c2python实验报告

穆俊成 张国威 张毅臣

环境配置

- 系统：`Windows`
- 语言：`python3.9`
- 安装`antlr4`包

```
 pip install antlr4-tools
```

- 安装`antlr4`运行时库

```
pip install antlr4-python3-runtime
```

程序运行方法

在`src`文件夹中打开终端，运行

```
python token_printer.py test/xxx.c
```

词法分析实现方法

- 编写`c_src.g4`文件，利用编译原理的知识在其中实现对于c语言源码的词法分析
- 运行命令`antlr4 -Dlanguage=Python3 c_src.g4`
- 利用生成的`c_srcLexer.py`可以实现词法分析：

```
lexer = c_srcLexer(FileStream(input_file))
token_stream = CommonTokenStream(lexer)
# 填充Token流
token_stream.fill()
# 获取Token列表
all_tokens = token_stream.getTokens(start=0,
stop=len(token_stream.tokens) - 1)
# 打印Token信息
for token in all_tokens:
print(f"Token: {token.text}, Type: {TOKEN_NAMES[token.type]}, Line:
{token.line}, Column: {token.column}")
```

- 编写`token`映射表`TOKEN_NAMES`，其内容对应`antlr4`生成的`c_src.tokens`中的`token`的编号
- 编写测试程序进行`token`流输出测试

语法分析实现方法

在实现了词法分析之后即可利用`antlr`库比较容易地实现语法分析功能，具体操作如下：

实现一个类来继承`antlr`生成的类`c_srcListener`，用来生成`json`格式的输出。

```
class JsonTreeListener(c_srcListener):
    def __init__(self):
        self.tree = {}
    def enterProg(self, ctx):
        self.tree["prog"] = self.visitChildren(ctx)
    def enterInclude(self, ctx):
        self.tree["include"] = ctx.getText()
    def visitTerminal(self, node):
        return node.getText()
        def visitChildren(self, node):
        result = []
        for child in node.getChildren():
            child_data = child.accept(self)
            child_name = type(child).__name__
            result.append({child_name: child_data})
        return result
```

接下来从词法分析中得到语法分析树：

```
lexer = c_srcLexer(FileStream(input_file))
token_stream = CommonTokenStream(lexer)
parser = c_srcParser(token_stream)
parser.removeErrorListeners()
tree = parser.prog()
json_listener = JsonTreeListener()
walker = ParseTreeWalker()
walker.walk(json_listener, tree)
parse_tree = json_listener.tree
json_output = json.dumps(parse_tree, indent=2)
print(json_output)
```

python代码生成

按照说明文档中的要求配置好环境后，在`src`文件夹中打开终端，运行命令`python python_code_generator.py test/xxx.c`即可生成`xxx.py`文件，运行文件即可测试功能。

首先通过`antlr4 -Dlanguage=Python3 c_src.g4 -visitor`命令得到`c_srcVisitor.py`文件，此文件可以对语法分析树进行遍历，只需要重载其中的遍历函数即可实现所有节点的访问，并进行代码生成。

具体实现上，我们建立了一个类`PythonCodeGenerator`继承`c_srcVisitor`基类，它重载了基类的成员函数用于遍历语法分析树的每个节点。

由于C语言中的函数`gets`、`atoi`、`printf`、`scanf`、`strlen`在`python`中并没有对应的实现，所以为了实现对应的代码转换，我们封装了一些函数实现在`utils.py`中：

并通过在生成的`python`文件中加入`from utils import *`来引入所有的函数。

具体的代码生成方法上，我们维护了类`PythonCodeGenerator`中的一个成员变量`python_code`，它是一个字符串，每次对代码进行新增时都会调用成员函数`addline`：

```
def add_line(self, line):
    self.python_code += "    " * self.indentation + line + "\n"
```

这样只需要调用此函数传入具体的`python`语句即可实现自适应的缩进调整以及代码追加。

难点和创新点

本次词法分析部分的难点在于`g4`文件的编写以及整个代码结构的构建。

本项目的**创新点**有以下几点：

- 支持结构体、一维数组
- `python`字符串可以单独对某个字符进行修改
- 支持各种`if else`语句以及各种循环语句
- 支持绝大多的`c`基本语法（提供了多个复杂的测试程序来进行验证，包括四则运算测试）

本项目**难点**有以下几点：

- 缩进问题

  生成源语言`c`并不需要严格控制缩进，但是目标语言`python`需要特别注意缩进格式等，针对这个问题我们的解决方法是专门设计一个控制当前缩进值的变量`indentation`，当进入`while`循环、函数体等时手动增加这个变量，而在退出时恢复这个缩进值，又定义了两个成员函数来调整缩进值：

  ```
  def increase_indentation(self):
      self.indentation += 1
  def decrease_indentation(self):
      self.indentation -= 1
  ```

  控制缩进的方式如下（以`while`循环为例）：

  ```
  def visitWhileBlock(self, ctx: c_srcParser.WhileBlockContext):
      self.add_line(f'while {self.visit(ctx.getChild(2))}:')
      self.increase_indentation()
      self.visit(ctx.getChild(5))
      self.decrease_indentation()
  ```

- 对于表达式的文法设计与遍历实现

  这部分的实现的文法如下：

  ```
  expression
      : '(' expression ')'               #parens
      | op='!' expression                   #Neg
      | expression op=('*' | '/' | '%') expression   #MulDiv
      | expression op=('+' | '-') expression   #AddSub
      | expression op=('==' | '!=' | '<' | '<=' | '>' | '>=') expression
  #Judge
      | expression Logical expression             # AND
      | (op='-')? intType             #int
      | (op='-')? doubleType          #double
      | charType                       #char
      | stringType                     #string
      | arrayItem                  #arrayitem
  ```

```
    | structMember                #structmember
    | varName                        #identifier
    | func                        #function
    ;
```

通过将表达式拆分成多种可能，并且分别注明`lable`来生成更加细致的节点遍历函数，这很大程度上简化了需要手动实现的代码，并且使得代码的可读性、可拓展性进一步增强。

- `python`中字符串的实现

  由于`python`中的字符串不能实现单个字符的修改，为此采用了`list`来对字符串进行模拟，具体的，当初始化一个字符串时，会将`list`中的所有元素都赋值为0，并将第一个出现的0作为字符串末尾的标识。这样的思路可以方便`strlen`等函数的实现：

```python
def gets(char_array):
    tmp = input()
    for i in range(len(tmp)):
        char_array[i] = tmp[i]
def strlen(char_array):
    if isinstance(char_array, str):
        return len(char_array)
    else:
        return char_array.index(0)
def atoi(char_array):
    tmp_str = ''
    for i in range(strlen(char_array)):
        tmp_str += str(char_array[i])
    if tmp_str.isdigit():
        return int(tmp_str)
    else:
        return -1
```

并且在`c`源码中对于字符串的一个位置赋值0来表示字符串结尾，这和`python`的实现相吻合，也大大简化了代码实现的难度。

小组分工

张国威：

学习`antlr`使用方法，利用`g4`文件生成中间代码，编写`token`流输出函数。

设计遍历语法分析树的类，搭建框架。编写部分遍历函数。

穆俊成：

学习`antlr`使用方法，编写`g4`文件。

编写语法分析部分的输出函数。编写遍历语法分析树生成`python`代码的函数，

张毅臣：

编写测试程序、`token`映射表、参与`g4`文件编写。

编写测试程序并根据设计的文法调整测试程序实现。编写部分具体函数。