# 编译原理小组作业: 基于 Lex 和 Yacc 的 C to Python 编译器

组员: 吴佳龙 赵伊书杰 黄舒炜

## 开发环境与使用说明

#### 项目说明

操作系统: Ubuntu 16.04 (运行在虚拟机上)

Python版本: Python 3.7

团队开发管理: Github, HedgeDoc

文件目录结构:

```
.

├── bin

├── parser # 可在Linux下运行的编译器

├── README.md # 运行说明

├── utils.py # 运行组件

├── doc

├── report.pdf # 此报告

└── src

├── examples

├── examples

├── model | └── model | ── model
```

#### 编译与运行方法:

```
$ cd src
$ make clean
$ make
$ ./parser [Input C File]
$ python out.py
```

若缺省输入文件,则默认编译同级目录下的 in.c 文件。(可在/src/examples下找到更多测试用例)

## 语法支持情况

本次作业支持C语言的大部分常见语法,具体说来,包括:**常量、变量、算术与逻辑表达式、条件与循环语句、一维数组、函数及其参数的 定义与使用**,包括对其**作用域的检测与判断。**除此之外,还支持部分**常见错误的检测**。

但是,下面的语法特性是不支持的:**指针与引用**(Python中并没有相关概念),**高维数组**,**continue, break和do - while语句,在if/while语句中使用单独的变**量作为判断条件。

#### 错误检测与处理

本次作业除了编译正常的C代码外,还能对部分常见错误进行检测与报错处理,如:**词法和语法错误,变量、数组的重定义或未定义**,**函数的重定义或未定义**。其中,对于重定义和未定义的错误,我们支持在遇见错误的情况下**继续对余下部分编译**,并在编译完成后统一给出提示信息,使得用户能够在一次编译过程中发现尽可能多的错误。

## 技术方案

本次作业分别使用Lex和Yacc作为词法和语法分析器,并通过对Yacc返回的抽象语法树(AST)进行遍历和解析,维护了符号表、完成了目标代码的生成和错误处理。详见下文。

## 亮点与难点

## 库函数的翻译: scanf 和 printf 等

由于 C 中的库函数在 Python 没有对应,因此我们用 Python 语言额外实现了示例程序用到的 C 函数,包括 scanf, printf, atoi, strlen, 放在 utils.py 中,并通过在生成的目标 Python 程序中 from utils import \* 来使用。以下说明两个难点函数的 Python 实现。

#### scanf 的 Python 实现

首先将 scanf 的格式化字符串中的限定符 %d 和 %s 转为相应的正则表达式([-+]?\d+)和(\s+),并通过正则匹配从输入中读取相应的值。此外,由于解释型语言 Python 的输入方法 input() 是逐行读入的,我们还实现了输入缓冲区,当正则匹配不能匹配到所有想要的值时,就新读取一行加入缓冲区,重新匹配。

**调用方法**:目前支持格式化字符串中仅含限定符 %d 和 %s 并且**必须用空格分割**。对于 %d , 返回数值类型; 对于 %s , 返回的是长度为 1 的 str (视为一个字符)的 list。

```
< scanf("%s%d", a, &b);
---
> [a, b] = scanf("%s %d")
```

## printf 的 Python 实现

使用 Python 中的格式化字符串实现。调用方式与 C 中稍有不同:

```
< printf("a=%s, b=%d, c=%d, d=%d", a, b, c, d);  // C
---
> printf("a=%s, b=%d, c=%d, d=%d", (a, b, c, d))  // utils.py
```

## 语义分析

## 变量、函数的重定义(未定义)检查

本次作业中,通过对符号表的维护,我们支持识别每个变量的定义情况及其所在的作用域。当用户试图引用一个变量时,程序将从当前所在作用域开始,从栈顶向栈底逐层移动,直到找到某个同名变量,就返回其信息。或者,如果程序在移动到栈底(对应于全局变量)时仍无法找到该变量,就记录一个变量未定义错误,并继续编译流程。

检查重定义的思路与检查未定义是相似的,只不过此时程序只需要检查当前所在的作用域(而不必在栈中向下移动)是否已有同名变量,并 对同名变量报错即可。

由于C语言不支持在函数体内另外定义新的函数,因此我们没有必要对函数维护作用域栈。只需要维护一个全局的函数符号表进行查询和删改即可。

## 变量的作用域检查及其重命名

正如上一节中所描述的那样,除了符号表的管理,程序还在编译时动态维护了一个作用域栈。栈顶指向了编译器当前所在的作用域,从栈顶向下移动,作用域逐渐扩大,直到达到全局作用域。对于变量的引用,只需要从栈中检查是否已有同名变量即可判断引用是否合法。而对于变量的定义,可能会遇见以下三种情况:

- 在整个栈中都没能找到该变量的定义
  - 此时,变量是第一次定义,只需要将其加入当前所在的作用域即可。
- 在当前作用域栈帧找到了该变量的定义
  - 此时,变量重定义,产生合适的报错信息即可。
- 在当前作用域栈帧没能找到该变量的定义,但在更早的栈帧中找到了同名变量的定义。

此时,变量没有重定义。但由于Python的语法特性,我们需要将更早栈帧中定义的变量与当前栈帧中的变量名区分开,以避免误操作外部变量。对这一问题的解决方案是将当前变量重命名,并同时保存该变量的原名称和新名称。每当引用一个变量时,我们就通过比较原名称的方式在栈中查找该变量。但是,一旦找到所需的变量,我们会返回其新名称,以避免对外部变量的误操作。

具体示例请见下节的"变量重命名示例"部分。

## 测试程序

#### 回文检测

```
$ ./parser examples/palindrome.c
```

输入字符串,输出 True 或者 False

测试样例:

```
# Sample 1
In[1]:
    palindrome
Out[1]:
    False
# Sample 2
In[2]:
    abcba
Out[2]:
    True
```

## 四则运算计算

```
$ ./parser examples/calc.c
```

#### 支持的四则运算表达式:

支持的四则运算表达式可由文法G[Expr]描述(不支持包含空格的表达式):

$$Expr 
ightarrow Expr \ Op \ Expr \ Expr 
ightarrow Sign \ (Expr) \ Expr 
ightarrow number \ Op 
ightarrow + |-| imes | \div \ Sign 
ightarrow + |-| arepsilon$$

测试样例:

```
# Sample 1
```

```
In[1]:
    -1*-(2+3)
Out[1]:
    5.000000
# Sample 2
In[2]:
    4/(3*2)
Out[2]:
    0.666667
# Sample 3
    -(+(-(+(-3))))/2+5*(-2)
Out[3]:
    -11.500000
```

## 语法检查示例

```
$ ./parser examples/special.c
```

#### 语法错误说明:

- 第9行重复定义函数func1 (第3行已定义)
- 第16行重复定义变量a (第14行已定义)
- 第17行使用未定义的变量b
- 第19行使用未定义的变量b
- 第19行使用未定义的函数func2
- 第22行重复定义数组变量arr

```
运行结果:
        You, 6 minutes ago | 1 author (You)
                                                     Generating code for 11NAssignment
        int c=0;
                                                     Creating assignment for b
                                                     Creating method call: func2, 1 arguments
       int func1(int a){
                                                     Generating code for 17NArrayDeclaration
            c=c+1;
                                                    Creating array statement: arr
Creating integer: 10
Generating code for 20NExpressionStatement
            a=c;
                                                     Generating code for 11NAssignment
                                                     Creating assignment for arr
                                                     Creating array variable
       int funcl(int a){
                                                     Creating integer: 1
            return 0;
                                                     Generating code for 20NVariableDeclaration
                                                     Leaving block
                                                    Current indents: 0
        int main(){
                                                    Leaving block
            int a=0;
                                                    Current indents: -1
            a=a+1;
                                                    error occurred when compiling, stoppoing...
                                                    Line 9: Redefinition of function "func1"
Line 16: Redefinition of identifier "a"
Line 17: Identifier "b" is undefined
Line 19: Identifier "b" is undefined
Line 19: Function "func2" is undefined
             int a=2;
            b=0;
            a=func1(a);
            b=func2(a);
             int arr[10];
                                                    Line 22: Redefinition of identifier "arr"
            arr[1]=a;
                                                    corn@corn-virtual-machine:~/桌面/simple-compiler$
             int arr;
```

## 变量重命名示例

```
$ ./parser examples/test.c
```

重命名说明(括号外的行号为原c代码,括号中的行号为翻译后的python代码):

- 程序多处使用变量名为a的变量,对不同的变量进行重命名
- 第1行 (第3行) 为全局变量,使用原名a,改变量在第8行 (第10行) 中使用
- 第3行(第4行)函数参数重命名为**a\_1**,该变量在第4行(第6行)中使用
- 第9行 (第11行) 在main中定义变量a,该变量为局部变量,重命名为**a\_2**,在第10、16、17行 (12、17、18行) 中使用
- 第11行(第11行)定义变量,使用原命名a\_4,在第11行(第14、16行)中使用
- 第12行 (第15行) 定义变量并使用, 重命名为a\_5

运行结果(左边为原c代码,右边为翻译后的python代码):

```
C test.c X G codegen.cpp
                                                         🔹 🕚 🗓 ... 🕒 codegen.cpp
examples > C test.c > 😭 main()
                                                                             out.py
                                                                                   You, seconds ago | 1 author (You)
     int dothis(int a){
                                                                                   def dothis(a_1):
                                                                                       global a
     int main(){
                                                                                       a_1 = (a_1 + 1)
         a=0;
         int a=1;
                                                                                   def main():
         if (a==0){
             for (int a=0;a<1;a=a+1){
                                                                                       a_2 = 1
                int a=1;
                                                                                       if (a_2 == 0):
                                                                                           a_4 = 0
                                                                                           while (a_4 < 1):
         if (a==0)[
                                                                                              a_5 = 1
            int x = a;
                                                                                               a_4 = (a_4 + 1)
                                                                                       if (a_2 == 0):
                                                                                           x = a_2
                                                                                  if __name__ == '__main__':
                                                                                       main()
```

# 分工

#### 吴佳龙:

- Lex 和 Yacc 的技术调研
- 实现词法分析和语法分析,支持**除数组外**大部分基本语法
- 支持编译错误信息显示行号
- 示例程序(回文检测和四则运算)的编写和正确性验证

#### 赵伊书杰:

- 一维数组的实现
- 部分语义识别与错误检测
- 部分程序测试、语法部分Bug修复与完善
- 文档撰写

#### 黄舒炜:

- 语法检查,对变量/函数的未定义使用/重定义进行检查
- 变量重命名,对同名不同作用域的变量进行重命名
- 错误处理示例程序编写及验证
- 示例程序 (四则运算) 的测试

## 参考资料

- Write text parsers with yacc and lex IBM Developer
- Writing Your Own Toy Compiler Using Flex, Bison and LLVM (gnuu.org)
- <u>re Regular expression operations Python 3.9.1 documentation</u>