部分，如 123.456。

指数部分(ExponentPart)形式为 e/E 后接可选符号(-/+)和整数。

浮点后缀(FloatingSuffix)表示数据类型后缀，f/F 表示 float 类型，或者 l/L 来表示 long double 类型。

十六进制浮点数以 0x 或 0X 开头，通过两种方式表示：

十六进制小数(HexadecimalFractionalConstant)+ 二进制指数部分(BinaryExponentPart)+ 可选的浮点后缀；

十六进制整数(HexadecimalDigitSequence)+ 二进制指数部分 + 可选的浮点后缀。

二进制指数部分(BinaryExponentPart)由 p/P 开头，后接可选的符号和十进制数字。

浮点后缀与十进制浮点数相同。

2.2 错误的检测与输出

Antlr 生成的词法、语法分析器已经提供了错误计数器和处理错误的接口，所以需要补充的是在正确的地方捕捉错误并处理。

2.2.1 作用域管理

使用 std::vector<SymbolTable> scopeStack 模拟作用域的栈结构，进入一个作用域（如访问新的函数）时 scopeStack.push_back，创建一个新的符号表来记录该作用域内的符号；离开作用域时 scopeStack.pop_back() 退出当前作用域。这是为了在同一作用域内重复定义时报错。

在使用 visitVarDecl/visitVarDecl 函数访问一个变量/常量定义时，会试图将新的常量或变量加入符号表。如果当前作用域的符号表已经存在这个常量或变量，就添加失败，返回错误。

2.2.2 常量/变量初始化合法性检查

在常量/变量的初始化时，初值表达式必须是常数。

在用 visitVarDecl/visitVarDecl 函数访问一个变量/常量定义时，调用 checkConstInitVal 递归地检查初始化值（ConstInitValContext）中的每个元素，确保它们是常量表达式。isConstLiteralExp 函数判断常量表达式是否为字面量常量。它通过遍历 CACTParser::ConstExpContext 的结构，确保表达式是简单的常量（例如，整数常量、浮点数常量或字符常量）。

2.2.3 常量/变量初始化类型检查

getConstExpType(ctx) 用于获取常量表达式类型，它识别三种字面量类型（int、float、char）用于后续初始化检查中的类型匹配；primaryExp() 推导变量或表达式的类型，支持对变量的类型推导，未声明变量会产生错误信息。

```
if (number->IntConst()) return "int";
if (number->FloatConst()) return "float";
if (number->CharConst()) return "char";



if (primary->lVal()) {
    std::string varName = primary->lVal()->Ident()->getText();
    for (...) {
```

```
        if (it->contains(varName)) return it->getType(varName);
    }
    errors.push_back("Undeclared variable: " + varName);
}
```

在用 visitVarDecl/visitVarDecl 函数访问一个变量/常量定义时,调用上面的函数以判断变量类型

```
if (!initVal->constExp()) {
    errors.push_back("Invalid scalar const initialization");
}
else {
    std::string literalType = getConstExpType(initVal->constExp());
    if (literalType != baseType) {
        errors.push_back("Type mismatch: expected " + baseType + ", got " + literalType);
    }
}
```

如果类型不完全一致,就会报错。

2.2.4 函数返回类型检查

函数返回时也需要检查返回值数据类型和函数类型是否一致。

currentFunctionReturnTypeStack 用于在访问一个函数定义时,将它的类型压入栈中。

```
std::any visitFuncDef(CACTParser::FuncDefContext *ctx) override {
    std::string returnType = ctx->funcType()->getText();
    currentFunctionReturnTypeStack.push_back(returnType);
    ...
    currentFunctionReturnTypeStack.pop_back();
    return nullptr;
}
```

真正用于检查函数返回类型和报错的部分在 visitStmt 种。如果是一个 return 语句,就进入返回语句检查:

```
    if (currentFunctionReturnTypeStack.empty()) {
        errors.push_back("Return statement outside of function");
        return nullptr;
    }
```

如果没有当前函数上下文(比如 return 出现在函数外),报错;如果 return 返回了一个值,首先检查 void 函数是否返回值(不允许返回值),否则检查返回值的表达式类型是否与函数返回类型匹配;如果 return 后没有表达式,如果不是 void 函数,就报错。

2.2.5 常量/变量赋值语句合法性、类型检查

和上面的初始化相似,赋值语句也需要保证类型一致。首先,从作用域栈中查找变量定义,如果左值类型已知,检查右值表达式类型是否一致,如果类型不匹配,则报错。

```
    for (auto it = scopeStack.rbegin(); it != scopeStack.rend(); ++it) {
        if (it->contains(name)) {
            lvalType = it->getType(name);
```

```
            break;
        }
    }
```

```
    else {
        std::string rtype = getExpType(ctx->exp());
        if (rtype != lvalType && rtype != "unknown") {
            errors.push_back("Type mismatch in assignment to '" + name +
                             "': expected " + lvalType + ", got " + rtype);
        }
    }
```

3 Debug 分析

3.1 文件结束符 EOF 的处理

在运行第一个测试文件时:

```
int main()
{
   int a = 0;
    return 0;
}
```

出现了这样的报错:

```
compiler15@teacher-PowerEdge-M640:~/cact/build\$ ./compiler
    ../test/samples_lex_and_syntax/00_true_main.cact
Syntax error at 3:5 - mismatched input 'a' expecting Ident
Syntax error at 6:0 - no viable alternative at input '<EOF>'
Segmentation fault (core dumped)
```

首先,词法分析器无法将字符 a 识别为 Ident,经检查后发现是因为,Tokens 优先级设置不对。其次,分析器在文件结束时无法处理 EOF,是因为语法规则中的某些部分未正确覆盖所有情况。

主要是修改了 CACT.g4 文件中的内容

```
compUnit
    : (decl | funcDef)* EOF // 明确要求解析到文件结束
    ;
```

3.2 无法正确的识别二维数组

修改 g4 文件:

```
    // 原规则
varDef
    : Ident ('[' IntConst ']')* ('=' constInitVal)?
    ;
```

```
// 修改后：明确区分多维数组和一维数组
varDef
    : Ident ('[' IntConst ']')+ ('=' constInitVal)?  #multiDimArray // 必须至少有一个维度
    | Ident ('=' constInitVal)?                      #singleVar
    ;

// 原规则
constInitVal
    : constExp
    | '{' (constInitVal(','constInitVal)*)? '}'
    ;

// 修改后：支持嵌套初始化列表
constInitVal
    : constExp
    | '{' (constInitVal (',' constInitVal)* )? '}'
    ;
```

修改 main.cpp

```
// 在 visitVarDecl 中添加多维数组支持
   std::any visitVarDecl(CACTParser::VarDeclContext *ctx) override {
      std::string baseType = ctx->bType()->getText();
      for (auto def : ctx->varDef()) {
         std::string name = def->Ident()->getText();
         std::string fullType = baseType;
         std::vector<tree::TerminalNode*> dims;
         for (auto dim : def->IntConst()) dims.push_back(dim);
         if (!dims.empty()) fullType += processArrayDims(dims);
//其他逻辑
```

3.3 变量声明规则完善

CACT 在显式变量声明时，不允许等号右边出现除数值外的其他符号，包括但不限于其他变量名（如 a、b 等等）和算式（如 2*3），为解决这个问题，我们在 main.cpp 中添加了 isConstLiteralExp 函数用于进行相关检查。

```
// 改进后的常量字面量检查函数
bool isConstLiteralExp(CACTParser::ConstExpContext* ctx) {
   if (!ctx || !ctx->addExp()) return false;

   auto addExp = ctx->addExp();
   // 必须是单个mulExp，不能有加减运算
   if (addExp->mulExp().size() != 1) return false;

   auto mulExp = addExp->mulExp(0);
   // 必须是单个unaryExp，不能有乘除模运算
   if (mulExp->unaryExp().size() != 1) return false;
```

```
    auto unaryExp = mulExp->unaryExp(0);
    // 不能有正负非运算符
    if (unaryExp->children.size() != 1) return false;

    auto primaryExp = unaryExp->primaryExp();
    if (!primaryExp) return false;

    // 必须是数字常量或字符常量
    return primaryExp->number() != nullptr;
}
```

3.4 定义顺序错误导致翻译优先级错误

如果两个规则能匹配相同的输入，ANTLR 会优先选择在文件中“先定义”的那一个规则。在 debug 的过程中，我们因为优先级考虑不周全导致了很多翻译问题，例如：

```
fragment FloatingSuffix
    : 'f' | 'F' | 'l' | 'L'
    ;
```

原本，用于匹配浮点数的”片段“如 FloatingSuffix，没有加上 fragment 记号，而且放在了 Ident 等词法规则之前。它们被 ANTLR 当作一个完整的 token 类型，加入了词法分析阶段的决策中，运行中就经常出现简单赋值语句被错误分割的情形。多次调整顺序后不再发生。

3.5 其余 g4 文件编写错误补充

测试时在报错中发现不应该出现的 token“number”，检查后原来是误将 parser 规则名首字母大写了；另外对注释的处理忘记加 skip 导致测试时对于注释的词法分析报错。