C-like Programming Language Interpreter

by DY, HL from HUST 有点菜 队伍

一 编译与运行

编译环境

OS	Compiler	Compiler Options	Compile Dependencies
Linux >= 4.7.0	g++ >= 6.2.1	-std=c++11 -Wall	cmake >= 2.8

为获得最佳用户体验，请使用Arch Linux (kernel: 4.8.4-1), g++6.2.1 编译并测试

编译步骤

1. 在工程目录下，运行 `cmake .`
2. 在工程目录下，运行 `make`

运行方法

直接运行可执行文件（SeedCup2016.exe），可执行文件将会读取当前目录下的input.txt，并根据比赛题目要求，输出运行结果至output.txt

二 基本特点

1. 采用C++11编写，编码风格基本符合Google Style Guider。
2. 结构清晰，代码耦合度低，可扩展性强，其中：正则引擎，语法分析器，抽象语法树等，皆为通用组件。
3. 源代码注释详尽，以头文件注释为主，源文件注释作为头文件的补充说明。
4. 注释采用doxygen格式书写，支持doxygen生成。
5. 实现效率高，核心算法都是线性级别。
6. 实现手法朴素易懂，

三 实现功能

（附录六以及 `doc/example.c` 中有的代码示例）

基本功能

比赛要求的基本功能和三种结构全部实现：

1. 能够处理识别并过滤注释
2. 能够正确识别并运行：顺序结构的5种语句类型(声明，定义，初始化，赋值，空语句）
3. 能够正确处理分支结构，并选择正确的分支执行
4. 能够正确处理循环结构，并根据循环条件执行循环体或者跳转
5. 能够正确处理一行内的多条语句
6. 能够正确处理变量作用域

拓展功能

（打星号的为比较叼的特性）

1. 支持单独出现的花括号作用域
2. *支持printf函数：能够像系统函数一样正确的输出结果，并且返回值。
3. 支持任意单列的表达式，如：a++; c + b;
4. 支持 if 的条件表达式中出现任意的表达式：e.g. if (a, c) printf("hello");
5. 支持十六进制，八进制。
6. *支持表达式中出现括号

在不超出测试环境的内存和CPU的限制之下：

1. 支持任意数量的变量
2. 支持任意数量的语句
3. 支持单行任意长度的语句
4. 支持任意数量的分支结构和循环结构相互嵌套
5. *支持8级运算符任意混合

```
1. ++ --
2. + - （正/负）
3. * /
4. + - （加/减）
5. < <= >= >
6. == !=
7. =    （e.g. a = b = c =d）
8. ,（逗号运算符）    （e.g. c = 1, 2, 3）
```

四 项目架构

1. 源码目录结构

```
Src
├── clean.sh
├── CMakeLists.txt
├── common
│   ├── mem_manager.h
│   ├── simplelogger.h
│   └── utility.h
├── input.txt
└── src
    ├── ast.cc
    ├── ast.h
    ├── clike_grammar.cc
    ├── clike_grammar.h
    ├── clike_interpreter.cc
    ├── clike_interpreter.h
    ├── clike_parser.cc
    ├── clike_parser.h
    ├── finite_automaton.cc
    ├── finite_automaton.h
    ├── main.cc
    ├── regex_parser.cc
    ├── regex_parser.h
    ├── symbol.h
    ├── token.h
    ├── tokenizer.cc
    ├── tokenizer.h
    ├── variable_table.cc
    └── variable_table.h
```

简单说明：

- `clean.sh` 清理cmake的生成文件和编译产物的脚本
- `common/` 包含了一些项目无关的通用组件
- `input.txt` 简单的测试数据
- `src/` 主要的实现代码

2.文档目录结构

```
Doc
├── README.pdf
└── resource
    ├── bnf-1.png
    ├── bnf-2.png
    ├── example.c
    ├── grammar_spec.xlsx
    ├── interpreter.png
    └── README.md
```

简单说明：

- `README.pdf` 程序文档
- `resource/` 构建README.pdf的必要资料

3. 程序结构

功能组件

- `include/mem_manager.h`
  通用的小块内存管理器组件，实现了不同抽象级别的内存管理器

- `include/simplelogger.h`
  一个简单小巧的日志库，实现了多级日志输出，日志过滤，包含自动输出当前函数和行号的宏

- `include/utility.h`
  作用域范围内自动释放资源的宏

正则表达式引擎

- `src/finite_automaton.h` `src/finite_automaton.cc`
  实现了确定有限状态机和非确定有限状态机。实现了构造NFA，NFA转DFA，以及DFA的最小化的功能。实现了基于NFA和DFA的字符串查找。并针对分词器做了特定的优化，能够区分词素的优先级。

- `src/regex_parser.h` `src/regex_parser.cc`
  使用了递归下降的手法实现了一个正则语法分析器。解析一串正则表达式，并生成的对应的DFA。

通用语法要素

- `src/symbol.h`
  定义了通用的语法符号类 `class Symbol` ，分为终止符和非终止符。 Symbol内含一个ID和字符串，并且支持比较型容器（std::set, std::map），和Hash型容器（std::unordered_set, std::unordered_map，etc），支持C++形式的标准输出（std::cout）。并且预定了一些常用的语法符号

- `src/ast.h` `src/ast.cc`
  定义了抽象语法树与抽象语法树节点，作为词法分析器的输出，以及解释器的输入。语法节点分为两类：含非终止符的节点/含终止符的节点。抽象语法树的树状结构本身代表了的逻辑表示，其具体的含义由具体的词法分析器和具体的解释器决定。

通用的词法分析器

- `src/token.h`
  定义了Token类，一个Token代表了源代码中存在的某个词素。用Symbol来区分Token类型，记录了Token所在的行号和列号。

- `src/tokenizer.h` `src/tokenizer.cc`
  基于上述的正则表达式引擎，实现了一个通用的词法分析器。给定一串词素对应的正则表达式，并安排合理的优先级，能够自动生成一个词法分析器，支持单行多行注释，支持记录Token在源文件中的位置。
  其中TokenizerBuilder是构建Tokenizer的帮助类。

重要接口:

```
class TokenizerBuilder;
// 设置忽略的Symbol集合
TokenizerBuilder &SetIgnoreSet(std::unordered_set<Symbol> ignore_set);
//设置行注释
TokenizerBuilder &SetLineComment(const string &line_comment_start);
// 设置块注释
TokenizerBuilder &SetBlockComment(const string &, const string &);

class Tokenizer;
// 对输入字符串进行词法分析，返回值表示是否分析成功，tokens参数用来保存分词结果。
bool LexicalAnalyze(const std::string &s, std::vector<Token> &tokens);
bool LexicalAnalyze(const char *beg, const char *end, std::vector<Token> &tokens);
```

C-like Language 语法要素 和 词法分析器

- `src/clike_grammar.h` `src/clike_grammar.cc`
  定义了C-like Language 的所有基本语法要素。基于上述的通用词法分析器，实例化了一个针对C-like Language 的词法分析器。

重要接口:

```
Tokenizer BuilderClikeTokenizer(); // 返回一个C-like Language的词法分析器
```

C-like Language 语法分析器

- `src/clike_parser.h` `src/clike_parser.cc`
  使用递归下降的手法，实现了C-like Language的语法分析，若输入源文件满足C-like Language的语法，则会生成一棵抽象语法树。

重要接口：

```
Ast Parse(std::vector<Token> &tokens); 针对tokens进行语法分析，返回一棵抽象语法树。
```

C-like Language 解释器：

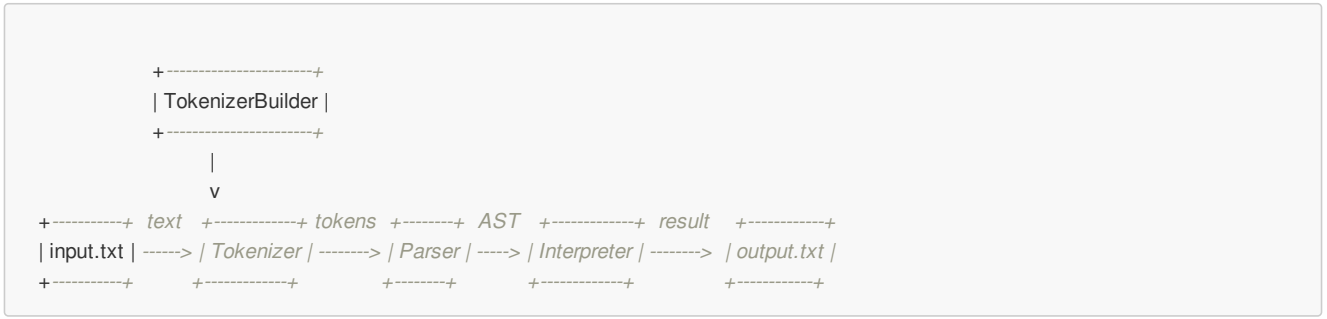
- `variable_table.h` `variable_table.cc`
  定义了变量表，为解释器实现了分级的变量作用域。
- `src/interpreter.h` `src/interpreter.cc`
  遍历生成的抽象语法树，根据语义执行相应计算和控制。

重要接口：

```
class Interpreter;
Interpreter(Ast &&ast); //使用生成的AST初始化解释器
void Exec();  //开始执行
void OutputLines(std::string filename); //输出结果
```

五 程序逻辑

主体流程



1. 先从文件中读入源代码 text
2. 调用 `tokenizer = BuildClikeTokenizer()` 构造词法分析器
3. 调用 `tokens = tokenizer.LexicalAnalyze(text)` 针对源代码进行词法分析，并返回一串Token。
4. 创建 `ClikeParser parser` ，调用 `ast = parser.Parse(tokens)` 对所有的Token进行词法分析。得到抽象语法树(ast)。
5. 创建 `ClikeInterpreter interpreter` ，调用 `interpreter.Exec()` 将对ast解释执行，并记录行号信息
6. 调用 `interpreter.OutputLines()` 输出结果

核心算法

- 基于正则表达式的通用词法分析

  正则表达式引擎：先通过一个简单的正则解析，将正则模式串转换为NFA或者NFA部件，引擎还提供NFA转DFA，DFA最小化的功能。并且针对分词需求做了特定优化，通过指定子表达式的优先级，能够正确地解决分词中的冲突（e.g. if 被识别为关键字而不是变量名）。
  词法分析器：基于正则引擎实现，通过传入 { 正则模式串 -> Token类型 } 来识别各种Token, 能做到分词即识别Token类型，无需再额外查表。并且支持行块注释，能够准确的记录行号列号，方便调试和输出。

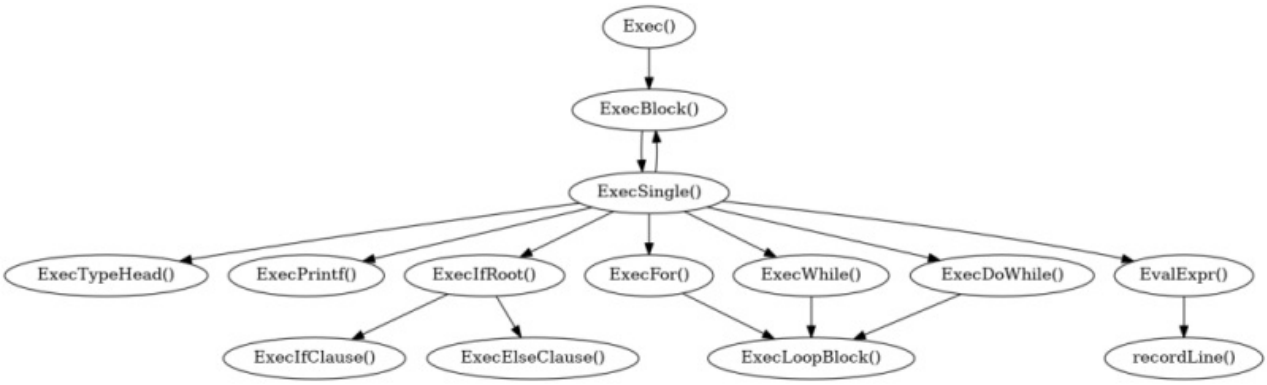
- 基于递归下降的语法分析：

  语法分析部分，基于附录七定义的BNF范式实现。附录所定义的BNF范式，经过我们的严格推导，基本符合比赛方的要求。而且代码实现（见clike_parser.[h/cc]）采用递归下降的手法，每一条BNF语法基本对应代码中的一个函数实现。函数责任划分明确，逻辑清晰，可读性强，配合BNF阅读食用更佳。

- 基于AST的代码解释执行

  解释器：传入一棵抽象语法树，使用深度优先算法，遍历整棵抽象语法树，对树上的各个节点进行解释执行。每一种节点基本对应代码中的一个函数实现，函数内部的算法逻辑基本借用C语言本身的逻辑实现，达到减少bug和代码量，增强可读性的目的。对于无法借用的printf函数调用，则手动实现其逻辑，达到等效的效果。

  变量表：使用std::unordered_map实现一个作用域内变量的快速添加和查找，利用std::vector实现了变量作用域的分级。



六 附录：支持的代码示例

(见 doc/example.c)

```
// 支持printf 返回值
int hello = printf("hello %d\n", 1024);

// 支持十六进制和八进制
int world = -0x10;

// 支持悬空的{}块
{
  printf("hello + world: %d\n", hello + world);
}

// 支持任意的表达式嵌套
int b, c;
int a = b = c = 1;
int comlex_expr = (hello = world = 1 == 10 < -a++ * b - c, 999);

// 支持括号 和 括号嵌套
int parentheses = ((3 + b) * (c > 9) , 999);

// 支持 if, while, do-while, for 的条件表达式中出现任意表达式
if (1, 2, 3) {
  for (int i = 0; printf("dead loop\n"); i++) {
    do {
      // 支持任意的结构嵌套
      c = 5;
      while (a = b = c) {
        // 支持非赋值形式
        c--, b = 1;
      }
    } while (a == b ++ + -c, 0);
    break;
  }
}
```

七 附录：Clike-Language的BNF范式

- [sth] 表示sth出现0次或1次
- { sth } 表示sth出现任意次数

BNF Part 1

左边	右边	预测符号
Start	BlockBody	<stmt_head>
BraceBlock	"{" BlockBody "}"	"{"
BlockBody	{ SingleStmt }	<stmt_head>
SingleStmt	TypeHeadStmt	int
	";"	";"
	break ";"	break
	IfStmt	if
	ForStmt	for
	WhileStmt	while
	DoStmt	do
	BraceBlock	"{"
	ExprStmt	<expr_head>
TypeHeadStmt	int DeclDef { "," DeclDef } ";"	int
DeclDef	id	id
	id = Expr	id
IfStmt	if "(" Expr ")" SingleStmt [else SingleStmt]	if
ForStmt	for "(" ForInit ForCond ")" SingleStmt	for
ForInit	TypeHeadStmt | ExprStmt | EmptyStmt	int, printf, ";", <expr_head>
ForCond	ExprStmt | EmptyStmt	printf, ";", <expr_head>
ForStep	ExprStmt | epsilon	printf, <expr_head>
WhileStmt	while "(" Expr ")" SingleStmt	while
DoWhileStmt	do BraceBlock while "(" Expr ")" ";"	do
ExprStmt	Expr ";"	<expr_head>

BNF Part 2

Expr	CommaExpr	<expr_head>
CommaExpr	AssignExpr { "," AssignExpr }	<expr_head>
AssignExpr	EquationExpr { "=" EquationExpr }	<expr_head>
EquationExpr	CompareExpr { ("==" | "!=") CompareExpr }	<expr_head>
CompareExpr	AddSubExpr { CompareOp AddSubExpr }	<expr_head>
CompareOp	"<" | ">" | "<=" | ">="	"<", ">", "<=", ">="
AddSubExpr	MulDivExpr { ("+" | "-") MulDivExpr }	<expr_head>
MulDivExpr	PosNegExpr { ("*" | "/") PosNegExpr }	<expr_head>
PosNegExpr	["+"] PostfixExpr	<expr_head>
	["-"] PostfixExpr	<expr_head>
PostfixExpr	PrimaryExpr ["++"]	id, number, "(", printf
	PrimaryExpr ["--"]	id, number, "(", printf
PrimaryExpr	id	id
	number	number
	"(" Expr ")"	")"
	PrintfStmt	printf
PrintfExpr	printf "(" string { "," Expr } ")"	printf
备注		
<expr_head> = "+", "-", "(", id, number, printf		
<stmt_head> = "{", if, for, while, do, int, <expr_head>		