实验一 基于 ANTLR4 的抽象语法树生成

姓名：邱明阳 王思源

日期：2024 年 12 月 2 日

1 实验内容

1. 熟悉 ANTLR4 语法描述。
2. 熟悉 ANTLR4 访问者模式和访问者模式下的调试方法。
3. 熟悉 C++类型转换与继承相关知识。

2 实验内容

2.1 LAB1-1

1. 完善 Safe C 语言的 ANTLR4 词法文件(SafeCLexer.g4)，生成 token 流。
2. 完善 Safe C 语言的 ANTLR4 语法文件(SafeCParser.g4)，生成语法分析树。
3. 基于语法理解，自行编写正确样例和错误样例对上述实现进行测试。

2.2 LAB1-2

根据已有框架，修改 AstBuilder.cpp，实现对语法分析树的遍历，生成 Json 表示的抽象语法树。

3 实验过程

3.1 LAB1-1

3.1.1 词法分析器

词法文件将源代码中的字符流分解为词法单元（tokens），为语法分析器（Parser）的进一步处理做好准备。笔者完善了 Safe C 语言的 ANTLR4 词法文件（SafeCLexer.g4），以下为基本文法。

```
Comma: ',';
SemiColon: ';';
```

```
Assign: '=';

LeftBracket: '[';
RightBracket: ']';
LeftBrace: '{';
RightBrace: '}';
LeftParen: '(';
RightParen: ')';

If: 'if';
Else: 'else';
While: 'while';

Equal: '==';
NonEqual: '!=';
Less: '<';
Greater: '>';
LessEqual: '<=';
GreaterEqual: '>=';

Plus: '+';
Minus: '-';
Multiply: '*';
Divide: '/';
Modulo: '%';

Int: 'int';
Void: 'void';
Obc: 'obc';
Const: 'const';

Identifier: [_a-zA-Z][a-zA-Z0-9_]*;
IntConst: ('0x' | '0X') [0-9a-fA-F]+ | [0-9]+;

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

实现词法文件后，进行编译。获取测例 0.c 的 token 流，如下所示。

```
antlr SafeCLexer.g4
javac SafeCLexer.java
grun SafeCLexer tokens -tokens ../tests/0.c
```

```
armin@armin-virtual-machine:~/work/github/2024-fall-CPP/Compiler-lab1/Lab1/Lab1-1/grammar$
grun SafeCLexer tokens -tokens ../tests/0.c
[@0,44:48='const',<'const'>,4:0]
[@1,50:52='int',<'int'>,4:6]
[@2,54:54='a',<Identifier>,4:10]
[@3,56:56='=',<'='>,4:12]
[@4,58:58='3',<IntConst>,4:14]
[@5,59:59=';',<';'>,4:15]
[@6,61:65='const',<'const'>,5:0]
[@7,67:69='int',<'int'>,5:6]
[@8,71:71='b',<Identifier>,5:10]
[@9,73:73='=',<'='>,5:12]
[@10,75:75='+',<'+'>,5:14]
[@11,76:77='13',<IntConst>,5:15]
[@12,78:78=',',<','>,5:17]
[@13,80:80='c',<Identifier>,5:19]
[@14,82:82='=',<'='>,5:21]
[@15,84:84='-',<'-'>,5:23]
[@16,85:88='0x13',<IntConst>,5:24]
[@17,89:89=';',<';'>,5:28]
[@18,90:89='<EOF>',<EOF>,5:29]
```

图 1：获取测例 0.c 的 token 流

3.1.2 语法分析器

语法分析器解析经过词法分析器处理后的 token 流，其描述了 SafeC 语言的语法规则和程序结构。笔者完善了语法文件 SafeCParser.g4，产生式如下所示。

```
compUnit → (decl | funcDef) + EOF
decl → constDecl | varDecl
funcDef → Void Identifier '(' ')' block
constDecl → Const bType constDef (',' constDef)* ';'
constDef → Identifier '=' exp
         | array '=' '{' exp (',' exp)* '}'
varDecl → bType varDef (',' varDef)* ';'
bType → Int
varDef → Identifier ('=' exp)?
       | array ('=' '{' exp (',' exp)* '}')?
array → obcArray | unobcArray
obcArray → Obc unobcArray
unobcArray → Identifier '[' exp? ']'
block → '{' blockItem* '}'
blockItem → decl | stmt
stmt → block
      | ';'
      | Identifier '(' ')' ';'
      | lval '=' exp ';'
      | If '(' cond ')' stmt (Else stmt)?
      | While '(' cond ')' stmt
cond → '(' cond ')'
      | exp (Equal | NonEqual | Less | Greater | LessEqual | GreaterEqual) exp
lval → Identifier ('[' exp ']')?
number → IntConst
exp → exp (Multiply | Divide | Modulo) exp
    | exp (Plus | Minus) exp
```

```
    | (Plus | Minus) exp
    | lval
    | number
    | LeftParen exp RightParen;
```

Safe C 的语法特点总结如下：

- 基本语言结构：如变量声明、常量定义、函数定义。
- 控制流：包括 if-else 分支、while 循环等。
- 表达式支持：支持算术运算、比较运算和括号优先级。
- 数组机制：支持两种数组类型（obcArray 和 unobcArray）。

值得注意的是，根据笔者后续的实验结果表示，语法文件中产生式分支的先后顺序可能会对后续语法树的产生造成较大的影响。

实现语法文件后，进行编译。生成 0.c 的语法树，如下所示。

```
antlr SafeCParser.g4
javac SafeCParser.java
grun SafeC compUnit -gui ../tests/0.c
```

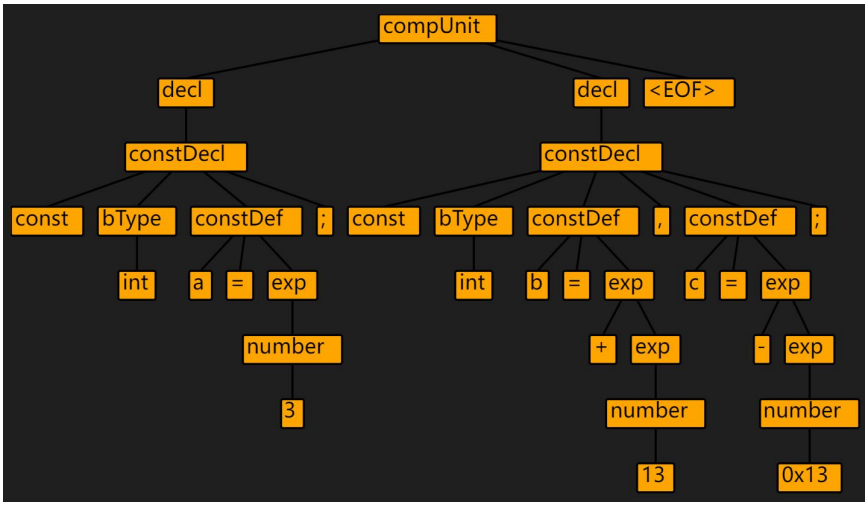

图 2：生成 0.c 的语法树

3.1.3 样例测试

实现并编译了 Safe C 的词法文件和语法文件后，运行 go.sh 进行批量测试，全部通过。

```
armin@armin-virtual-machine:~/work/github/2024-fall-CPP/Compiler-lab1/Lab1/Lab1-1$ sudo ./go.sh
[sudo] armin 的密码:
Compiling...
```

```
grun SafeC compUnit -tree ../tests/0.c > ../output/0.c.output
grun SafeC compUnit -tree ../tests/10.c > ../output/10.c.output
grun SafeC compUnit -tree ../tests/11.c > ../output/11.c.output
grun SafeC compUnit -tree ../tests/12.c > ../output/12.c.output
grun SafeC compUnit -tree ../tests/13.c > ../output/13.c.output
grun SafeC compUnit -tree ../tests/14.c > ../output/14.c.output
grun SafeC compUnit -tree ../tests/15.c > ../output/15.c.output
grun SafeC compUnit -tree ../tests/16.c > ../output/16.c.output
grun SafeC compUnit -tree ../tests/17.c > ../output/17.c.output
grun SafeC compUnit -tree ../tests/18.c > ../output/18.c.output
grun SafeC compUnit -tree ../tests/19.c > ../output/19.c.output
grun SafeC compUnit -tree ../tests/1.c > ../output/1.c.output
grun SafeC compUnit -tree ../tests/2.c > ../output/2.c.output
grun SafeC compUnit -tree ../tests/3.c > ../output/3.c.output
grun SafeC compUnit -tree ../tests/4.c > ../output/4.c.output
grun SafeC compUnit -tree ../tests/5.c > ../output/5.c.output
grun SafeC compUnit -tree ../tests/6.c > ../output/6.c.output
grun SafeC compUnit -tree ../tests/7.c > ../output/7.c.output
grun SafeC compUnit -tree ../tests/8.c > ../output/8.c.output
grun SafeC compUnit -tree ../tests/9.c > ../output/9.c.output
```

图 3：批量测试

3.2 LAB1-2

3.2.1 抽象语法树生成

AstBuilder 从语法分析树生成抽象语法树（AST）,其目的是将具体的代码结构转换为更易于语义分析或代码生成的抽象表达形式。其处理流程为：

1. 通过 visit 方法，访问语法分析树的节点。
2. 构建对应的抽象语法树节点。
3. 递归处理子节点，构建出完整的树状结构。

分析树不同节点的访问逻辑是不同的，本实验需要对语法规则补充自定义的构建逻辑。于是笔者参考 AstNode.h 对抽象语法树节点的定义，根据已有框架，完善 AstBuilder.cpp，补充完成各 visitXXX 函数，完成对语法分析树的遍历。

下面以 visitExp 函数为例。

```
antlrcpp::Any AstBuilder::visitExp(SafeCParser::ExpContext* ctx) {
    auto exps = ctx->exp();
    if (ctx->lval()) { // lval
        auto result = visit(ctx->lval()).as<lval_node*>();
        return dynamic_cast<expr_node*>(result);
    } else if (ctx->number()) { // number
        auto result = visit(ctx->number()).as<number_node*>();
        return dynamic_cast<expr_node*>(result);
    } else if (exps.size() == 2) { // exp(Plus|Minus|Multiply|Divide|Modulo)exp
        auto result = new binop_expr_node;
        result->line = ctx->getStart()->getLine();
        result->pos = ctx->getStart()->getCharPositionInLine();
```

```
        if (ctx->Plus()) {
            result->op = BinOp::PLUS;
        } else if (ctx->Minus()) {
            result->op = BinOp::MINUS;
        } else if (ctx->Multiply()) {
            result->op = BinOp::MULTIPLY;
        } else if (ctx->Divide()) {
            result->op = BinOp::DIVIDE;
        } else if (ctx->Modulo()) {
            result->op = BinOp::MODULO;
        }
        result->lhs.reset(visit(exps[0]).as<expr_node*>()); // 左表达式
        result->rhs.reset(visit(exps[1]).as<expr_node*>()); // 右表达式

        return dynamic_cast<expr_node*>(result); // 返回二元表达式节点
    } else if (exps.size() == 1) { // (Plus | Minus) exp
        auto result = new unaryop_expr_node;
        result->line = ctx->getStart()->getLine(); // 行号
        result->pos = ctx->getStart()->getCharPositionInLine(); // 列号

        if (ctx->Plus()) {
            result->op = UnaryOp::PLUS;
        } else if (ctx->Minus()) {
            result->op = UnaryOp::MINUS;
        }
        result->rhs.reset(visit(exps[0]).as<expr_node*>()); // 右表达式

        return dynamic_cast<expr_node*>(result); // 返回一元表达式节点
    } else if (ctx->LeftParen() && ctx->RightParen()) { // LeftParen exp RightP
aren
        return visit(ctx->exp(0));
    }else {
        assert(0 && "Unknown exp.");
    }
}
```

visitExp 函数用于处理 Exp 规则，解析表达式并构建相应节点，支持以下情况：左值表达式、数值表达式、二元运算表达式、一元运算表达式、括号表达式。处理逻辑如下：

- 获取子表达式列表（其大小决定了表达式形式）
- 若包含 lval 子规则，则为左值表达式。
    - 调用 visit(ctx->lval()) 递归解析左值节点，转换类型后返回。

- 否则，若包含 number 子规则，则为数值表达式。
  - 调用 visit(ctx->number()) 递归解析数字节点，转换类型后返回。
- 否则，若子表达式数量为 2，则为二元运算表达式。
  - 创建 binop_expr_node 节点，记录行列号。
  - 根据包含的运算符子规则，设置操作符 result->op。
  - 递归调用 visit 解析左表达式和右表达式，分别赋值给 lhs 和 rhs。
  - 转换类型后返回。
- 否则，若子表达式数量为 1，则为一元运算表达式。处理参考二元。
- 否则，若包含左右括号，直接递归解析括号内的子表达式并返回。

3.2.2 样例测试

对于编译后的 astbuilder，运行 go.sh 进行批量测试，无报错发生，通过测试。

```
armin@armin-virtual-machine:~/work/github/2024-fall-CPP/Compiler-lab1/Lab1/Lab1-2$ sudo ./go.sh
[sudo] armin 的密码：
scan ./tests/decl
./build/astbuilder ./tests/decl/const_obcarray_decl_test1.c > ./output/decl/const_obcarray_decl_test1.c.output
./build/astbuilder ./tests/decl/const_unobcarray_decl_test1.c > ./output/decl/const_unobcarray_decl_test1.c.output
./build/astbuilder ./tests/decl/const_var_decl_test1.c > ./output/decl/const_var_decl_test1.c.output
./build/astbuilder ./tests/decl/global_const_obcarray_decl_test1.c > ./output/decl/global_const_obcarray_decl_test1.c.output
./build/astbuilder ./tests/decl/global_const_unobcarray_decl_test1.c > ./output/decl/global_const_unobcarray_decl_test1.c.output
./build/astbuilder ./tests/decl/global_const_var_decl_test1.c > ./output/decl/global_const_var_decl_test1.c.output
./build/astbuilder ./tests/decl/global_obcarray_decl_test1.c > ./output/decl/global_obcarray_decl_test1.c.output
./build/astbuilder ./tests/decl/global_unobcarray_decl_test1.c > ./output/decl/global_unobcarray_decl_test1.c.output
./build/astbuilder ./tests/decl/global_var_decl_test1.c > ./output/decl/global_var_decl_test1.c.output
./build/astbuilder ./tests/decl/obcarray_decl_test1.c > ./output/decl/obcarray_decl_test1.c.output
./build/astbuilder ./tests/decl/unobcarray_decl_test1.c > ./output/decl/unobcarray_decl_test1.c.output
./build/astbuilder ./tests/decl/var_decl_test1.c > ./output/decl/var_decl_test1.c.output
scan ./tests/expr
./build/astbuilder ./tests/expr/expr_test1.c > ./output/expr/expr_test1.c.output
./build/astbuilder ./tests/expr/expr_test2.c > ./output/expr/expr_test2.c.output
scan ./tests/ifelse
./build/astbuilder ./tests/ifelse/if_else_test1.c > ./output/ifelse/if_else_test1.c.output
./build/astbuilder ./tests/ifelse/if_else_test2.c > ./output/ifelse/if_else_test2.c.output
./build/astbuilder ./tests/ifelse/if_else_test3.c > ./output/ifelse/if_else_test3.c.output
./build/astbuilder ./tests/ifelse/if_else_test4.c > ./output/ifelse/if_else_test4.c.output
scan ./tests/stmt
./build/astbuilder ./tests/stmt/assign_stmt_test1.c > ./output/stmt/assign_stmt_test1.c.output
./build/astbuilder ./tests/stmt/block_test1.c > ./output/stmt/block_test1.c.output
./build/astbuilder ./tests/stmt/func_call_test1.c > ./output/stmt/func_call_test1.c.output
./build/astbuilder ./tests/stmt/func_call_test2.c > ./output/stmt/func_call_test2.c.output
./build/astbuilder ./tests/stmt/math_test1.c > ./output/stmt/math_test1.c.output
scan ./tests/while
./build/astbuilder ./tests/while/while_test1.c > ./output/while/while_test1.c.output
./build/astbuilder ./tests/while/while_test2.c > ./output/while/while_test2.c.output
```

图 4：批量测试

4 实验问题

在最初进行实验时，笔者以为 visitXXX 函数中 if else 语句块的顺序并不会对语法分析树的遍历结果产生影响。以 visitStmt 函数为例，笔者在函数中先执行 ctx->SemiColon() 语句，也就是判断是否有分号子表达式，事实上该语句只能在及其靠后的顺序进行，因为有分号子表达式不代表整个语句就可以直接化为空语句。事实上越“充分”的条件处理优先级

应当越靠前，当优先级高的条件不满足时再依次执行其余条件，这样才能保证处理的正确进行。后来当笔者对 SafeC 语法有了更加深刻的了解后，便将判断空语句的条件分支放到顺序靠后的位置，解决了问题。

实验 1 中 exp 的产生式子式的先后顺序会对实验 2 的运行结果产生很大影响，而这在实验 1 中无法通过测例发现，笔者后续进行了更正。

另外，由于 C++类型转换不当，补充代码时出现了波浪形曲线，提示类型不匹配。结合 AstNode.h 等文件进行检查，最终解决了该问题。

5 实验总结

本次实验笔者完善了 Safe C 语言的 ANTLR4 词法、语法文件，实现了 token 流、语法分析树的生成。另外还根据已有框架，完善了 AstBuilder.cpp，补充完成各 visitXXX 函数，实现对语法分析树的遍历，生成 Json 表示的抽象语法树。

通过本次实验，笔者学习了 ANTLR4 的语法描述，并初步掌握了 ANTLR4 访问者模式和访问者模式下的调试方法。同时还了解了 C++类型转换与继承相关知识。笔者进一步研究 AstBuilder，学习到了其面向对象的节点访问、解耦分析树与抽象语法树、递归构建模式等设计思想，这对笔者以后的学习和科研工作都有很大的帮助。