This file is the document for project 2

1. Understanding the lex process of the Kaleidoscope
gettok 函数的解释：
其分为以下几种情况：
若当前读到的字符为空格，则直接跳过
若读到的第一个字符是字母，则认为其是标识符，并在读完标识符后检查其是否是保留字，若是则按照保留字返回
若读到的第一个字符是数字或者'.'，则按照数字来处理，直到读到的不是数字或者'.'，但其有一个小 bug，即无法检查出多个小数点(已部分修复以支持多个小数点)
若读到的第一个字符是'#'，则跳过改行的所有字符并递归调用自己寻找下一个 token
若读到的第一个字符是 EOF，则返回 EOF 对应的 token
若读到的第一个字符都不是以上的情况，则直接返回其 ASCII 码值

八进制和 16 进制整数以及多行注释的实现：
八进制和 16 进制整数
若读到的第一个字符是'0'，则读取其后连续的字符[0-7]，若读到的长度大于 0 且该数字不是 0，则返回一个 8 进制数(已经转化为 10 进制的)，否则返回 0
若读到的前两个字符是"0x"或"0X"，则读取其后连续的字符[0-9a-fA-F](直接利用 isxdigit 函数)，若读取的长度大于 0 则返回其转化为 10 进制的数值，否
则将 0x 均回退，'0'按照数字 0 处理，'x'或'X'按照标识符处理
但要注意其优先级比普通数字要高，代码必须在其前面
多行注释的支持(不支持'\'换行符)：
进入条件显然是前两个字符是"/*"，若只遇到'/'则将其回退，否则一直吃掉遇到的字符，直至遇到 EOF 或者"*/"，若为 EOF 返回之，否则递归调用自己寻找
下一个 token

2. 学习使用 flex

一般一个简单的 lex 文件的格式是这样的：

```
definitions
%%
rules
%%
user code
```

第一部分是一些定义，比如字符定义为[a-z]之类的，其中*表示一个或多个匹配， .代表 0 个或多个匹配
还有可以将一些头文件的包含写在%{……%}之中，但要注意%{前面不能有空格
第二部分是一些匹配的规则，以及相应的操作
第三部分是使用者的代码，其会被直接复制到生成的 lex.yy.c 文件中
同时其也支持类 C 的注释
匹配的 pattern 与一般的正则表达式类似，在此略去
如果当匹配到一个字符串时，会将其存入到一个叫做 yytext 的 char 型数组中，可以其他地方引用它
对于匹配到后的一些操作，除了自己可以定义的一些操作之外，其本身提供了一些操作，例如 REJECT，BEGIN 和 ECHO,下面对其中的一些进行说明：
ECHO:将 yytext 的字符输出
BEGIN：其可以设定一些起始的条件，使一些匹配直到触发某些条件后才能生效
REJECT：即虽然已匹配到这个字符，但还已这个字符继续进行剩下规则的匹配
yymore()：即在下一次匹配到这个字符时将其加到 yytext 中而不是替换
yyless(n)：将匹配到的字符串的前 n 个字符返回到输入流之中
一些其提供的变量：
yytext：已经在之前提过
int yyleng：当前字符串的长度

FILE *yyin：输入文件的指针
void yyrestart(FILE *new_file)：从新文件指针重新开始
FILE *yyout：输出文件的指针
YY_CURRENT_BUFFER：返回当前状态的一个句柄
YY_START：返回当前的开始条件
其还有一大堆的选项可以在 lex 文件中定义，在此略去
对于其的使用，只要输入 flex 并将 lex 文件名输入，其会在当前目录下生成一个默认名字为 lex.yy.c 的文件，可以将其与使用者提供的一些文件一起编译生成一个简单的词法分析器。

3. 用 flex 生成 C1 的 Lexer
利用两个变量 identifier 和 num 记录返回的 yytext，并在 printToken 函数中打出来

4.阅读 clang 源码

Token.h:

在该头文件中主要是定义了一个叫做 Token 的类，用于定义词法分析中扫描到的各种 token 的一些特性

成员变量的解释：
Loc:当前 token 的位置
UintData:当 token 是一个普通的 token 时其包含 token 的长度，当 token 是注释是其包含 token 的结尾
PtrData: 指向当前数据的指针
Kind：当前 token 的类别
Flag：一些 token 的 flag，每位表示不同的意思，每位在之后有解释
TokenFlag: 一个枚举类型，其中包括 flag 每位代表的含义
StartOfLine：在一行的开始(看变量名也能看出来吧)
LeadingSpace：在 token 之前的空格
DisableExpand：禁止标识符进行宏展开
NeedsCleaning：包含三连词和续行符
LeadingEmptyMacro: 其之前包含空的宏定义
HasUDSuffix: 其含有用户定义的后缀
HasUCN: 其包含 UCN(universal character name)
IgnoredComma: 这个逗号不是宏定义的分隔符
StringifiedInMacro: 该 string 是由宏定义中的 stringizing 或者 charizing 运算符(#和#@)组成

成员函数(全为 public)：
大部分是 trivial 的，比如各种 set……,get……和 is……,看到名字就可知其作用，在此略去不表，仅介绍几个比较重要的函数
void startToken():
该函数主要进行初始化，即将所有的成员变量都置零
bool is(TokenKind K) const：
其判断当前 token 的种类是否为 K,若不是返回 false
还有一个 isNot 函数刚好与其相反，在此亦略去

Lexer.h & Lexer.cpp:

在 Lexer.h 文件中，其定义了一个名为 Lexer 的类，对词法分析的行为进行了定义。
主要的成员变量有：
const char *BufferStart：buffer 的开始指针
const char *BufferEnd：buffer 的结束指针
SourceLocation FileLoc：文件的路径
LangOptions LangOpts：语言的一些选项

const char *BufferPtr：当前的指针，即词法分析器当前处理到的位置
其他都是一些 flag 变量

主要的成员函数有：

DiagnosticBuilder Diag (const char *Loc, unsigned DiagID) const：
这个函数是词法分析中进行错误反馈的一个函数，其他函数在发现错误时调用它，它只是创建了一个 DiagnosticBuilder 对象并将其返回，具体原理在后面诊断函
数处进行说明。

static std::string Stringify (const std::string &Str, bool Charify=false)：
其去除 str 中的转义字符与“”(其同名的一个重载函数也是完成类似功能)

getSpelling(其重载较多就不放它们的声明了):
得到当前 token 的 spelling，若其不需要特殊处理则返回其的一个拷贝，否则交给 getSpellingSlow 做

static SourceLocation AdvanceToTokenCharacter (SourceLocation TokStart, unsigned Character, const SourceManager &SM, const LangOptions &LangOpts):
可以认为其返回当前 token 前进 character 个位置后的位置，但要考虑换行符和 trigraph

getCharAndSize 系列:
若当前字符不是'/'和'?',则其相当于 getchar，只是返回了一个 size；若是则交给其的 slow 版本处理
其中有 Nowarn 版本，只是不输出警告信息

bool Lexer::Lex(Token &Result):
这个函数中，其首先检查了几个 bool 类型的全局变量，并且修改了 result 中的相关 flag，然后调用 LexTokenInternal 函数处理。

bool Lexer::LexTokenInternal(Token &Result, bool TokAtPhysicalStartOfLine):
该函数为整个文件词法分析的核心部分，其对各种情况通过调用相关函数进行处理并将结果输出至 result 对象中。
下面对一些典型的进行举例说明：
LexNumericConstant:
其为处理常量数字的函数
若当前指针指向的字符为普通的数字，则 CurPtr 一直前进
下面为处理一些特殊情况：
若不在微软模式或者为 16 进制(只需检查这个 token 前两个字符是不是“0x”或者“0X”即可)，则将之后的指数也算入一个 token，即类似 0x12E+12 之类的
若语言为 c99 或为 16 进制且字符中无'_'，则将其作为 16 进制浮点数处理，类似 0x1.2p+1,表示 1.01*2^1
若字符串中含'‘'且支持 c++14 的特性，则将其作为分隔符，直接忽略，对其前后进行分别处理
关于 UCN 和 UTF-8 的处理也类似
每次若遇到特殊情况均会递归调用自己，但 CurPtr 已是当前读的位置，直到处理完整个 token，最后调用 FormTokenWithChars 函数将 BufferPtr 和 CurPtr 之间的所
有字符组成一个整个的 token

LexIdentifier:
其为处理标识符的函数
若当前指针指向的字符为[_A-Za-z0-9]，则 CurPtr 一直前进，直至不是这些字符；
若下一个字符是 ASCII 码且不是\,?和$，则 identifier 结束，进行与处理数字时差不多的返回处理过程，只是若不在纯词法分析的模式下还要在标识符表中进行
查找，然后把工作交给预处理的类

对$的支持由一个 flag 控制，若为真则将其当做组成标识符的字符，否则返回一个错误信息
对 UTF8 和 UCN 的处理在此略去
与处理数字的有些不同，其设置了一个标识符 FinishIdentifier，每次可以结束时即跳转到该处

SkipLineComment:
其为处理单行的注释的函数
一开始先检查单行注释模式是否开启，若没开启则报警告
其主要的处理部分在一个 do...while 循环中，出循环的条件是其不遇到换行符或者回车
在循环中有以下几种情况：
其创建了一个 EscapePtr 用来指向下一行的开始，
一般情况：把这行的字符读完，未读到换行符，退出循环
遇到文件结尾：调用结束词法分析的函数并返回 false
若在行尾读到换行符其继续循环
若在行尾的换行符后读到空格则发出一个警告
若下一行也是以"//"开始的注释，则进入下一次循环，与处理上一行的方式一样
对于其处理结果的部分，其依据模式决定是否保存注释，然后吃掉\n，并处理相关的 flag 并返回

SkipBlockComment:
其为处理多行注释即在“/**/”之间的注释的函数
LexerInternal 函数在调用它时已经将注释前的“/*”去掉了，因此其最先开始读一个字符并判断是否到文件结尾
其主要部分是一个 while(1)的循环，通过 break 或者 return 语句退出循环。
在进循环前先检查下一个字符是否是'/'，即不认为"/*/"是一个完整的注释
在循环之中，为了加快速度，其每个小循环最多检查 16 个字符，另一个循环每次同时检查 4 个字符，直至找到'/'或'\0'为止.
当找到'/'时，若其之前一个字符是'*'或者其在两者之间用了换行符，则所有注释已经读完，退出循环
若在其中找到嵌套的注释，则会发出一个警告
而如果到文件结尾，则结束词法分析过程
退出循环后依旧是和行处理类似的善后过程，在此略去不表

接下来是一些问题的回答：
关于 goto 语句：
在 lexer 的源文件中，使用了很多 goto 语句，仅在 LexerTokenInternal 函数中就接近 20 次，而以往的经验告诉我们尽量少用 goto，而在此处：
由于一个函数很长，用 goto 语句可以实现模块化设计，使程序的条理更加清晰，例如在 LexerTokenInternal 函数中，其分为 LexerNextToken 和
HandleDirective，在 LexerNextToken 中还有一个 SkipIgnoredUnits
其次整个函数可以认为是一个有限状态机，使用 goto 语句相对于其他方式可以更加清晰的表示状态的转换情况
最后可能是性能方面的考虑，相对于函数调用，goto 理论上可以更加节省时间，因为其只需修改一个寄存器的值即可，而函数调用需要传参压栈等

对于位置信息的记录：
这个之前在说明各个函数原理时也提到过，在此统一讲一下。其中有几个 const 的指针，BufferStart,BufferPtr,CurPtr 和 BufferEnd,分别指向 buffer 中的各个位置和当前的位置
BufferStart：buffer 的开始指针
BufferPtr：当前处理的 token 前的指针
CurPtr: 词法分析器当前处理到的位置，可能在一个 token 之中
BufferEnd：buffer 的结束指针
例如在 LexNumericConstant 函数中，其用 CurPtr 记录当前处理到的位置，每次依据读到的字符

size 前进，直到完成整个 token 的读取，最后将 BufferPtr 和 CurPtr
一减就是这个 token 的 size
识别出的字符串转化为 token：
每次识别出一个字符，其将 CurPtr 向后移动相应 size 个单位，直至识别结束后，将 CurPtr 与 BufferPtr 的值相减就是这个 token 的 size,最后调用 FormTokenWithChars
函数完成对 token 的创建，其只保留 token 的起始地址，大小和类型，并没有保存一个备份
这样做的目的是减少内存开销和提高词法分析的效率

Diagnostic.h & Diagnostic.cpp

这一部分比较复杂，我写了一个小程序，在最后一个函数末尾去掉了一个'}'，然后将其送入 clang 进行编译，并使用 gdb 调试(之前在 makefile 的编译选项中加了-g)。
在 FormatDiagnostic 函数处加了断点，发现运行后其调用栈是这样的

```
caz@caz-SVP11217SCS: ~/download
<char const*>) ()
#5  0x00000000006d984b in main ()
(gdb) n
Single stepping until exit from function _ZN5clang21TextDiagnosticPrinter16Handl
eDiagnosticENS_17DiagnosticsEngine5LevelERKNS_10DiagnosticE,
which has no line number information.

Breakpoint 11, 0x0000000001840990 in clang::Diagnostic::FormatDiagnostic(llvm::S
mallVectorImpl<char>&) const ()
(gdb) bt
#0  0x0000000001840990 in clang::Diagnostic::FormatDiagnostic(llvm::SmallVectorI
mpl<char>&) const ()
#1  0x000000000075c939 in clang::TextDiagnosticPrinter::HandleDiagnostic(clang::
DiagnosticsEngine::Level, clang::Diagnostic const&) ()
#2  0x00000000018467ae in clang::DiagnosticIDs::ProcessDiag(clang::DiagnosticsEn
gine&) const ()
#3  0x00000000018408fa in clang::DiagnosticsEngine::EmitCurrentDiagnostic(bool)
    ()
#4  0x000000000007a614f in clang::driver::Driver::BuildJobs(clang::driver::Compil
ation&) const ()
#5  0x000000000007a1d97 in clang::driver::Driver::BuildCompilation(llvm::ArrayRef
<char const*>) ()
#6  0x00000000006d984b in main ()
(gdb)
```

然后我便由调用堆栈开始找打印错误信息的地方，但在 BuildJobs 函数中并没有找到 EmitCurrentDiagnostic 函数，最后发现其实是一个 DiagnosticEngine 类的对象在返回时
调用了析构函数，从而调用了 EmitCurrentDiagnostic 函数

根据调用栈我找到了 TextDiagnosticPrinter 类下的 HandleDiagnostic 函数，在这个函数中，其先调用 FormatDiagnostic 函数将其格式化，主要是进行
，然后在 printDiagnosticMessage 函数中将诊断信息输出到一个叫 OS 的输出流中，直到退出编译过程回到 main 函数后才会一起打到默认的标准输出中

根据不同诊断信息的重要程度，其对之设定了各种 level，可以通过设定 flag 来决定是否需要对其进行输出

总的来说，在遇到一些错误或者需要输出诊断信息的情况时，程序会创建一个 Diagnostic 对象，并将需要输出的信息通过一个 ID 的方式输出给它，然后其便启动诊断处理程序，在结束函数调用时自动调

用其对象的析构函数从而将对象中包含的错误信息先进行格式化后输出到一个输出流中。