使用LLVM实现一门语言（一）Lexer

这个系列来自LLVM的Kaleidoscope教程，增加了我对代码的注释以及一些理解，修改了部分代码。

现在开始我们要使用LLVM实现一个编译器，完成对如下代码的编译运行

```
# 斐波那契数列函数定义
def fib(x)
    if x < 3 then
        1
    else
        fib(x - 1) + fib(x - 2)

fib(40)

# 函数声明
extern sin(arg)
extern cos(arg)
extern atan2(arg1 arg2)

# 声明后的函数可调用
atan2(sin(.4), cos(42))
```

这个语言称为Kaleidoscope，从代码可以看出，Kaleidoscope支持函数、条件分支、数值计算等语言特性。为了方便，Kaleidoscope唯一支持的数据类型为float64，所以示例中的所有数值都是float64.

编译的第一个步骤称为Lex，词法分析，其功能是将文本输入转为多个tokens，比如对于如下代码

```
atan2(sin(.4), cos(42))
```

就应该转为

```
tokens = ["atan2", "(", "sin", "(", .4, ")", ",", "cos", "(", 42, ")", ")"]
```

接下来我们使用C++来写这个Lexer，由于这是教程代码，所以并没有使用优秀的代码设计

```
// 如果不是以下5种情况，Lexer返回[0-255]的ASCII值，否则返回以下枚举值
enum Token {
  TOKEN_EOF = -1,          // 文件结束标识符
  TOKEN_DEF = -2,          // 关键字def
  TOKEN_EXTERN = -3,       // 关键字extern
  TOKEN_IDENTIFIER = -4,   // 名字
  TOKEN_NUMBER = -5        // 数值
};

std::string g_identifier_str;  // Filled in if TOKEN_IDENTIFIER
double g_number_val;           // Filled in if TOKEN_NUMBER

// 从标准输入解析一个Token并返回
int GetToken() {
  static int last_char = ' ';
  // 忽略空白字符
  while (isspace(last_char)) {
    last_char = getchar();
  }
  // 识别字符串
  if (isalpha(last_char)) {
```

```
    g_identifier_str = last_char;
    while (isalnum((last_char = getchar()))) {
      g_identifier_str += last_char;
    }
    if (g_identifier_str == "def") {
      return TOKEN_DEF;
    } else if (g_identifier_str == "extern") {
      return TOKEN_EXTERN;
    } else {
      return TOKEN_IDENTIFIER;
    }
  }
  // 识别数值
  if (isdigit(last_char) || last_char == '.') {
    std::string num_str;
    do {
      num_str += last_char;
      last_char = getchar();
    } while (isdigit(last_char) || last_char == '.');
    g_number_val = strtod(num_str.c_str(), nullptr);
    return TOKEN_NUMBER;
  }
  // 忽略注释
  if (last_char == '#') {
    do {
      last_char = getchar();
    } while (last_char != EOF && last_char != '\n' && last_char != '\r');
    if (last_char != EOF) {
      return GetToken();
    }
  }
  // 识别文件结束
  if (last_char == EOF) {
    return TOKEN_EOF;
  }
  // 直接返回ASCII
  int this_char = last_char;
  last_char = getchar();
  return this_char;
}
```

使用Lexer对之前的代码处理结果为（使用空格分隔tokens）

```
def fib ( x ) if x < 3 then 1 else fib ( x - 1 ) + fib ( x - 2 ) fib ( 40 ) extern sin ( arg )
extern cos ( arg ) extern atan2 ( arg1 arg2 ) atan2 ( sin ( 0.4 ) , cos ( 42 ) )
```

Lexer的输入是代码文本，输出是有序的一个个Token，

在下一节会介绍Parser如何处理有序的Tokens.