编译小组作业 - C到LLVM编译器

李奕杉 闵安娜 李乐程

一、开发环境

编译器整体使用Python 3开发。

二、实现原理

自制词法分析器

- 词法分析器使用Python 3编写，通过读取源代码文件，根据读取到的字符进行词法分析。
- 词法分析器会跳过注释与空白字符，根据读取到的首字符进行判断，然后使用正则表达式匹配后续字符，匹配成功后继续跳过注释与空白，根据首字符判断，使用正则表达式匹配后续字符，直到读取到文件末尾。
- 将调用的系统函数处理为关键字，实现了对系统函数的识别。
- 测试代码使用的是KMP字符串匹配与四则运算计算，并增加了一个排序程序（包括快排、归并、插入排序、基数排序）。四则运算代码参考了数据结构这门课程的课件。

自制语法分析器

- 语法分析器使用Python 3编写，实现了对.g4文件的读取，将g4文件中的语法规则转换成了`dict`类型。支持*,+,?,|,()等运算符。
- 实现了LR(1)的闭包计算，项目集规范族计算，First集和Follow集计算，Action表和Goto表的构建。
- 实现了对LR冲突,RR冲突进行报错。
- 可以对计算出的状态、Action表和Goto表进行输出，输出为JSON格式，并进行读取。
- 语法分析器将输出的AST转换成了`dict`类型，输出为JSON格式。
- 实现了自制的Visitor，可以遍历AST，生成中间代码。

编译器

- 使用Python 3编写，调用了Antlr4.7.2工具，实现了对C语言的词法分析和语法分析，并进行语义分析。使用Llvmlite进行代码生成。
- 使用符号表保存变量的类型，会根据语义进入、退出作用域。在进入函数、循环、选择时会进入作用域。
- 使用结构表保存结构体的类型，类似数组，保存结构各个分量的地址、类型、名称。
- 支持`int`,`char`,`float`,`double`,`void`等基本类型。初步的类型转换（int转char，int转float，float转int等）。
- 支持`if`,`else`等条件语句。将程序分块，每个块中的变量都是局部变量，不同块中的变量不会冲突。
- 支持`while`,`for`等循环语句。支持`break`和`continue`，指向对应块的入口。同时会保证各层循环的变量不会冲突。
- 通过声明，可以调用库函数和自定义函数。允许变长参数。
- 支持指针与取地址运算符`&`。
- 结构体支持使用`.`与`->`运算符访问成员。

三、难点

- 在进行符号的匹配时，可能会将连续的符号匹配成一个Token，无法识别，例如会将'()'匹配成一个Token，无法识别为左右括号；对于这种情况，如果识别到的符号不在符号列表中，我们会回溯，将多读取的字符重新读取，然后再进行匹配。
- 自行实现一个`LR(1)`的语法分析器，成功调通，但是由于Python运行速度较慢，项目集规范族约有600多条，每次修改文法之后都要重新计算项目集规范族、Action表等，即便使用Pypy3运行，也需要花费较长时间。自制词法分析器、语法分析器在./src/diy文件夹中。
- 隐式类型转换要考虑的情况很多，例如表达式计算，函数调用等；既要保证方便也要保证安全。
- 浮点数的运算与整型不同，需要对应的指令，例如`fadd`、`fsub`等。
- 符号表的管理，需要在进入、退出作用域时进行相应的操作，同时需要保证各层作用域的变量不会冲突。
- 声明函数时，需要将函数的参数与局部变量分开，同时需要保证各层作用域的变量不会冲突。
- 声明结构体时，需要将结构体的成员与局部变量分开，同时需要保证各层作用域的变量不会冲突。
- 使用条件语句或循环语句时，如果有`break`或`continue`，需要将其指向对应块的入口。如果有`return`，需要将其指向函数的出口。

四、创新点

- 自己实现了词法分析器与语法分析器。
- 在语义分析时，使用结构表对自定义的数据结构进行管理，实现了对数据结构的定义与使用。

五、小组分工

李奕杉 - 自制词法分析器、测试代码编写，自制语法分析器。目标代码生成
李乐程 - 自制语法分析器，文法编写，目标代码生成。
闵安娜 - Antlr语义分析，目标代码生成的大部分实现。

六、可以进行的改进

- 目前对宏的支持不够完善，只能识别，但没有进行宏替换。
- 继续完善自己研发的语法分析器，使用C进行重写，提高运行速度。
- 不支持显式类型转换，对于隐式类型转换不够完善，例如char和int同时出现在表达式中会错误，有时转换结果不正确。
- 对于浮点数的支持不够完善。
- 可以继续完善目标代码生成，实现支持更多的语法。

七、程序使用说明

自制词法分析器

运行lex.py实现词法分析。

指令示例：

```
python lex.py < ./test/calculator.cpp
```

输出示例：

```
Token<Const, const, 211, 216>
Token<Int, int, 217, 220>
Token<ID, N_OPTR, 221, 227>
Token<ASSIGN, =, 228, 229>
Token<DecimalLiteral, 7, 230, 231>
```

Token分为四个部分：类型、值、起始位置、结束位置。
如下是一个Token的示例，其中类型是Float，值是float，起始位置是第1171，结束位置是第1171。

```
<Float, float, 1171, 1176>
```

用于测试的程序位于test文件夹中，包括了四则运算、KMP字符串匹配，使用方法符合文档要求。

自制语法分析器

指令示例：

```
python LRParser.py < ./test/calculator.c > ./calculator.json
```

输出示例：


```
{
 "name": "DeclSpecifierContext",
 "children": [
  {
   "name": "TypeSpecifierContext",
   "children": [
    {
     "name": "TrailingTypeSpecifierContext",
     "children": [
      {
       "name": "SimpleTypeSpecifierContext",
       "children": [
        {
         "name": "TerminalNodeImpl",
         "symbol": "char"
        }
       ]
      }
     ]
    }
   ]
  }
 ]
}
```


编译器

需要安装llvmlite库与Antlr4库,Antlr版本为4.7.2。教程见https://zhuanlan.zhihu.com/p/423928097。
指令示例：

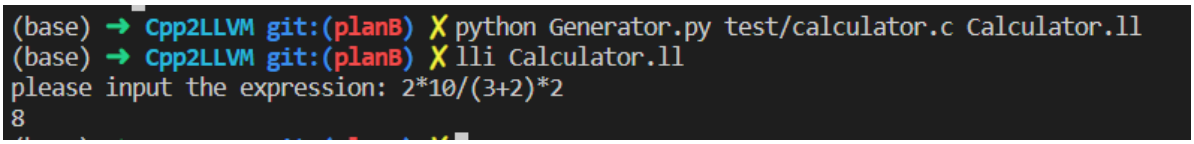
```
python Generate.py ./test/calculator.c  ./calculator.ll
```

第一个参数为输入文件，第二个参数为输出文件。
输出示例：

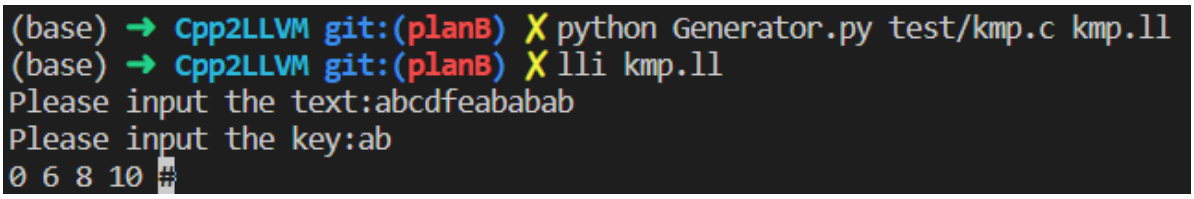
```
; ModuleID = "main"
target triple = "x86_64-pc-linux-gnu"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"

declare i32 @"printf"(i8* %".1", ...)

declare i32 @"gets"(i8* %".1", ...)

declare i32 @"strlen"(i8* %".1", ...)

define i32 @"main"()
{
mainentry:
  %"text" = alloca [1024 x i8]
  %".2" = bitcast [1024 x i8]* %"text" to i8*
  %".3" = alloca i8*
  ......
```

需要自行使用lli进行.ll文件的运行。
运行结果如下：

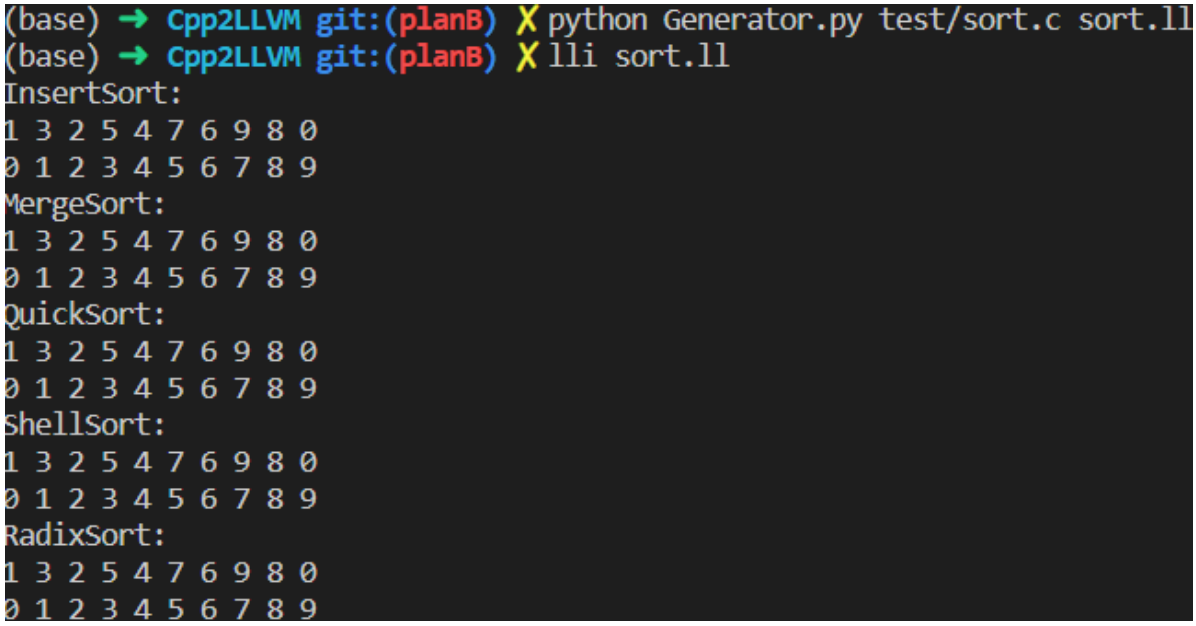
计算器



KMP字符串匹配



排序



八、参考资料

- https://github.com/antlr/grammars-v4/tree/master/cpp
- https://github.com/antlr/grammars-v4/tree/master/c
- https://github.com/dabeaz/ply
- https://zhuanlan.zhihu.com/p/423928097
- http://www.lysator.liu.se/c/ANSI-C-grammar-y.html
- http://llvm.org/docs/
- https://llvmlite.readthedocs.io/en/latest/