lexer paser 协同工作parser会lookahead至少一个token，所以可有如下两种方案：lexer运行整个文件，再让parser生成AST；或是生成了一小部分用于parser来完成相应工作，这意味着两者交替运行。前者考虑是源于C++是任意lookahead的，因为语法并非上下文无关，但这也许会消耗相当多的内存。

实际情况比较接近后者，但也并不完全这样，因为典型情况下lexer和parser都用相对简单的状态机实现的，lexing过程由如下两个条件驱动 外部索取一个新token或 一个新的字符到来，而parser状态机由如下两个条件驱动 ： 索取一个parse或 来了一个新的token

这两个方向分别对应top-down, bottom-up的实现，传统方案来说后者居多，以构造AST为中心

阅读手册：

annotation tokens

在parser中被合成，并注入预处理器的toke流中，以替代原有的token。

这使得parser在回溯时，不需要重做语义分析，来确定一个token 序列是一个变量类型，模板等。

注释标记的有效字段与普通标记不同，但它们被多路复用到普通Token中，

SourceLocation “Location” 标记第一个被替换的token

SourceLocation “AnnotationEndLoc” 标记最后一个被替换掉token

void\* “AnnotationValue” 这包含parser从中获取的Sema对象，parser仅保留信息，以供Sema以后根据注释标记类型进行解析

TokenKind “Kind 表示token种类

包括 tok::annot\_typename 表示一个已解析的类型名标记

tok::annot\_cxxscope 表示一个c++范围说明符

tok::annot\_template\_id 标记C++模板ID

DeclContext class

每种声明在程序中存在一些声明的上下文，如翻译单元，名称空间，class ，function，这些声明上下文在clang中被定义为DeclContext 类，包括(TranslationUnitDecl, NamespaceDecl, RecordDecl, FunctionDecl) 本次实验的pragma overflowcheck所需要的上下文为FunctionDecl(FD), DeclContext提供包括存储声明，查找声明，声明所有权查找等功能