中国科学院大学网络空间安全学院
编译原理
实验报告

姜俊彦[1], 刘镇豪[2]

目录

[1]学号：2022K8009970011，邮箱：jiangjunyan22@mails.ucas.ac.cn
[2]学号：2022K8009929027，邮箱：liuzhenhao22@mails.ucas.ac.cn

1 实验一

1.1 题目

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. 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。

1.2 文法

在本实验中，我们设计并实现了一个简单编程语言的词法分析器（Lexer）和语法分析器（Parser），以探讨编译原理中词法分析和语法分析的基本概念。词法分析器负责将源代码分解成词法单元（tokens），语法分析器则根据文法规则检查这些词法单元的顺序是否符合语言的语法结构，并生成语法树（AST）。实验的目标是实现一个简单语言的基本词法分析和语法分析功能，为后续的编译优化与代码生成打下基础。

1.3 词法分析器（Lexer）

词法分析器的主要任务是将源代码分解成一系列词法单元。在本实验中，词法单元包括标识符、常量、运算符、分隔符等，具体的词法规则如下：

- 标识符（Identifier）：由字母或下划线开头，后面可以跟字母、数字或下划线，表示变量名、函数名等。

$$\text{Identifier: } [\_a-zA-Z][a-zA-Z0-9\_]^*$$

- 整数常量（IntConst）：可以是十进制或十六进制表示的数字。

$$\text{IntConst: } (0x|0X)[0-9a-fA-F]+ \mid [0-9]+$$

- 关键字和符号：如逗号（`,`）、分号（`;`）、赋值符号（`=`）、括号、运算符等。

词法分析器的任务是扫描源代码并将其分解成这些基本单元，生成一系列的词法单元序列供语法分析器进一步处理。

1.4 语法分析器（Parser）

语法分析器的任务是根据给定的文法规则检查词法单元序列是否符合语言的语法结构，并构建语法树（AST）。在本实验中，我们根据词法分析器生成的词法单元，设计了一个符合特定语法规则的语法分析器。以下是语法规则的详细描述。

首先，整个程序的语法结构由多个声明（decl）和函数定义（funcDef）组成，程序以文件结束符（EOF）结尾。该语法规则可以表示为：

$$\text{compUnit}: (\text{decl} \mid \text{funcDef})^{+}\ \text{EOF}$$

其中，decl 表示声明，funcDef 表示函数定义。

1.4.1 声明 decl

声明分为常量声明（constDecl）和变量声明（varDecl）。其语法规则为：

$$\text{decl}: \text{constDecl} \mid \text{varDecl}$$

常量声明的语法规则为：

$$\text{constDecl}: \text{Const bType constDef (Comma constDef)}^{*}\ \text{SemiColon}$$

其中，bType 表示基本类型（如整数类型），constDef 表示常量定义。常量定义的语法规则如下：

$$\text{constDef}: (\text{Identifier} \mid \text{array})\ \text{Assign}\ (\text{exp} \mid (\text{LeftBrace exp(Comma exp)}^{*}\ \text{RightBrace}))$$

常量可以是标识符或数组，且必须有一个初始值（可以是单个值或数组）。
变量声明的语法规则为：

$$\text{varDecl}: \text{bType varDef (Comma varDef)}^{*}\ \text{SemiColon}$$

变量定义的语法规则为：

$$\text{varDef}: (\text{Identifier} \mid \text{array})\ (\text{Assign}\ (\text{exp} \mid (\text{LeftBrace exp(Comma exp)}^{*}\ \text{RightBrace})))?$$

其中，变量可以是标识符或数组，并且可以有初始值。

1.4.2 数组声明（array）

数组声明有两种形式：OBC 数组和普通数组。OBC 数组的语法规则为：

$$\text{obcArray}: \text{Obc unobcArray}$$

普通数组的语法规则为：

unobcArray: Identifier LeftBracket (exp)? RightBracket

其中，`obcArray` 是 `Obc` 类型的数组，`unobcArray` 是普通数组，后者可能包括数组的大小。

1.4.3 语句（stmt）

语句的语法规则如下，涵盖了多种语句类型，包括赋值语句、条件语句、循环语句等：

stmt: block
| lval Assign exp SemiColon
| SemiColon | exp SemiColon
| If LeftParen cond RightParen stmt(Else stmt)?
| While LeftParen cond RightParen stmt

其中，`block` 是语句块，`lval` 是左值，`exp` 是表达式，`cond` 是条件表达式，`If` 和 `While` 分别表示条件语句和循环语句。`Else` 部分是条件语句的可选部分。

1.4.4 表达式（exp）

表达式的语法规则非常重要，它涵盖了变量、常量、运算符等基本构件。表达式的语法规则如下：

exp: lval
| number
| LeftParen exp RightParen
| (Minus | Plus) exp
| exp (Multiply | Divide | Modulo) exp
| exp (Plus | Minus) exp
| exp (Equal | NonEqual | Less | Greater | LessEqual | GreaterEqual) exp

其中，`lval` 表示左值，`number` 表示常量，`Plus`、`Minus`、`Multiply`、`Divide`、`Modulo` 表示运算符，`Equal`、`NonEqual`、`Less`、`Greater` 等表示关系运算符。

1.5 工程修改说明

本次实验并未修改除了 `SafeCLexer.g4`、`SafeCParser.g4`、`AstBuilder.cpp` 之外的任何文件。

1.6 实现过程及困难

1.6.1 SafeCLexer.g4

本文件主要是对 Safe C 语言的词法分析器进行定义，包括了 Safe C 语言的关键字、标识符、常量、运算符等的词法规则。

本文件的编写过程中并没有遇到太大的困难，主要是根据 Safe C 语言的词法规则进行定义。

1.6.2 SafeCParser.g4

本文件的编写过程主要在于理解 Safe C 语言的语法规则，并将其转化为 ANTLR4 的语法规则。（呈现为 token 流的方式）

本文件的完善过程主要是循序渐进的根据测试脚本的执行结果进行逐步的完善。测试脚本会给出当前语法分析器在解析文件时的错误信息（通常为遇到的未定义 token 的位置），根据这些信息，找到对应的规则进行 Debug。

本文件的编写过程中遇到的主要困难是对 Safe C 语言的语法规则的理解，以及如何将其转化为 ANTLR4 的语法规则。在编写过程中，我们发现了一些 Safe C 语言的语法规则与 C 语言的语法规则不同的地方，例如 Safe C 语言中的数组声明需要在变量名后面加上 `obc` 关键字，这在 C 语言中是不需要的。而且在编写 `AstBuilder.cpp` 时，我们发现之前通过测试的 Safe C 语法规则在实现抽象语法树构建时会引入不必要的麻烦与冗余（主要体现在对于赋值语句被错误的划分到表达式逻辑中），故我们在编写 `AstBuilder.cpp` 过程中对语法规则进行了修改。

```
stmt:
    block
    | lval Assign exp SemiColon // 此行为编写 AstBuilder.cpp 时添加
    | SemiColon
    | exp SemiColon
    | If LeftParen cond RightParen stmt (Else stmt)?
    | While LeftParen cond RightParen stmt;
exp:
    lval
    | lval Assign exp // 此行于编写 AstBuilder.cpp 时删去
    | number
    | LeftParen exp RightParen
```

```
    | exp (Plus | Minus | Multiply | Divide | Modulo) exp
    | (Minus | Plus) exp
    | exp (Equal | NonEqual | Less | Greater | LessEqual | GreaterEqual)
    ↪  exp;
```

1.6.3 AstBuilder.cpp

本文件的编写过程主要是根据 ANTLR4 生成的语法分析树，遍历语法分析树，生成抽象语法树。

本文件的编写过程先借助 `Copilot` 对整体代码结构进行了建构，但是这一版代码并不能正确执行功能，其会遇到的主要问题有：

1. 不能进入正确的建构逻辑片段，导致节点类型与内容错误
2. 不能进行正确的类型转换，出现 `bad cast` 错误
3. 不能正确的实现抽象语法树节点的字段填写，出现空字段，从而在建构逻辑正确的情况下，输出抽象语法树时出现 `Segmentation Fault` 错误

首先我们简述我们调试分析过程中插入的终端输出代码：

```
#define DEBUG 1
#define DEBUG_SIGN "[d] "
if (DEBUG)
  printf("%s %s %d %d\n", DEBUG_SIGN, __func__, result->line, result->pos);
```

下面简述我们调试分析建构逻辑错误的过程：

1. 编译运行，观察终端输出，初步了解本次运行的函数调用情况
2. 通过手动跟踪调用过程，发现错误的分支进入，定位错误发生的行号
3. 修改，回到步骤 1

下面简述我们调试分析类型转换错误的过程：

1. 编译运行，观察预定义的终端输出，初步了解本次运行的函数调用情况
2. 使用 gdb 调试运行，使用 `bt` 指令查看函数调用栈，定位错误发生的行号
3. 通过手动跟踪调用过程，观察类型转化流程，发现错误的类型转化
4. 修改，回到步骤 1

```
antlrcpp::Any AstBuilder::visitNumber(SafeCParser::NumberContext *ctx) {
  auto result = new number_node;
  /* 略去其它代码 */
  // return result;
  // 需要进行正确的类型转换（子类到父类的转换）
  return dynamic_cast<expr_node *>(result);
}
```

下面简述我们调试分析空字段错误的过程：

1. 编译运行，观察预定义的终端输出，发现 Segmentation Fault 错误
2. 使用 gdb 调试运行，发现是在写入抽象语法树 Json 时出现错误
3. 此时只能手动排查每一个涉及的抽象语法树节点，查找可能的空字段[1]
4. 修改，回到步骤 1

```
antlrcpp::Any AstBuilder::visitConstDef(SafeCParser::ConstDefContext *ctx){
  /* 前文省略 */
  result->btype = BType::INT;
  // 此字段若不进行赋值，会导致后续在输出抽象语法树时出现Segmentation
  ↪  Fault错误
  /* 后文省略 */
}
```

[1]起初我们并未发现是空字段问题，是在逐个排查抽象语法树节点时偶然发现