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

1.3 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。

1.4 实现过程及困难

1.4.1 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。

1.4.2 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;
```

1.4.3 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