实验导引（一）

2021 年 9 月

1. MiniDecaf 编译实验项目综述

1.1 项目回顾

2001 年，我们引进了 Stanford 课程 CS143 [1] 的课程实验框架（其原始框架由 Julie Zelenski 设计）。该实验框架设计实现一种简单面向对象语言 Decaf 的编译器，因此称之为 Decaf 项目。

Decaf 是一种强类型的、单继承的简单面向对象语言，是用于教学的语言，曾经在 Stanford，MIT，University of Tennessee，Brown，Texas A&M，Southern Adventist 等多所大学的相关课程中使用。Decaf 仅代表一种语言设计的理念，各校的课程实验框架和语言版本不尽相同。

在 98 级本科生的“编译原理”课程（2001 年秋季学期）中，我们首次采用了 Decaf 项目。根据实际需要，之后我们对实验框架进行了一定的调整与简化，以及对源语言进行适当的改动等。比如在 02 级，我们对该项目进行一定的简化之后，称之为 TOOL 项目。

从 03 级的课程之后，我们对原始的 Decaf 项目实验框架进行 3 次实质性改动：

- 在 03~04 级的 Decaf 项目中，我们将原先实验框架的开发语言由 C++ 改为 Java。
- 在为计 50 班（2005 级姚班）单开的“编译原理”课程中，我们参考了 U.C.Berkeley 课程 CS164 [2] 的 COOL 课程项目（设计者 Alex Aiken），将实验框架由原来的单遍组织改为多遍组织，称之为 Mind（Mind is not decaf）项目。同时，对 Decaf 语言进行了较大精简，称之为 Mind 语言。
- 由于计 50 班的编译课程安排在 Java 程序设计课程之前，所以首次 Mind 项目的开发语言为 C++。随后，在 2005 级其他班（计 51~计 55）的课程中，我们又将开发语言由 C++ 改回到 Java。

从 06 级开始，实验框架没有发生大的变动，只是对其进行微调或是进行适当简化，语言特征在 Mind 语言基础上有所扩展。如，增加 static、instanceof 等。08 级仅对针对抽象语法树（AST）的设计进行了改动。

有些年级的实验曾尝试增加手工实现语法分析程序的环节。

之前的实验框架均是以 Java 为实现语言，17 级开始尝试增加 Rust 和 Scala 版本。18 级开始进行了新一轮的课程实验改革，称为 MiniDecaf 项目，简化输入语言为 C 语言的子集，并打通 RISC-V 后端代码生成环节，让学生一开始就看到一个简化而可运行的完整编译器，并通过渐进式开发不断完善这个编译器。19 级进一步将 Decaf/Mind 的词法语法分析、AST 设计、TAC 代码生成等模块移植到 MiniDecaf 项目中，形成了 Python/C++两种语言实现的编译器框架，使课程实验进一步贴合教学内容。

附录 A 列举了历年来参与 Decaf/Mind 编译实验项目维护的大部分助教信息，仅供参考。

1.2 MiniDecaf 项目框架的总体结构

MiniDecaf 项目的实验框架是设计实现 C 语言子集的编译器，该编译器的工作原理如图 1 所示。我们将实验框架分成如下 5 个阶段：

阶段一：自动构造工具实现词法和语法分析程序。借助 Lex 和 Yacc 实现词法和语法分析一遍扫描源程序后直接产生一种高级中间表示（实验指定的抽象语法树 AST）。针对不同的实现语言，Lex 和 Yacc 版本会有所不同，比如，在 C++版的实现框架中所使用的 Lex 和 Yacc 版本分别为 Flex 2.6.4 [3] 和 Bison 3.7.6（类 Yacc）[4]。Python 版的实现框架中所使用的 Lex 和 Yacc 版本为用 Python 语言实现的 ply 包中的 ply.lex 和 ply.yacc [5]。

阶段一（手写词法语法分析器）：基于递归下降分析程序实现语法分析。参考课程组助教提供的参考文档，手工编写满足需求的递归下降分析程序。用所完成的递归下降分析函数替代自动构造工具生成的相应函数，可以实现与自动构造工具相同的任务（一遍扫描源程序后直接产生抽象语法树）。

阶段二：语义分析。遍历抽象语法树构造符号表、实现静态语义检查（非上下文无关语法检查以及类型检查等），产生带标注的抽象语法树。这一阶段将对抽象语法树 AST 进行两遍扫描：第一遍扫描完成符号表的建立，并且检测符号声明冲突以及跟声明有关的符号引用问题（例如，函数 A 调用函数 B，但是函数 B 没有定义）；第二遍扫描检查所有的语句以及表达式的参数的数据类型。

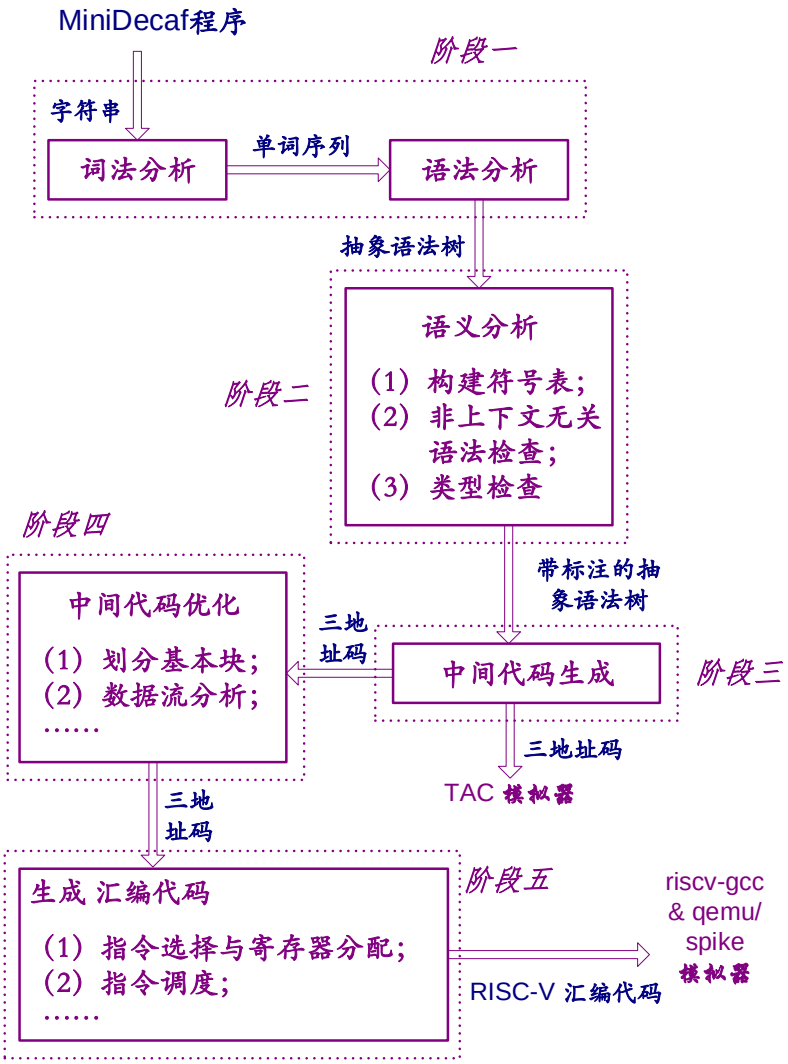

图 1　MiniDecaf 编译器总体结构

阶段三：中间代码生成。将带标注的抽象语法树（decorated AST）所表示的输入程序翻译成适合后期处理的另一种中间表示方式，即三地址码 TAC，并在合适的地方加入诸如检查数组访问越界、数组大小非法等运行时错误处理的内容。阶段三完成后所得的三地址码程序可在 TAC 模拟器上执行。

阶段四：中间代码优化。主要是基于 TAC 代码实现简单的数据流分析（如活跃变量数据流、到达-定值数据流、UD 链和 DU 链等）模块，也可以进行简单的优化（如常量合并、常量传播、公共子表达式消除等）。经过数据流分析所求得的活跃变量信息，后续的寄存器分配具有重要支撑作用。

阶段五：目标代码生成。实验框架包括汇编指令选择、寄存器分配和栈帧管理。实验框架中包含一个面向 RISC-V 架构的后端，可生成适合实际 RISC-V 机器上的汇编代码。这些 RISC-V 汇编代码也可以利用开源模拟处理器 QEMU [6] 或 Spike [7] 进行模拟执行。这一阶段的实验框架仅实现了一种启发式的寄存器分配算法。如果同学们有兴趣，可以思考如何实现更加高效的寄存器分配算法（比如，基于图着色方法的寄存器分配算法），并生成正确的汇编代码。

1.3 2021 秋 MiniDecaf 项目总述

2021 秋的课程实验要求同学们实现一个小型语言 MiniDecaf（C 的小子集）的编译器，经过 6 个阶段（12 个步骤）的学习与开发过程，使编译器的功能不断完善，从支持仅有一个 return 语句的 main 函数，逐步支持常量表达式、变量、语句、作用域、控制语句、函数、全局变量、数组等语言特性。此外，在词法语法分析方法学习的过程中，为了贴合词法语法分析的重点教学内容，除了要求同学们掌握由自动构造工具实现词法和语法分析程序的方法，也要求大家能够手写词法语法分析器，由此设置了一个对应的实验环节（关卡）。

课程实验的参考文档请见如下链接（通知之时起生效）：

https://decaf-lang.github.io/minidecaf-tutorial/

2. 阶段一（自动构造工具）实验指导

针对每个阶段的实验，都为同学们提供了必要的实验文档和基础框架，帮助大家较详细地了解实验要求和实验内容。这一讲，我们先对阶段一对应的实验内容做简短的描述。

2.1 词法分析

词法分析的功能是从左到右扫描 MiniDecaf 源程序的字符流，识别出一个个的单词。所识别的每一个**单词**，是下一个有意义的词法元素，如标识符、保留字、整数常量、算符、分界符等。每识别出一个单词，词法分析程序都会产生一个**单词记录**（包括单词类别，单词值，及源程序中位置等信息），以传递给后续阶段使用。在单词识别的过程中，还需要检测词法相关的错误，例如字符@并非 MiniDecaf 程序中的合法符号，若这个字符在注释以外出现，则需要向用户提示一个词法错误。

例如，对于以下 MiniDecaf 程序片段 P1 ：

```
int main() {
    int a = 2021;
    return a;
}
```

词法分析程序所识别的单词序列参见表 1，其中列举了每个单词的类别和值。

对于每一个单词类别，编译器都有一个内部表示，我们称之为**单词记号**（token）。在后续的语法分析中，这些单词符号对应于语法规则中的终结符。

单词类别	单词值
保留字 **int**	
标识符	**main**
分隔符 **(**	
分隔符 **)**	
分隔符 **{**	
保留字 **int**	
标识符	**a**
操作符 **=**	
整数型常量	**2021**
分隔符 **;**	
保留字 **return**	
标识符	**a**
分隔符 **;**	
分隔符 **}**	

表 1　识别单词（得到单词类别/单词值等信息）

2.2 语法分析

语法分析是在词法分析的基础上将单词符号串分解成各类语法短语，如“程序”，“语句”，“表达式”等，建立起语法分析树。建立语法分析树的过程，就是识别出符合语法规则的 MiniDecaf 程序的过程。对于不符合语法规则的 MiniDecaf 程序，将会报告语法错误。比如常见的少写分号的问题，就属于语法错误，会在这个阶段被发现。

这里的语法规则是由一个上下文无关文法定义的，每个产生式都是一条规则，可识别一类语法短语。对于我们实验框架中的源语言，一个可能的上下文无关文法（片段）为 G[Program]：

```
Program -> Function

Function -> Type identifier '(' ')' '{' Block '}'

Block -> StatementList

StatementList -> Statement StatementList | ε

Statement -> Type identifier '=' intLiteral ';' | 'return' identifier ';'

……
```

文法 G[Program] 中，大写字母开头的符号为非终结符，其余是终结符（对应于单词符号）。（请注意，在采用自底向上的语法分析方法进行语法分析时，“StatementList -> StatementList Statement”一般比“StatementList -> Statement StatementList”的移进归约效率更高）。

实现语法分析的过程是归约或推导，对于符合语法规则的源程序，分析结果得到一棵语法分析树。例如，对于上述程序片段 P1，得到如图 2 所示的语法分析树。

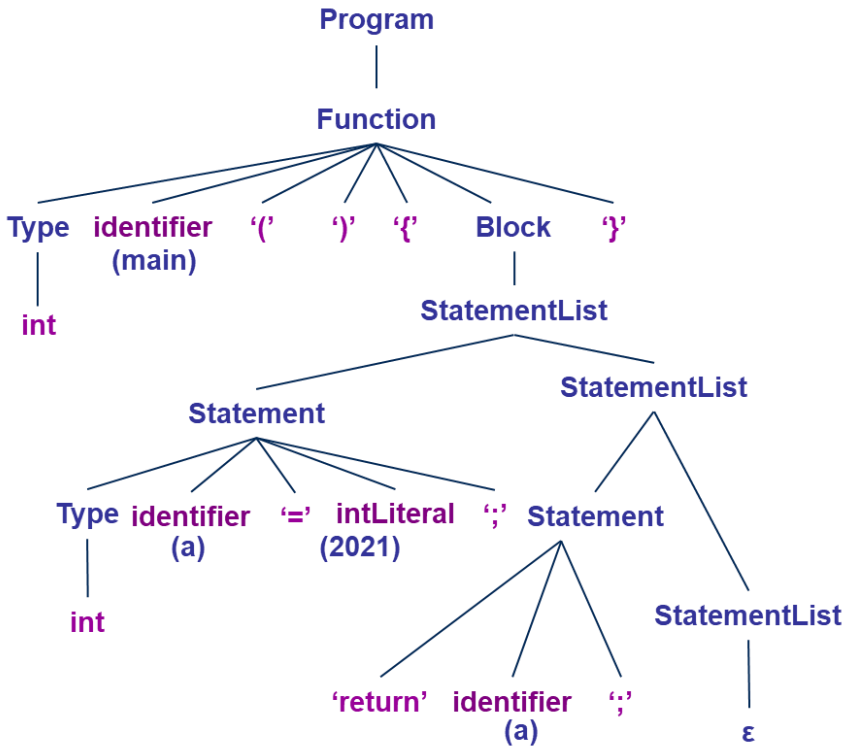

图 2　语法分析树（具体语法树）

2.3 抽象语法树

虽然语法分析的结果得到一棵语法分析树，但我们在后续阶段将不使用它，而是使用一种更加实用的**抽象语法树**（AST，Abstract Syntax Tree）。抽象语法树是一种只存储编译关键信息的语法树表示形式，其特点包括：（1）不包含我们不关心的终结符，例如逗号、分号等（实际上只含标识符、常量等终结符）；（2）不具体体现语法分析的细节步骤，例如对于 $A \rightarrow A\ E \mid \varepsilon$ 这样的规则，按照语法分析的细节步骤来记录的话应该是一棵二叉树，但是在 *AST* 中我们只需要表示成一个链表，这样更易于后续处理；（3）能够完整体现源程序的语法结构，使用 *AST* 表示程序的好处是把语法分析结果保存下来，后续过程可以反复使用。

相对于抽象语法树，我们把图 2 中的语法分析树称为**具体语法树**，它更适合于指导语法分析过程，而不方便后续遍历使用。

例如，图 3 是上述程序片段 P1 的一种抽象语法树表示形式。

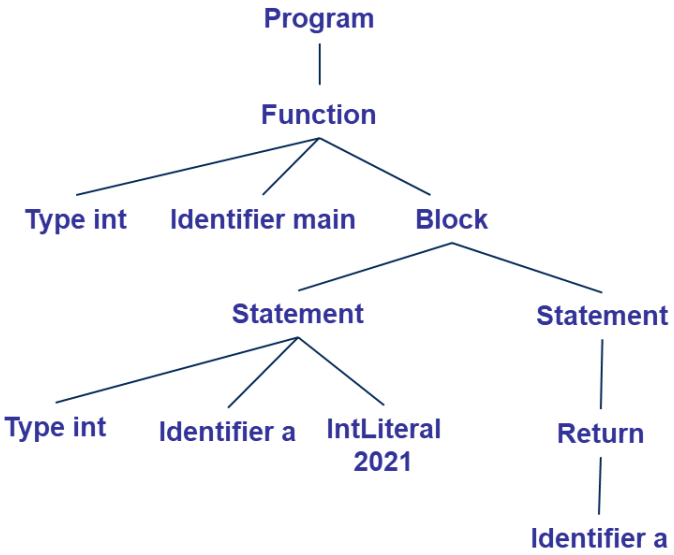

图 3　抽象语法树

实验代码框架中给出了代表性的抽象语法树结点对应的数据结构（具体文件请参考课程实验文档）。在动手实现实验要求之前，建议熟悉这些数据结构。

2.4 前端实验内容

前端实验的重点是掌握 Lex 和 Yacc（课程实验中的 Bison 与 Yacc 用法基本一致）的用法，体会使用编译器自动构造工具的优势，并且结合实践体会正规表达式、有限自动机、上下文无关文法、LALR(1)分析、语法制导翻译等理论是如何在实践中得到运用的。

使用 Lex 和 Yacc 的核心内容在于基于正规表达式给出词法规则，以及基于上下文无关文法给出语法规则。另外还要注意 Lex 和 Yacc 是如何联用的。

在我们所提供的文档中，源语言的语法定义是通过 BNF 形式给出的。这要求我们在实验中，在理解这种形式的语法定义基础上，给出等价的上下文无关文法定义。另外，还要注意的是，Lex 中的正规表达式在形式上要比我们在“形式语言与自动机”课程中学习的形式（只含三种运算）更加丰富。

在第 3 节，我们将对 Lex 和 Yacc 的使用进行简要介绍。然而，为轻松完成实验，建议大家还要较系统地查阅和学习 Lex 和 Yacc 的用户手册。

本次实验中，建议大家在开始前先读懂所提供的实验框架。特别要关注 AST 的数据结构以及 Lex 和 Yacc 的联用等。

3. Lex & YACC 简介

Lex 是一个实用的词法分析程序自动构造工具，Yacc 是一个实用的语法分析/语义处理程序自动构造工具，二者早期作为 UNIX 操作系统的实用程序发布，后来根据需求衍生出多种版本。图 4 是使用 Lex 和 Yacc 工具的一个简单示意图。

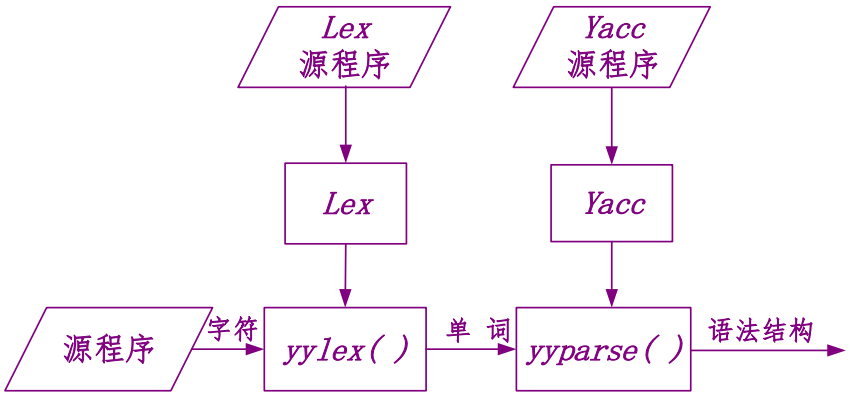

图 4　Lex 和 Yacc 工具的使用

Lex 工具的功能是读入用户编写的一个 lex 描述文件，生成一个名为 lex.yy.c 的 C 源程序文件。lex.yy.c 中包含一个核心函数 yylex()，它是一个扫描子程序，读入源程序的字符流，识别下一个单词，并返回单词记录。

Yacc 工具的核心功能是读入用户编写的一个 yacc 描述文件，生成一个名为 y.tab.c 的 C 源程序文件。y.tab.c 中包含一个函数 yyparse，它描述了一个基于 LR 分析表的 LR 分析程序，并且可以实现基于这个分析程序的语义处理。每当 yyparse 需要下一个单词记录时，它就调用称为 yylex() 的词法扫描子程序返回下一个单词记录的动作。yylex()可以由用户自己编写，也可以通过 Lex 自动生成。

3.1 Lex

下面我们分几个小节简要介绍 *lex* 描述文件的格式和内容，*lex* 的使用，*lex* 和 *yacc* 联用时的接口约定等。

3.1.1 *Lex* 描述文件中使用的正规表达式

Lex 描述文件中，在书写词法单元的识别规则时，需要用到正规表达式。以下列举了 lex 中主要的正规表达式表示形式：

- x ， 可以匹配字符 x。
- . ， 可以匹配除换行符 '\n'之外的任何字符。
- 用 '[' 和 ']' 括起来的字符列表，可以匹配该字符列表中的所有字符。字符列表中除字符外还可以出现由间隔符'-'表示的字符范围。如，[xyz]，匹配字符 'x'，'y'，或'z'。又如，[x-yA4-6O] ，匹配字符'x'，'y'，'z'，'A'，'4'，'5'，'6'，或'O'。除了 \ 之外，其它元字符在方括号中没有特殊含义。若第一个字符是 '-'，则不被当作元字符。
- 用 '[^' 和 ']' 括起来的字符列表，可以匹配该字符列表之外的所有字符。如，[^A-Z]，匹配所有除大写字母之外的字符。特别地，[^]，可匹配任何字符。

- 用双引号‘”’括起来一个串，可以匹配这个串本身。这个串里面的所有元字符，除了 \ 和 ” 之外，都会失去元字符的作用。例如，可用 ”if” 匹配序列 if，可用“[”匹配单个左方括号，可用 “/*” 匹配序列 /*。
- 用 ‘{’ 和 ‘}’ 括起来的正规表达式宏名字，相当于将这个宏名字展开为相应的正规表达式。 正规表达式宏名字的定义见下一小节。
- 除了加双引号‘”’，匹配单个元字符的另一种方法是利用转义字符 \ 。例如，*，可匹配一个字符‘*’；如果需要匹配序列*，就必须写作 *。若 \ 后面的字符是小写字母，则可能表示 C 转义字符，如 ‘\t’ 表示制表符。
- 反斜杠 \ 后面跟 8 进制数值，而 \x 后面跟 16 进制数值，则匹配这个数值对应的 ASCII 字符。如，‘\0’匹配 NUL 字符 (ASCII 码为 0) ，‘\123’ 匹配 8 进制数 123 对应的 ASCII 字符，‘\x2a’ 匹配 8 进制数 2a 对应的 ASCII 字符。
- r* ， 匹配正规表达式 r 的星闭包。
- r+ ， 匹配正规表达式 r 的正闭包。
- r? ， 匹配正规表达式 r 的任选。
- r{n} ， 匹配正规表达式 r 的 n 次幂。
- r{m, n} ，匹配正规表达式 r 的 m 到 n 次幂。
- r{m, } ， 匹配正规表达式 r 的大于等于 m 次幂。
- (r) ， 匹配正规表达式 r，括号用于重新规定优先级。
- rs ， 匹配正规表达式 r 与正规表达式 s 的连接。
- r|s ， 匹配正规表达式 r 与正规表达式 s 的并。
- r/s ， 匹配正规表达式 r，但仅限于随后的输入符号可以匹配正规表达式 s。 要注意，在确定是否可以匹配 s 期间不管读入过多少个输入符号都将被退回。
- ^r ， 匹配正规表达式 r，但仅限于在一行的开始处。
- r$ ， 匹配正规表达式 r，但仅限于在一行的结尾处。
- <c>r ， 匹配正规表达式 r，但仅限于开始条件为 c。开始条件用来区分不同上下文，其定义见下一小节。c 也可以是一个开始条件的列表，或者是 *，后者用来表示任意的开始条件。

关于lex 中正规表达式的正确使用还有不少技术细节的问题，限于篇幅我们不可能涉及所有细节，所以在实际应用中手头最好准备一份较详细的技术手册。

3.1.2 *Lex* 描述文件的格式

Lex 描述文件由三部分组成，各部分之间被只含 %% 的行分隔开：

辅助定义部分

%%

规则部分

%%

用户子程序部分

其中，“辅助定义部分”，“规则部分”，和“用户子程序部分”都是可选的，可以不出现。在没有“用户子程序部分”时，第二个%%也可省略。

“辅助定义部分”包含正规表达式宏名字的声明，以及开始条件的声明。它们可能出现在规则部分的正规表达式中，用法见 3.1.1。

声明正规表达式宏名字的格式为

宏名字　正规表达式

例如，

DIGIT　[0-9]

NUMBER　{DIGIT}+"."{DIGIT}*

这样，正规表达式中若出现 { NUMBER }，就相当于 ([0-9])+"."([0-9])* 。

开始条件的声明始于 %Start（可缩写为 %s 或 %S） 的行，后跟一个名字列表，每个名字代表一个开始条件。开始条件可以在规则的活动部分使用 BEGIN 来激活。直到下一个 BEGIN 执行时，拥有给定开始条件的规则被激活，而不拥有开始条件的规则变为不被激活。

开始条件主要是用来区分不同上下文。限于篇幅，这里不打算给出有关开始条件的声明和使用的例子。

“规则部分”是描述文件的核心，一条规则由两部分组成：

正规表达式　动作

“正规表达式”的形式参见 3.1.1。“正规表达式”必须从第一列写起，而结束于第一个非转义的空白字符。这一行的剩余部分即为“动作”。“动作”必须从正规式所在行写起。当某条规则的“动作”超过一条语句时，必须用花括号括起来。如果“动作”部分为空，则匹配该“正规表达式”的输入字符流就会被直接弃掉。

输入字符流中不与任何规则中的正规表达式匹配的串默认为将被照抄到输出文件。如果不希望照抄输出，就要为每一个可能出现的词法单元提供规则。

例如，以下描述对应的程序将从输入流中删掉 "remove these characters"：

%%

"remove these characters"

又如，以下描述对应的程序将多个空白或 Tab 字符约减为一个空白字符，同时滤掉每行行尾的所有空白或 Tab 字符：

%%

[\t]+ putchar(' ');

[\t]+$ /* ignore this token */

“动作”可以是任意 C 代码，包括 return 语句，它在 yylex() 被调时返回某个值。每一次调用 yylex() 之后，将会从上一次离开的位置继续处理输入字符流，直到文件结束或执行了一个 return 语句。

“动作”中可以用到 yytext，yyleng 等变量。其中，yytext 指向当前正被某规则匹配的字符串；yyleng 存储 yytext 中字符串的长度，被匹配的串在 yytext[0]～yytext[yyleng-1]中。

此外，“动作”中还允许包含特定的指导语句或函数：ECHO，BEGIN，REJECT， yymore()，yyless(*n*) ，unput(c) ，input() 等。技术细节可参考有关 lex 的技术文档。

在“辅助定义部分”和“规则部分”，任何未从第一列开始的文本内容，以及被 ‘%{’ 和 ‘%}’ 括起来的部分，将被复制到 lex.yy.c 文件中（不包括 ‘%{}’）。注意，这里的 ‘%{’ 必须从所在行的第一列开始。

在“规则部分”，出现在第一条规则之前的从第一列开始的或 ‘%{’ 和 ‘%}’ 括起来的部分里可以声明扫描子程序 yylex() 的局部变量，以及每次进入 yylex() 时执行的代码。

在“辅助定义部分”中，第一列开始的注释（即始于"/*"的行）也将被复制，直到遇到下一个"*/"。但“规则部分”中不可以这样。

最后，“用户子程序部分”中的调用扫描子程序或被扫描子程序调用的所有 C 函数将被原样照抄到 lex.yy.c 文件中。

值得提到的是，当遇到文件结尾时，词法分析程序将自动调用 yywrap() 来确定下一步做什么。如果 yywrap() 返回 0，那么就继续扫描；如果 yywrap() 返回 1，那么就认为对输入串的处理已结束。*Lex* 库中的 yywrap() 标准版本总是返回 1。用户可以根据需要在“用户子程序部分”写一个自己的 yywrap()，它将取代 *lex* 库中的版本。

例 分析由下列 *lex* 描述文件，说明由它产生的扫描子程序的功能。

```
%{
            int num_lines = 0, num_chars = 0;
%}
%%
\n          {++num_lines; ++num_chars;}
.           {++num_chars;}
%%
main(){
    yylex();
    printf(  “# of lines = %d, # of chars = %d\n",num_lines, num_chars );
}
```

解 首先，第 1 行到第 3 行都位于分隔符%{和%}之间，这些行将被直接插入到由 *lex* 产生的 C 代码中，它将位于任何过程的外部。第二行中定义了两个全局变量：行计数器 num_lines 和字符计数器 num_chars。

在第 4 行的%%之后，第 5、6 行描述了两个规则。在第一个规则中，正规表达式只包含一个换行符 '\n'，对应的动作是行计数器 num_lines 加 1，以及字符计数器 num_chars 加 1。在第二个规则中，正规表达式是 '.'，可以可以匹配除换行符'\n'之外的任何字符，对应的动作是字符计数器 num_chars 加 1。

最后，在“用户子程序部分”中包括了一个调用函数 yylex() 的 main 函数，且输出行计数器 num_lines 和字符计数器 num_chars 的值。

由它产生的扫描子程序的功能是：统计并输出给定输入文本中的行数和字符数。

3.1.3 *Lex* 的使用

设上一节例子中的 *lex* 描述文件的名字为 *count.l*。在 *Linux* 环境（假设安装了相应的开发包，并且设置了正确的环境变量）中，可以通过下列以下步骤编译和执行：

```
$  lex  count.l
$  cc  -o  count  lex.yy.c  -ll
$  ./count  < count.l
    ……
$
```

其中，$ 为系统提示符。

第一行命令执行后，将会产生文件 lex.yy.c。

第二行命令是用编译器 cc 对 lex.yy.c 进行编译 。选项 '-o count' 指定了可执行文件名为 count，不指定时默认为 a.out。'-ll' 是 *lex* 库文件的选项。

第三行是执行 count。输入参数是文件 count.l 中的文本。执行结果将输出文件 count.l 中文本的行数和字符数。

例 给定 *lex* 描述文件 toupper.l 如下：

```
%{
    #include <stdio.h>
}%
%%
[a-z]        Printf("%c",yytext[0]+'A'-'a')
%%
```

试指出正确执行如下命令序列后的输出结果：

```
$  lex  toupper.l
$  cc  -o  toupper  lex.yy.c  -ll
$  ./toupper  <toupper.l
```

解 输出结果为：

```
%{
    #INCLUDE <STDIO.H>
```

```
}%
%%
[A-Z]        PRINTF("%C",YYTEXT[0]+'A'-'A')
%%
```

3.1.4 与 Yacc 的接口约定

Lex 的一个主要应用方面是与 *yacc*（参见 3.2 节）的联用。*Yacc* 产生的分析子程序在申请读入下一个单词时将会调用 *yylex()*。*yylex()* 将返回一个单词符号，并将相关的属性值存入全局量 *yylval*。

为了联用 *lex* 和 *yacc*，需要在运行 *yacc* 程序时加选项 '-d'，以产生文件 'y.tab.h'，其中会包含在 *yacc* 描述文件中（由 '%tokens' 定义）的所有单词符号。文件 'y.tab.h' 将被包含在 *lex* 描述文件中。

例如，如果有一个单词符号是 INTEGER，那么 *lex* 描述文件的一部分可能是：

```
%{
#include "y.tab.h"
extern int yylval;
}%
%%
0|[1-9][0-9]*          { yylval = atoi(yytext); return    INTEGER; }
[+*()\n]               { return yytext[0];}
.                      {   /*do nothing*/   }
%%
```

3.1.5 Flex

Flex 被认为是 Lex 的替代者，比 Lex 更加先进，效率更高。其用法与 Lex 工具相似。可以在 lex 文件（*.l）起始位置加入输出文件说明，如下：

```
%option outfile="scanner.cpp"
```

这样可以使 flex 输出指定名称的词法分析程序。Flex 的编译方式为：

```
flex   exp.l              /* 产生 lex.yy.c，其中包含 yylex()*/
```

此外，在程序链接过程中，可以通过-lfl 的 gcc 编译选项链接 flex 库文件。

3.2 Yacc

这一小节我们简要介绍 *yacc* 描述文件的格式和内容，以及 *yacc* 使用的例子。

3.1.4 *Yacc* 描述文件

Yacc 描述文件形如：

```
%{
声明部分
```

%}

辅助定义部分

%%

规则部分

%%

用户函数部分

其中，“声明部分”，“辅助定义部分”，和“用户函数部分”都是可选的，可以不出现。若“声明部分”为空，则 %{ 和 %} 的两行可去掉；若“用户函数部分”为空，则第二个 %% 的行也可去掉。这样，*yacc* 描述文件可以只包含“规则部分”，具有如下形式：

%%

规则部分

下面我们分别介绍各个部分所描述的最基本信息，关于 *yacc* 描述文件更多的内容读者可参考有关 *yacc* 的详细技术手册。

3.2.1.1 声明部分

“声明部分”定义常规的 C 声明，所有嵌在 %{ 和 %} 之间的内容将被原样拷贝至所生成的语法分析/语义处理程序中。在声明中，可以引入头文件、宏定义、以及全局变量的定义等。例如：

```
%{
#include  <stdio.h>
#define IDEN 5
int global_variable = 0;
%}
```

3.2.1.2 辅助定义部分

“辅助定义部分”主要包括如下几方面的定义：

- 定义语法开始符号。形如

  %start　非终结符

  如果语句 %start 被省略掉，那么规则部分第一条规则左端的符号将被认为是文法的开始符号。

- 语义值类型定义。缺省情形下，语义动作和词法分析程序的返回值为整型。其它语义值的类型（包括结构类型）可由 %union 声明，形如

  ```
  %union   {
        ……
        }
  ```

由 %union 声明的类型可以通过 <类型名> 的形式置于 %token，%type，%left，%right 和 %nonassoc 等之后，用于声明相应符号的语义值类型。缺省时，这些符号的语义值类型为整型。

- 终结符定义。形如

  %token 终结符

  例如，我们用 %token NUMBER ID 声明单词符号 NUMBER 和 ID，可作为文法的终结符。

  另，用作终结符的单个字符置于单引号之间；如，运算符 + 和 – 分别写作 '+' 和 '–'。

- 非终结符的类型说明。形如

  %type <类型名> 非终结符

  其中，<类型名> 可由 %union 声明。缺省时，非终结符对应的语义值类型为整型。

- 优先级和结合性定义。我们分别用%left、%right 和 %nonassoc 来定义左结合、右结合以及无结合的运算符。例如，为了声明运算符 + 和 – 具有左结合性，我们写作

  %left '+' '–'

  其中，运算符被单引号括起来，并且用空格分隔。对于运算符分好几行描述的情形，后边行中的运算符具有比前边行中的运算符更高的优先级；当然，同一行中的运算符具有相同的优先级。用于说明，我们来考虑

  %left '+' '–'
  %right UMINUS

  此例中，运算符 + 和 – 有同样的优先级，低于 UMINUS 的优先级，低于 UMINUS 的优先级。如果运算符出现在二义文法中，而需要构造 LR 分析过程，那么我们通过声明这些运算符的结合性和优先级常常可以做到这一点。

以下是一个"辅助定义部分"定义的片断：

```
%start Program
%union {
        …
        double doubleConstant;
        …
        char identifier[128];
        declaration*    decl;
    }
%token T_Void T_Bool T_Int
%token <identifier> T_Identifier
%token <doubleConstant> T_DoubleConstant
….
```

%type <decl> VariableDecl

3.2.1.3 规则部分

“规则部分”包含语法制导的语义处理所依据的翻译模式，由一个或多个规则（产生式）组成。各规则形如：

A : *Body* ;

其中，*A* 表示非终结符，*Body* 表示 0 个或多个名字及文字组成的序列，';' 为规则之间的分隔符。名字可以是终结符（单词符号）或非终结符；文字由单引号内的字符（含转义字符）组成。

如果若干规则具有同样的左边符号，则可用竖线‘|’来避免重复左边符号。此外，处于规则末端而在竖线之前的';'可以省略。这样，规则集合

A : *B C D* ;
A : *EF* ;
A : *G* ;

可以表示为

A : *B C D*
| *EF*
| *G*
;

若是 ε 规则，则可表示为：

A : ;

每个规则可以关联若干语义动作。语义动作既可以出现在规则的末尾，也可以出现在规则右端的中间位置。语义动作出现在规则的末尾时，*yacc* 在归约前执行它。语义动作出现在规则的中间时，*yacc* 在识别出它前面的若干文法符号后执行它。

规则中的每个文法符号以及语义动作本身都可以有自己对应的语义值。终结符（单词符号）的语义值由词法分析程序给出，并保存在 yylval 中。非终结符的语义值在语义动作中获得。在语义动作中可以通过$伪变量访问语义值，左部非终结符的语义值为$$，右部文法符号或语义动作的语义值依次为$1，$2，…。例如，如下规则

expr : '(' *expr* ')' { $$:= $2; }

表示按该规则归约时返回左端非终结符的语义值就是右边第二个符号返回的语义值。若规则中没有显式地为 $$ 赋值，则返回左端非终结符的语义值默认为是右边第一个符号或语义动作返回的语义值。

当语义动作位于右部第 *n* 个位置时，可以引用的语义值包括 $$，$1，$2，…，$*k*（*k*<*n*）；该动作的语义值为$*n*，动作中可以用$$为它赋值，后续动作中可以通过$*n* 引用它的值。例如，如下规则

A : *B* { $$:= 1; }
C { *x* := $2; *y* := $3; }

;

归约时语义动作的执行结果将 *x* 置为 1，*y* 置为 *C* 返回的语义值。注意，语义动作 { $$:= 1; } 中的 $$ 指向该动作本身的语义值，而不是指向左端非终结符 *A* 的语义值。如果语义动作 { *x* := $2; *y* := $3; } 中没有显式地为 $$ 赋值，那么归约后 *A* 的语义值将被置为 *B* 返回的语义值。

对处于中间位置的语义动作，*yacc* 内部的处理过程是增加一个新的非终结符和一条相应的ε-规则，当按照这条规则进行归约时触发这个语义动作。对于上一个例子中的规则，*yacc* 内部处理时就像是把这条规则替换为以下两条规则：

```
$ACT   :     { $$ := 1; }
       ;
A      :  B  $ACT  C  { x := $2; y := $3; }
       ;
```

其中，$ACT 为 *yacc* 处理过程中的一个内部符号。

为进行错误恢复，yacc 保留了一个名为“error”的特殊终结符，可以出现在规则中，用来表示预期的出错及进行错误恢复的位置。例如，若有如下规则

```
expr  :  error  ';'
```

当出现错误时，分析程序试图忽略过相应的语法成分（表达式），它从输入序列中滤掉除 *';'* 之外的单词记录；当遇到 *';'* 时，分析程序将根据这条规则进行归约，分析过程得以恢复。

为了更好地实现错误恢复，*yacc* 提供了一些特殊的语句或宏，如 *yyerrok*、*yyclearin* 等，可以与 *error* 配合使用，即用于 *error* 后的语义动作中。限于篇幅，这里不作进一步讨论，读者可以参考有关的技术手册。

3.2.1.4 用户函数部分

这一部分应当包含一个用 C 编写的主函数，它会调用分析函数 *yyparse*。

这一部分还应当包括一个错误处理函数 *yyerror*。每当分析程序发现语法错误时，将调用 *yyerror* 输出错误信息。*Yacc* 缺省的错误处理是遇到第一个语法错误就退出语法分析程序。

如果用户没有提供这两个函数，则会使用由库函数提供的缺省版本。如，缺省的 *main* 函数可能是：

```
main {
    return ( yyparse() );
}
```

缺省的 *yyerror* 函数可能是：

```
#include <stdio.h>

yyerror(char *s); {
    fprint ( stderr, "%s\n", s );
}
```

另外，“规则部分”的语义动作可能需要使用一些用户定义的子函数，这些子函数都必须遵守C语言的语法规定，这里不必赘述。

3.2.2 使用 yacc 的一个简单例子

下面介绍用 *yacc* 实现一个简单计算器的例子。同时，这也是 *lex* 与 *yacc* 联合使用的一个例子。

在 3.1.4 节，我们定义了如下的 *lex* 描述文件（*flex* 描述文件）：

```
%{
#include “y.tab.h”
extern int yylval;
}%
%%
0|[1-9][0-9]*          { yylval = atoi(yytext); return    INTEGER; }
[+*()\n]               { return yytext[0];}
.                      {    /*do nothing*/    }
%%
```

我们将这一 *lex* 描述文件命名为 *exp.l*。

现在，我们定义如下 *yacc* 描述文件：

```
%{
#include <stdio.h>
%}
/* 终结符 */
%token INTEGER
/* 优先级和结合性 */
%left '+'
%left '*'

%%
input    :    /* empty string */
         |    input line
         ;
line  : '\n'
         | exp '\n' { printf ("\t%d\n", $1); }
         | error '\n'
         ;
exp   : INTEGER { $$ = $1; }
         | exp '+' exp { $$ = $1 + $3; }
         | exp '*' exp { $$ = $1 * $3; }
         | '(' exp ')' { $$ = $2; }
         ;
%%
```

```
/* 用户函数 */

main () {
  yyparse ();
}

int yylex() {                /* 自行编写或由 Lex 自动生成, 在随后介绍 Lex 和 Yacc
                                的联用，需删去这里的 yylex()定义 */
}

yyerror (char *s) {
printf ("%s\n", s);
}
```

我们将这一 *yacc* 描述文件命名为 *exp.y*。

在 *Linux* 环境（假设安装了相应的开发包，并且设置了正确的环境变量，假设系统提示符为 $）中，可以通过如下步骤产生这一简单计算器的可执行文件：

```
$ lex   exp.l          /* 产生包含 yylex() 的 C 文件 lex.yy.c，其中包含 yylex() */

$ yacc   -d   exp.y    /* 产生包含 yyparse() 的 C 文件 y.tab.c，及头文件 y.tab.h */

$ cc   y.tab.c   lex.yy.c   -ly   -ll   -o exp    /* 产生可执行文件 exp，'-ly' 和 '-ll' 分别
                                                    为 yacc 和 lex 库文件的选项 */
```

可执行文件 *exp* 的执行效果如：

```
$ ./exp

$ 4+3*5

$ 19
```

3.2.3 Bison

Bison 被认为是 Yacc 的替代者，Bison 比 Yacc 更加先进，功能也更全面。Bison 的用法与 Yacc 工具相似，可以用 Yacc 的语法规则编写输入文件（*.y）。此外，为了产生 C++语言的输出文件，需在输入文件（*.y）的起始位置添加额外的说明内容：

- 可以指定输出文件名: %output "parser.cpp"
- 指定使用 C++ 模式: %skeleton "lalr1.cc"

Flex & Bison 编译方法 Lex/Yacc 类似，下面给出 Flex/Bison 联用举例：

```
flex   exp.l              /* 产生 lex.yy.c，其中包含 yylex()*/

bison -d exp.y

/* 此处缺省产生 exp.tab.c，其中包含 yyparse() 及 exp.tab.h，注意与 Yacc 默认文件名不同 */
```

```
gcc    exp.tab.c lex.yy.c -o exp -lfl
```

3.3 PLY (Python Lex/Yacc)

这一小节我们简要介绍 PLY 的基本用法，相关内容参考[8]。

PLY 是由纯 Python 代码实现的 Lex 和 Yacc。PLY 的设计目标是尽可能的沿袭传统 lex 和 yacc 工具的工作方式，包括支持 LALR(1)分析法、提供丰富的输入验证、错误报告和诊断。因此，如果你曾经在其他编程语言下使用过 Lex/Yacc，应该能够很容易的迁移到 PLY 上。

2001 年，作者在芝加哥大学教授“编译器简介”课程时开发了的早期的 PLY。学生们使用 Python 和 PLY 构建了一个类 Pascal 语言的完整编译器，编译器特性包括：词法分析、语法分析、类型检查、类型推断、嵌套作用域，并针对 SPARC 处理器生成目标代码等。自 2001 年以来，PLY 继续从用户的反馈中不断改进。

由于 PLY 是作为教学工具来开发的，你会发现它对于标记和语法规则是相当严谨的，这一定程度上是为了帮助新手用户找出常见的编程错误。不过，高级用户也会发现这有助于处理真实编程语言的复杂语法。还需要注意的是，PLY 没有提供太多花哨的东西（例如，自动构建抽象语法树和遍历树），作者也不认为它是个分析框架。相反，你会发现它是一个用 Python 实现的，基本的，但能够完全胜任的 Lex/Yacc。

3.3.1 PLY 安装与概要

安装方法一：直接在 Linux shell 终端执行“pip install ply”

安装方法二：通过 git clone 下载课程实验的 Python 框架源码，然后执行“pip install -r ./requirements.txt”

PLY 包含两个独立的模块：lex.py 和 yacc.py，都定义在 ply 包下。lex.py 模块用来将输入字符通过一系列的正则表达式分解成标记序列，yacc.py 通过一些上下文无关的文法来识别编程语言语法。yacc.py 使用 LR 解析法，并使用 LALR(1)算法（默认）或者 SLR 算法生成分析表。

这两个工具是一起工作的。lex.py 提供 token()方法作为接口，该方法每次会从输入中返回下一个有效的标记。yacc.py 将会不断调用该 token()方法来获取标记，并匹配语法规则。yacc.py 的功能通常是生成抽象语法树(AST)，不过，这完全取决于用户，如果需要，yacc.py 可以直接用来完成简单的翻译工作。

yacc.py 提供的特性包括：丰富的错误检查、语法验证、支持空产生式、错误的标记、通过优先级规则解决二义性。事实上，传统 Yacc 能够做到的 PLY 应该都可以支持。yacc.py 与 Unix 下的 yacc 的主要区别在于：yacc.py 不包含一个独立的代码生成阶段。PLY 依赖反射（reflection）机制来构建其词法分析器和语法解析器。不像传统的 lex/yacc 工具需要一个特别的输入文件，并将输入文件转化成一个独立的源文件，PLY 的输入规范本身就是合法的 Python 程序，这意味着 PLY 不会产生额外的源文件，也没有特殊的编译器构造过程。

3.3.2 一个完整的 PLY 例子

首先给出一个完整的可以运行的 PLY 应用实例，将下面的代码保存成 calc.py，并在 Linux 终端执行“python calc.py”即可打印常量表达式"1*(2+1)+3*4"的结果。完整的程序代码如下：

```
import ply.lex as lex
import ply.yacc as yacc

# 定义用到的所有 token
t_PLUS = r"\+"
t_TIMES = r"\*"
t_NEWLINE = r"\n"
t_LPAREN = r"\("
t_RPAREN = r"\)"
# 定义的同时将值转换为 int 类型
def t_INTEGER(t):
    r"\d+"
    t.value = int(t.value)
    return t

def t_error(t):
    print(t)

# 列举所有要用到的 token
tokens = (
    "PLUS",
    "TIMES",
    "INTEGER",
    "NEWLINE",
    "LPAREN",
    "RPAREN",
)
# 创建 lexer 对象
lexer = lex.lex()

# 以下为 Yacc 相关内容
# 设置算符优先级
precedence = (
    ('left', 'PLUS'),
    ('left', 'TIMES'),
)
# 此处 p[0] 类似 yacc 中的$$，而 p[1] 类似$1
def p_exp(p):
    "exp : INTEGER"
    p[0] = p[1]

def p_binary(p):
    """exp : exp PLUS exp
           | exp TIMES exp"""
```

```
        if p[2] == "+":
            p[0] = p[1] + p[3]
        else:
            p[0] = p[1] * p[3]

    def p_paren(p):
        "exp : LPAREN exp RPAREN"
        p[0] = p[2]

    def p_error(p):
        print(p)

    # 创建 parser 对象
    parser = yacc.yacc(start="exp")

    if __name__ == "__main__":
        print(parser.parse("1*(2+1)+3*4", lexer=lexer))
```

3.3.3 *PLY Lex*

3.3.3.1 标记（token）列表

词法分析器必须提供一个标记的列表，这个列表定义分析器可以输出的所有可能的标记。这些标记用于分析器的合法性验证，同时也提供给 yacc.py 用以识别终结符。在第 3.3.2 节的例子中，是这样给定标记列表的：

```
tokens = (
    "PLUS",
    "TIMES",
    "INTEGER",
    "NEWLINE",
    "LPAREN",
    "RPAREN",
)
```

3.3.3.2 标记的规则

每种标记都用一个正则表达式规则来表示，该正则表达式规则必须可以用 Python 的 re 包解析。每个规则对应的变量或函数名称都以“t_”开头来声明，表示“t_”后面定义了一个标记。对于简单的标记，可以定义成这样（其中的 r 表示使用 raw string 定义正则表达式）：

```
t_PLUS = r"\+"
```

这里，紧跟在“t_”后面的单词，必须跟标记列表中的某个标记名称对应。

如果识别标记时需要执行动作，规则可以写成一个函数。例如，下面的规则匹配整数字符串，并且将匹配的字符串转化成 Python 的整型数据：

```
def t_NUMBER(t):
```

```
    r"\d+"
    t.value = int(t.value)
    return t
```

如果使用函数的话，正则表达式规则是在函数的文档字符串中指定的。该函数总是需要接受一个 LexToken 对象作为参数，该对象有如下属性：

- t.type：以字符串表示的标记类型。默认情况下，t.type 设置为以 t_开头的变量或函数中 t_前缀后面的名字。
- t.value：标记值（匹配所得的实际的字符串）
- t.lineno：表示当前在源输入串中的行号
- t.lexpos：表示标记相对于输入串起始位置的偏移

函数执行末尾，应该返回该 LexToken 对象，否则，该标记将被丢弃。

在 PLY 内部，lex.py 用 re 包处理模式匹配。在多个标记都需要匹配时，存在匹配的优先顺序问题，用户提供的规则按照下面的顺序进行匹配：

- 所有由函数定义的标记规则，按照他们在词法分析程序中出现顺序依次匹配；
- 接下来匹配由字符串变量定义的标记规则，匹配次序为其正则表达式长度的倒序（优先匹配长表达式）

以上匹配次序的约定对于精确匹配是必要的。比如，如果你想区分‘=’和‘==’，需要确保‘==’优先匹配。如果用字符串变量来定义正则表达式，通过上面的优先匹配规则可以帮助解决这个问题。用函数定义标记，可以显式控制哪个规则优先检查，先定义先检查。

为了处理保留字，可以写一个规则函数来匹配这些标识符，并在函数里面做特殊的查询：

```
reserved = {
  'if' : 'IF',
  'then' : 'THEN',
  'else' : 'ELSE',
  'while' : 'WHILE',
  ...
}
tokens = ['LPAREN','RPAREN',...,'ID'] + list(reserved.values())
def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value,'ID')       # Check for reserved words
    return t
```

上面的例子中用到了 Python 字典数据结构，其 get 函数返回 key 所对应的 value，即所定义的标记，如果找不到 key，则返回“ID”。该例子的做法可以大大减少正则表达式的个数，并稍微加快处理速度。

注意：应该避免为每个保留字编写单独的规则，例如，如果像下面这样写：

```
t_FOR   = r'for'
t_PRINT = r'print'
```

这些规则照样也能够匹配以对应字符开头的单词，比如 forget 或者 printed，这通常不是我们想要的。

3.3.3.3 标记的值

标记被 lex 返回后，它们的值被保存在 value 属性中。正常情况下，value 保存的是匹配所得的实际文本。事实上，value 可以被赋为任何 Python 支持的类型。例如，当扫描到标识符的时候，你可能不仅需要返回标识符的名字，还需要返回其在符号表中的位置，可以像下面这样写：

```
def t_ID(t):
    ...
    # Look up symbol table information and return a tuple
    t.value = (t.value, symbol_lookup(t.value))
    ...
    return t
```

需要注意的是，不推荐用其他属性来保存值，因为 yacc.py 模块只会暴露出标记的 value 属性，访问其他属性会变得不自然。如果想保存多种属性，可以将元组、字典、或者对象实例赋给 value。

3.3.3.4 丢弃标记

想丢弃像注释之类的标记，只要不返回 value 就行了，像这样：

```
def t_COMMENT(t):
    r'\#.*'
    pass
    # No return value. Token discarded
```

为标记声明添加 ignore_ 前缀可以达到同样的目的：

```
t_ignore_COMMENT = r'\#.*'
```

如果有多种文本需要丢弃，建议使用函数来定义规则，因为函数能够提供更精确的匹配优先级控制（根据函数定义的顺序确定匹配顺序）

3.3.3.5 行号和位置信息

默认情况下，lex.py 对行号一无所知。因为 lex.py 根本不知道“行”的概念（换行符本身被视为文本的一部分）。不过，可以通过写一个特殊的规则来记录行号：

```
# Define a rule so we can track line numbers

def t_newline(t):

    r'\n+'

    t.lexer.lineno += len(t.value)
```

在这个规则中，当前 lexer 对象 t.lexer 的 lineno 属性被修改了，此后标记被简单的丢弃，因为没有任何的返回值。

lex.py 也不自动进行列跟踪。但是，位置信息被记录在每个标记对象的 lexpos 属性中，这样，就有可以计算列信息。例如：每当遇到新行的时候就重置列值：

```
# Compute column.

    #         input is the input text string

    #         token is a token instance

    def find_column(input, token):

            line_start = input.rfind('\n', 0, token.lexpos) + 1

            return (token.lexpos - line_start) + 1
```

通常，计算列的信息是为了指示上下文的错误位置，所以只在必要时有用。

3.3.3.6 忽略字符

t_ignore 规则比较特殊，是 lex.py 所保留用来忽略字符的，通常用来跳过空白或者不需要的字符。虽然可以通过定义像 t_newline()这样的规则来完成相同的事情，不过使用 t_ignore 能够提供较好的词法分析性能，因为相比普通的正则表达式，t_ignore 规则进行了特殊处理。

3.3.3.7 字面字符

字面字符可以通过在词法模块中定义一个 literals 变量来指定，例如：

```
literals = [ '+','-','*','/' ]
```

或者

```
literals = "+-*/"
```

字面字符是指单个字符，表示把字符本身作为标记，标记的 type 和 value 都是字符本身。不过，字面字符是在其他正则表达式之后被检查的。

3.3.3.8 错误处理

在词法分析中遇到非法字符时，t_error()用来处理这类错误。这种情况下，t.value 包含了余下还未被处理的输入字符串，在之前的例子中，错误处理方法是这样的：

```
# Error handling rule
```

```
def t_error(t):
    print "Illegal character '%s'" % t.value[0]
    t.lexer.skip(1)
```

在这个例子中，我们只是简单的输出不合法的字符，并且通过调用 t.lexer.skip(1)向前跳过一个字符。

3.3.3.9 构建和使用 lexer

函数 lex.lex()使用 Python 的反射机制读取调用上下文中的正则表达式，从而创建 lexer 对象：lexer = lex.lex()。创建 lexer 对象之后，可以通过如下两个接口函数使用 lexer 对象：

- lexer.input(data)：重置 lexer 并设置新的输入字符串 data；
- lexer.token()：返回下一个 LexToken 类型的标记实例，如果到达输入字符串的尾部则返回 None。

3.3.3.10 @TOKEN 装饰器

在一些应用中，可能需要定义一系列辅助的记号来构建复杂的正则表达式，例如：

```
digit            = r'([0-9])'
nondigit      = r'([_A-Za-z])'
identifier    = r'(' + nondigit + r'(' + digit + r'|' + nondigit + r')*)'
def t_ID(t):
    # want docstring to be identifier above. ?????
    ...
```

在这样的情况下，我们希望 ID 的规则引用上面已定义的标识符（identifier）。然而，使用文档字符串无法做到，为了解决这个问题，可以使用@TOKEN 装饰器：

```
from ply.lex import TOKEN

@TOKEN(identifier)
def t_ID(t):
# want docstring to be identifier above. ?????
...
```

装饰器可以将 identifier 关联到 t_ID 函数的文档字符串上，以使 lex.py 正常工作。

3.3.3.11 优化模式

为了提高性能，可能希望使用 Python 的优化模式（比如，使用-o 选项执行 Python）。然而，使用-o 选项运行时，Python 会忽略文档字符串，这是 lex.py 的特殊问题。可以通过在创建 lexer 的时候使用 optimize 选项解决该问题：

```
lexer = lex.lex(optimize=1)
```

用 Python 常规的模式运行上面的语句，lex.py 将在当前目录下创建一个名为 lextab.py 的文件，这个文件会包含所有的正则表达式规则和词法分析阶段的分析表。lextab.py 可以被导入用来构建 lexer，从而显著改善词法分析程序的启动时间，同时也可以在 Python 的优化模式下工作。

如果要更改生成的文件的名字，可以使用如下参数：

```
lexer = lex.lex(optimize=1,lextab="footab")
```

需要注意的是，在优化模式下执行时，lex 禁用大部分的错误检查。因此，建议只在确保无误准备发布最终代码时使用。

3.3.3.12 状态维护

在词法分析器中，可能想要维护一些状态。这可能包括模式设置、符号表和其他细节。例如，假设想要跟踪 NUMBER 标记出现的个数。一种方法是维护一个全局变量：

```
num_count = 0
def t_NUMBER(t):
    r'\d+'
    global num_count
    num_count += 1
    t.value = int(t.value)
    return t
```

如果你不喜欢全局变量，另一个记录信息的地方是 lexer 对象。可以通过当前标记对象的 lexer 属性来访问：

```
def t_NUMBER(t):
    r'\d+'
    t.lexer.num_count += 1         # Note use of lexer attribute
    t.value = int(t.value)
    return t
lexer = lex.lex()
lexer.num_count = 0                # Set the initial count
```

上例所采用方法的优点是当同时存在多个 lexer 实例的情况下，简单易行。然而，这看上去似乎严重违反面向对象（lexer 对象）的封装原则。但也不必过于担心，lexer 的内部属性（除了 lineno）都是以 lex 开头命名的（如 lexdata、lexpos）。因此，只要不以 lex 开头来命名自己所使用的属性就是安全的。

如果不喜欢给 lexer 对象增加属性，也可以定义一个 Lexer 类：

```
class MyLexer:
    ...
    def t_NUMBER(self,t):
        r'\d+'
        self.num_count += 1
        t.value = int(t.value)
        return t
    def build(self, **kwargs):
        self.lexer = lex.lex(object=self,**kwargs)
    def __init__(self):
        self.num_count = 0
```

如果应用将会创建很多 lexer 的实例，并且需要维护很多状态，上面定义类的方法可能是最容易管理的。此外，状态也可以用闭包来管理，比如，在 Python3 中：

```
def MyLexer():
    num_count = 0
    ...
    def t_NUMBER(t):
        r'\d+'
        nonlocal num_count
        num_count += 1
        t.value = int(t.value)
        return t
    ...
```

3.3.3.13 Lexer 克隆

如果有必要的话，lexer 对象可以通过 clone()方法来复制：

```
lexer = lex.lex()
...
newlexer = lexer.clone()
```

当 lexer 被克隆后，复制品能够精确的保留输入串和内部状态，不过，新的 lexer 可以接受一个不同的输入字符串，并独立运作起来。这在如下情形下可能有用：所编写的分析器或编译器涉及到递归或者重入处理。比如，如果需要预扫描输入字符串，可以 clone 并使用复

制品预扫描，或者假如需要实现一个预处理器，可以 clone 多个 lexer 来处理不同的输入文件。

创建克隆与重新调用 lex.lex()的不同点在于，创建克隆时 PLY 不会重新构建任何的内部分析表或者正则表达式。如果 lexer 是用类或者闭包创建的，需要注意克隆的 lexer 对象与原 lexer 共享原 lexer 的这些状态。比如：

```
m = MyLexer()
a = lex.lex(object=m)          # Create a lexer
b = a.clone()                  # Clone the lexer
```

此时，对象 a 和 b 共享对象 m，任何一个对象对 m 的修改都同时影响两个对象。因此，需要强调的是，lexer 的克隆只是意味着复用原 lexer 的正则表达式和环境来创建一个新的 lexer。如果需要创建一个全新的 lexer 对象，建议调用 lex()函数。

3.3.3.14 Lexer 的内部状态

lexer 有一些内部属性在特定情况下可能用到：

- lexer.lexpos：表示当前分析点位置的整型值。如果修改这个值，将会影响下一个 token()函数的调用行为。在标记的规则函数中，这个值表示紧跟匹配字符串后面的第一个字符的位置，如果在规则中修改这个值，则下一个返回的标记将从新的位置开始匹配；
- lexer.lineno：表示当前行号。PLY 只是声明这个属性的存在，却不会自动更新这个值。如果想要跟踪行号的话，需要自行添加代码（见第 3.3.3.5 节行号和位置信息）；
- lexer.lexdata：当前 lexer 的输入字符串，这个字符串就是 input()方法的输入字符串，更改它的值可能是个糟糕的做法，除非你自己清楚在干什么；
- lexer.lexmatch：Python 的 re.match()函数得到的当前标记的原始 Match 对象。如果正则表达式中包含分组，可以通过这个对象获得分组的值。注意：这个属性只在标记规则定义的函数中才有效。

3.2.3.14 其他问题

- Lexer 要求输入的是一个字符串。由于大多数机器都有足够的内存，这很少导致性能的问题。然而，lexer 现在还不能用来处理流数据，如文件流或者 socket 流。这主要是由 re 模块的限制导致的；
- lexer 支持用 Unicode 字符描述标记的匹配规则，也支持输入字符串包含 Unicode；
- 如果要向 re.compile()函数提供 flag 以指定匹配选项，可以使用 reflags 参数：lex.lex(reflags=re.UNICODE | re.VERBOSE)
- 由于 lexer 是全部用 Python 代码编写的，其执行性能很大程度上取决于 Python 的 re 模块，当接收大量输入文件时性能表现并不尽人意。如果担忧性能问题，可以将 Python 升级到最新版本，或者手工创建词法分析器，或者用 C 语言写 lexer 并做成扩展模块；
- 如果要创建一个手写的词法分析器并与 yacc.py 结合使用，只需满足下面的要求：

- 需要提供一个 token()函数来返回下一个标记，如果不存在可用的标记，则返回 None；
- token()方法必须返回一个 tok 对象，且 tok 对象具有 type 和 value 属性。如果需要跟踪行号，还需要定义 lineno 属性。

3.3.4 *PLY Yacc*

yacc.py 用于进行语法分析，例如，如果想要分析简单的算术表达式，应首先写下其无二义的文法：

```
expression : expression + term
           | expression - term
           | term
term       : term * factor
           | term / factor
           | factor
factor     : NUMBER
           | ( expression )
```

在上面的文法中，NUMBER,+,-,*,/等符号被称为终结符，对应原始输入。而 term，factor 等称为非终结符，它们由一系列终结符或其他规则的符号组成，用来指代语法规则。

通常采用一种称为语法制导翻译（syntax directed translation）的技术来指定某种语言的语义。在语法制导翻译中，符号及其属性出现在每个语法规则后面的语义动作中。每当一个语法（产生式）被识别，语义动作将描述需要做什么。比如，对于上面给定的文法，想要实现一个简单的计算器，应该写成下面这样：

Grammar	Action
expression0 : expression1 + term	expression0.val = expression1.val + term.val
| expression1 - term	expression0.val = expression1.val - term.val
| term	expression0.val = term.val
term0 : term1 * factor	term0.val = term1.val * factor.val
| term1 / factor	term0.val = term1.val / factor.val

| factor term0.val = factor.val

factor : NUMBER factor.val = int(NUMBER.lexval)

| (expression) factor.val = expression.val

一种理解语法指导翻译的好方法是将符号看成对象，与符号相关的值代表符号的“状态”（比如上面的 val 属性），语义动作用一组操作符号及符号值的函数或者方法来表达。Yacc 用的分析技术是著名的 LR 分析法或者叫移进-归约分析法。LR 分析法是一种自下而上的技术：首先尝试识别右部的语法规则，每当右部得到满足，相应的语义动作代码将被触发执行，当前语法规则（产生式）右边的语法符号将被替换为左边的语法符号（归约）。

3.3.4.1 ply.yacc 模块

ply.yacc 模块实现了 PLY 的语法分析功能，在第 3.3.2 节中已经给出了带有 ply.yacc 的完整例子。对于上面的简单算术表达式的语法分析例子而言，其对应代码如下：

```
# Yacc example

import ply.yacc as yacc

# Get the token map from the lexer.  This is required.

tokens = (

    'NUMBER',

    'PLUS',

    'MINUS',

    'TIMES',

    'DIVIDE',

    'LPAREN',

    'RPAREN',

 )

def p_expression_plus(p):

    'expression : expression PLUS term'

    p[0] = p[1] + p[3]

def p_expression_minus(p):

    'expression : expression MINUS term'

    p[0] = p[1] - p[3]

def p_expression_term(p):
```

```
    'expression : term'
    p[0] = p[1]
def p_term_times(p):
    'term : term TIMES factor'
    p[0] = p[1] * p[3]
def p_term_div(p):
    'term : term DIVIDE factor'
    p[0] = p[1] / p[3]
def p_term_factor(p):
    'term : factor'
    p[0] = p[1]
def p_factor_num(p):
    'factor : NUMBER'
    p[0] = p[1]
def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'
    p[0] = p[2]
# Error rule for syntax errors
def p_error(p):
    print "Syntax error in input!"
# Build the parser
parser = yacc.yacc()
while True:
    try:
        s = raw_input('calc > ')
    except EOFError:
        break
    if not s: continue
        result = parser.parse(s)
        print result
```

在这个例子中，每个语法规则被定义成一个 Python 的方法，方法的文档字符串描述了相应的上下文无关文法，方法的语句实现了对应规则的语义行为。每个方法接受一个单独的 p 参数，p 是一个包含有当前匹配语法的符号的序列，p[i]与语法符号的对应关系如下：

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    #    ^            ^          ^      ^
    #   p[0]         p[1]       p[2]   p[3]
    p[0] = p[1] + p[3]
```

其中，p[i]的值相当于词法分析模块中对 p.value 属性赋的值，对于非终结符的值，将在归约时由 p[0]的赋值决定，这里的值可以是任何类型，当然，大多数情况下只是 Python 的简单类型、元组或者类的实例。在这个例子中，我们依赖这样一个事实：NUMBER 标记的值保存的是整型值，所有规则的行为都是得到这些整型值的算术运算结果，并传递结果。

在 yacc 中定义的第一个语法规则被默认为起始规则（这个例子中的第一个出现的 expression 规则）。一旦起始规则被分析器归约，而且再无其他输入，分析器终止，最后的值将返回（这个值将是起始规则的 p[0]）。注意：也可以通过在 yacc()中使用 start 关键字参数来指定起始规则。

p_error(p)规则用于捕获语法错误。

为了构建分析器，需要调用 yacc.yacc()方法。这个方法查看整个当前模块，然后试图根据提供的文法构建 LR 分析表。第一次执行 yacc.yacc()，将得到如下输出：

```
$ python calcparse.py
Generating LALR tables
calc >
```

由于分析表的得出相对开销较大（尤其包含大量的语法的情况下），分析表被写入当前目录的一个叫 parsetab.py 的文件中。除此之外，会生成一个调试文件 parser.out。在接下来的执行过程中，yacc 直到发现文法发生变化，才会重新生成分析表和 parsetab.py 文件，否则 yacc 会从 parsetab.py 中加载分析表。注：如果有必要的话这里输出的文件名是可以改的。

如果在你的文法中有任何错误的话，yacc.py 会产生调试信息，而且可能抛出异常。一些可以被检测到的错误如下：

- 方法重复定义（在语法文件中具有相同名字的方法）；
- 二义文法产生的移进-归约和归约-归约冲突（LR 分析方法中的冲突）；
- 指定了错误的文法；
- 不可终止的递归（规则永远无法终结）；
- 未使用的规则或标记；
- 未定义的规则或标记。

这个例子的最后部分展示了如何执行由 yacc()方法创建的分析器。只需要简单的调用parse()，并将输入字符串作为参数就能运行分析器。它将运行所有的语法规则，并返回整个分析的结果，这个结果就是在起始规则中赋给 p[0]的值。

3.3.4.2 将语法规则合并

如果语法规则类似的话，可以合并到一个方法中。例如，考虑前面例子中的两个规则：

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]
def p_expression_minus(t):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]
```

比起写两个方法，可以像下面这样写在一个方法里面：

```
def p_expression(p):
    '''expression : expression PLUS term
                  | expression MINUS term'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
```

总之，方法的文档字符串可以包含多个语法规则。所以，像这样写也是合法的（尽管可能会引起困惑）：

```
def p_binary_operators(p):
    '''expression : expression PLUS term
                  | expression MINUS term
       term       : term TIMES factor
                  | term DIVIDE factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
```

```
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

如果所有的规则都有相似的结构，那么将语法规则合并才是个不错的注意（比如，产生式的项数相同）。不然，语义动作可能会变得复杂。如果产生式的项数不同，可以使用 len() 方法区分，比如：

```
def p_expressions(p):
    '''expression : expression MINUS expression
                  | MINUS expression'''
    if (len(p) == 4):
        p[0] = p[1] - p[3]
    elif (len(p) == 3):
        p[0] = -p[2]
```

如果考虑解析的性能，你应该避免像这些例子一样在一个语法规则里面用很多条件来处理。因为，每次检查当前究竟匹配的是哪个语法规则的时候，实际上重复做了分析器已经做过的事（分析器已经准确的知道哪个规则被匹配了）。为每个规则定义单独的方法，可以消除这点开销。

3.3.4.3 字面字符

如果愿意，可以在语法规则里面使用单个的字面字符，例如：

```
def p_binary_operators(p):
    '''expression : expression '+' term
                  | expression '-' term
    term          : term '*' factor
                  | term '/' factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
```

```
p[0] = p[1] / p[3]
```

字符必须像 '+' 那样使用单引号。除此之外，需要将用到的字符定义单独定义在 lex 文件的 literals 列表里：

```
# Literals.  Should be placed in module given to lex()

literals = ['+','-','*','/' ]
```

字面的字符只能是单个字符。因此，像 '<=' 或者 '==' 都是不合法的，只能使用一般的词法规则（例如 t_EQ = r'==')。

3.3.4.4 空产生式

yacc.py 可以处理空产生式，像下面这样做：

```
def p_empty(p):
    'empty :'
    pass
```

现在可以使用空匹配，只要将 'empty' 当成一个符号使用：

```
def p_optitem(p):
'optitem : item'
'        | empty'
...
```

注意：你可以将产生式保持“空”，来表示空匹配。然而，我发现用一个 'empty' 规则并用其来替代“空”，更容易表达意图，并有较好的可读性。

3.3.4.5 改变起始符号

默认情况下，在 yacc 中的第一条规则是起始语法规则（顶层规则）。可以用 start 标识来改变这种行为：

```
start = 'foo'

def p_bar(p):
    'bar : A B'

# This is the starting rule due to the start specifier above

def p_foo(p):
    'foo : bar X'

...
```

用 start 标识有助于在调试的时候将大型的语法规则分成小部分来分析。也可把 start 符号作为 yacc 的参数：

```
yacc.yacc(start='foo')
```

参考文献

1. Compilers, http://www.stanford.edu/class/cs143/ , CS143, Stanford University.
2. Programming Languages and Compilers ，http://inst.eecs.berkeley.edu/~cs164/archives.html ，CS143, University of California at Berkeley.
3. Flex, https://github.com/westes/flex/releases/ .
4.Bison, http://ftp.gnu.org/gnu/bison/ .
5. https://www.dabeaz.com/ply/ply.html
6. https://www.qemu.org/
7. https://github.com/riscv-software-src/riscv-isa-sim
8. PLY (Python Lex-Yacc): https://www.dabeaz.com/ply/ply.html#ply_nn1
   中文翻译版本：https://www.jianshu.com/p/0eaeba15ee68
9. Winskel, Glynn . The Formal Semantics of Programming Languages: An Introduction. MIT Press, 1993.
10. Robert Harper, Practical Foundations for Programming Languages. Cambridge University Press, 2012. 第 2 版，2016.（一些草版或缩减版可自行从网上下载，使用时注意版权的限定）

课后作业

1. 阅读有关 Lex & Yacc 的技术文档。
2. 掌握 Flex & Bison 的使用。

附录 A 历年来参与 Decaf/Mind 编译实验项目维护的助教

杨俊峰（Stanford 助教）
张迎辉（计 99-计 00 助教）
高崇南（计 00 助教）
毛雁华（计 00-计 01 助教，X86 后端）
刘天淼（计 01 助教）
唐 硕 （计 02 助教，TOOL）
梁英毅（计 03-计 05 助教，Java 版，C 版 Mind）
张 铎 （计 05-计 07 助教，改 Mind 至 Java 版）
曹 震 （计 08 助教）
李 叠 （计 09-10 助教）
蒋挺宇（计 10-13 助教）

许建林（计 12 助教）
王 耀 （计 13-14 助教）
刘 昊 （计 13-14 助教）
尚 书 （计 13-14 助教）
甄艳洁（计 15-16 助教）
沈游人（计 15-16 助教）
朱俸民（计 15-17 助教）
冀伟清（计 15-17 助教）
戴臻旸（计 16-17 助教）
贾越凯（计 16-17 助教）
寇明阳（计 16-17 助教）
李晨昊（计 16-17 助教）

此外，还有一些编译原理课程的助教，仅负责 PL0 实验，如龚珩、王曦、陈磊、李鹏，等等，这里不一一列举。

可能有疏漏，如有同学提供信息，将随时补遗。

参与 MiniDecaf 编译实验项目维护的助教

冀伟清（计 18 助教）
戴臻旸（计 18 助教）
贾越凯（计 18 助教）
寇明阳（计 18 助教）
李晨昊（计 18 助教）
张译仁（计 18 助教）
曾 军（计 18-19 助教）
唐适之（计 18-19 助教）
谢兴宇（计 18-19 助教）
刘润达（计 19 助教）
周 智（计 19 助教）
陈之杨（计 19 助教）
杨耀良（计 19 助教）

附录 B 最近几年部分有代表性的拓展实验选题（学生自选）

1. 拓展 Decaf 实现对 Lambda 表达式的支持（计 34，马坚鑫）
2. 添加 Java8 Lambda 表达式至 Decaf 语言（计科 30，陈晓奇）
3. Tac 模拟器层面实现垃圾回收机制（计科 30，陈立杰，王若松，占玮）
4. Tac 代码的 Decaf 源级调试器（电子系无 37，林梓楠）

5. 为 decaf 增加类似于 java 的 interface 机制（计科 30，彭天翼，董方宏，李志远）
6. 为现有 Decaf 语言框架添加 SSA 中间表示（计科 00，吴佳俊）
7. 初步实现 Decaf 实验框架的一个 Rust 语言版本（计 72，李晨昊）