实验导引（一）

2016-09-21

1. Decaf/Mind 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。
- 在计 50 班（姚班）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，我们称之为 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 05 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。

1.2 Decaf/Mind 项目框架的总体结构

Decaf/Mind 项目的实验框架是设计实现 Decaf/Mind 语言的编译器，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（A）：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）。使

用的 Lex 和 Yacc 版本分别为 Flex [3] 和 BYACC/J [4]。

阶段一（B）：手工编写词法和语法分析程序。手工编写自上而下语法分析/翻译程序，替代阶段一（A）中生成的 yylex 和 yyparse 等函数，可以实现与阶段一（A）同样的任务（一遍扫描源程序后直接产生实验指定的抽象语法树）。

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 进行两趟扫描：第一趟扫描的时候建立符号表的信息，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 A 继承于 B，但是 B 没有定义的情况）；第二趟扫描的时候检查所有的语句以及表达式的参数的数据类型。

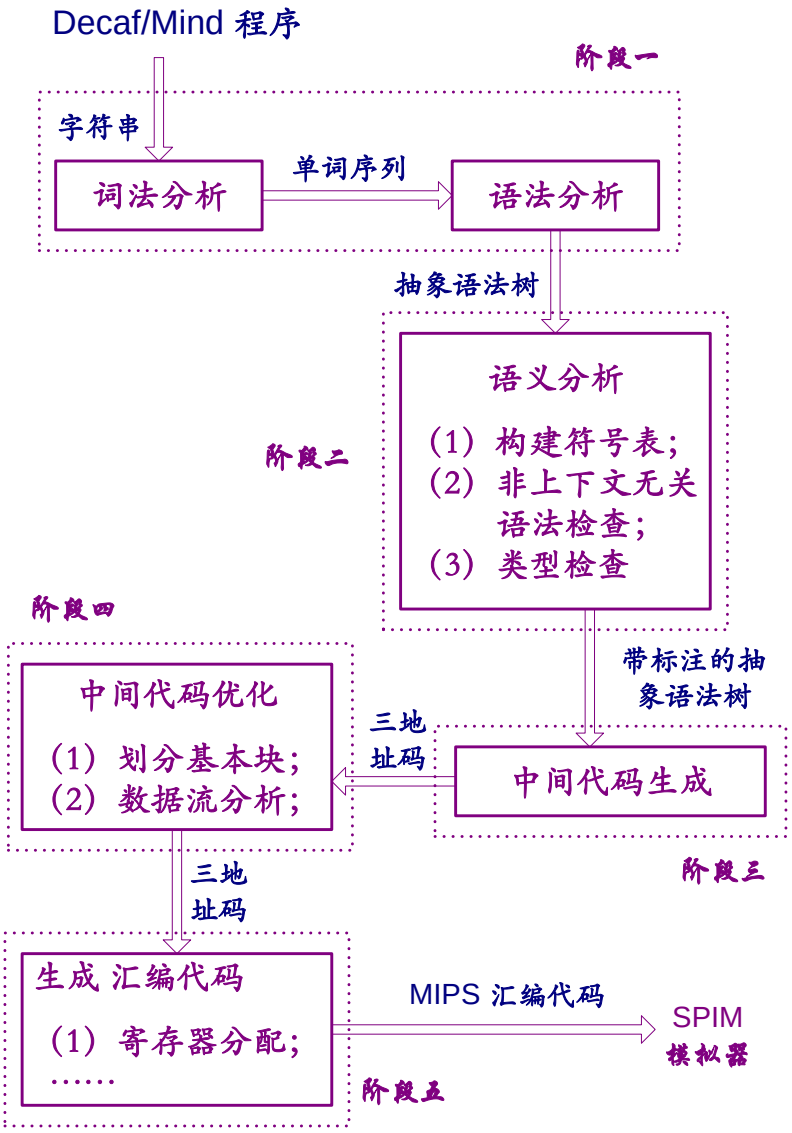

图 1　Decaf/Mind 编译器总体结构

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的内容。

阶段三完成后，三地址码程序可在 TAC 模拟器上执行。

阶段四：中间代码优化。根据教学计划，目前的实验只要求选做，基于 TAC 实现简单的数据流分析（活跃变数据流、到达-定值数据流、UD 链和 DU 链等）或者简单的优化（如常量合并、常量传播、公共子表达式消除、等等）。

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，实验内容可以设计为对这些部分进行改进。考虑到学生负担问题，在之前的“编译原理”课程中，我们没有安排过这一阶段的实验任务。后续可选的“计算机系统综合实验”课程涉及到这一阶段的内容。

完成这些阶段以后，即可产生出适合实际 MIPS 机器上的汇编代码，可以利用由美国 Wisconsin 大学所开发的 MIPS R2000/R3000 模拟器 SPIM [5] 来运行这些汇编代码。

1.3 本学期主体实验内容

本学期的主体实验将基于 Decaf/Mind 项目框架完成。该框架是针对 Decaf 语言（参见《Decaf 语言规范》中的描述）的实现。主体实验的要求是对 decaf 语言增加新的语言特性。

增加的语言特性如下：

1. 三元运算符：实现三操作数表达式，形如 A ? B : C。其语义解释与 C 语言的条件表达式一致。

2. 一种特殊的二元对象运算，形如 A << B：其语义解释为：A 和 B 为对象，A << B 计算结果返回另外一个对象，其所属类为 A 和 B 所属类的最小公共父类（基于继承层次关系），该对象的成员变量取值为 A 中相应成员变量的取值。假定继承层次关系为自反传递关系，所以 A << B 的所属类也可能是 A 或 B 的所属类。

3. switch-case 语句：实现 switch-case 控制结构，与 C 语言相同，形如：

```
switch(表达式) {
    case 常量表达式1: 语句块1;
    case 常量表达式2: 语句块2;
    …
    case 常量表达式n: 语句块n;
    default: 语句块n+1;
}
```

4. repeat-until 循环结构：实现 repeat 循环结构，形如：

```
repeat{
    语句序列
}until（布尔表达式）;
```

5. 循环内部控制语句 continue：其语义是跳过本次循环的后续语句，使控制转移到下一次循环。

这些语言特性的更精确描述，参见各阶段实验的说明文档。

前三个阶段的实验内容对应于实验框架（图 1）中的三个阶段，其中第一阶段分 A 和 B 两部分（分别通过工具和手工方式实现词法、语法分析，并构造抽象语法树）。阶段一（A）实验内容的一个简短描述参见第 4 节。本学期拟将阶段一（B）实验作为辅助实验。

第四个阶段将布置一个是可选的实验，完成简单的数据流分析或优化，具体内容待定。

本课程不安排与框架（图 1）中第五个阶段相对应的实验，“计算机系统综合实验”课程将会涉及到相关内容，届时欢迎选修。

2. 辅助实验（ 基于 Decaf/Mind 项目或 PL0 项目）

本学期实验内容将包括一个辅助性实验，主要训练自顶向下语法分析器的构造（手工实现），本学期主体实验中未能涵盖这一重要内容。辅助性实验 Decaf/Mind 项目或者基于本课程早期使用的 PL0 项目框架完成。

辅助实验的具体内容及实验框架将随课程的进程发布给大家。

3. 拓展实验

对自行扩展部分的评价是综合考虑创新性、实用性、合理性、难度、工作量等因素进行的。我们希望自行扩展部分的选题最好是在已有实验框架基础上有意义的改进工作，最好与教师或助教沟通后再确定扩展部分选题。提交时需要有详尽的设计文档，通常会要求成果演示和集中答辩。

拓展实验可能选择：

- 在 Decaf / Mind 实验框架基础上有意义的改进工作。

  如：垃圾回收机制，例外处理机制，多继承机制，某些函数式语言特征，SSA 中间表示，浮点数支持，后端重定向，调试器，等等

- 学生自主设计的课程实验方案。最好预先提交方案，接受难度、工作量和价值方面的评估。

- 与本学期“计算机原理”课程中有关“计算机系统综合实验”的内容协同要求。

4. 阶段一（A）实验

在每一阶段实验开始前时，将会为大家提供必要的实验文档和基础框架，可以帮助大家较详细地了解实验要求和实验内容。这一讲，我们先对阶段一实验的内容作简短的描述。

4.1 词法分析

词法分析的功能是从左到右扫描 Decaf/Mind 源程序（今后均称为 Decaf 源程序或 Decaf 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。在识别出下一单词，词法分析程序就会产生一个**单词记录**（包括单词类别，单词值，及源程序中位置等信息），传递给后续阶段使用。在识别的过程中，我们还需要检测词法相关的错误，例如字符@并非 Decaf 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 Decaf 程序片断 P1 ：

```
class Main {
    static void main() {
        Print("hello world");
    }
}
```

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，这些单词符号即对应语法规则中的终结符。

单词类别	单词值
保留字 **class**	
标识符	Main
分隔符 **{**	
保留字 **static**	
保留字 **void**	
标识符	main
分隔符 **(**	
分隔符 **)**	
分隔符 **{**	
保留字 **Print**	
分隔符 **(**	
字符串常量	“hello world”
分隔符 **)**	
分隔符 **;**	
分隔符 **}**	
分隔符 **}**	

表 1　识别单词（得到单词类别/单词值等信息）

4.2 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 Decaf 程序的过程。对于不符合语法规则的 Decaf 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。

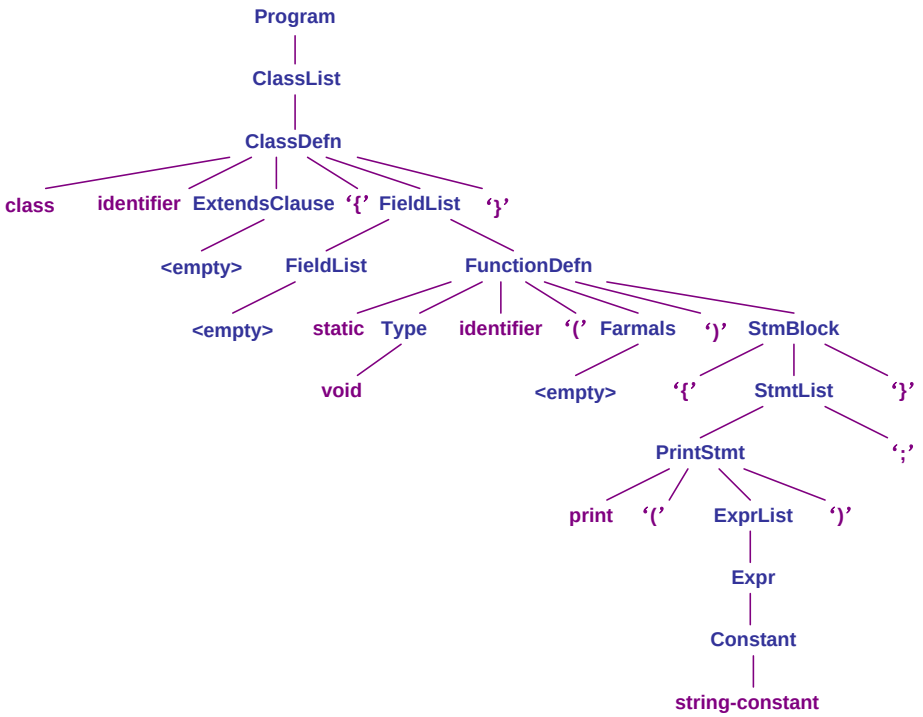

图 2　语法分析树（具体语法树）

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（片断）为G[Program]：

Program → ClassList
ClassList → ClassList ClassDefn | ClassDefn
VariableDecl → Variable ';'
Variable → Type identifier
Type → int | void | … | class identifier | …
ClassDefn → class identifier ExtendsClause '{' FieldList '}'
ExtendsClause → extends identifier | <empty>
FieldList → FieldList VariableDecl | FieldList FunctionDefn | <empty>
Formals → VariableList | <empty>
VariableList → VariableList ',' Variable | Variable
FunctionDefn → static Type identifier '(' Formals ')' StmtBlock
| Type identifier '(' Formals ')' StmtBlock
StmtBlock → '{' StmtList '}'
StmtList → StmtList Stmt | <empty>
Stmt → VariableDecl | IfStmt | WhileStmt | … | PrintStmt ';' | StmtBlock
Expr → Constant | Expr '+' Expr | …
Constant → int_const | string_const| …
ExprList → ExprList ',' Expr | Expr
WhileStmt → while '(' Expr ')' Stmt
PrintStmt → print '(' ExprList ')'

……

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），<empty> 对应 ε-产生式。

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，对于上述程序片断 P1，得到如图 2 所示的语法分析树。

4.3 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），是一种只跟我们关心的内容有关的语法树表示形式。抽象语法树的特点是：（1）不含我们不关心的终结符，例如逗号等（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，这样更便于后续处理；（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，后续过程可以反复利用。

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，而不方便后续遍的使用。

例如，图 3 是上述程序片断 P1 的一种抽象语法树表示形式。

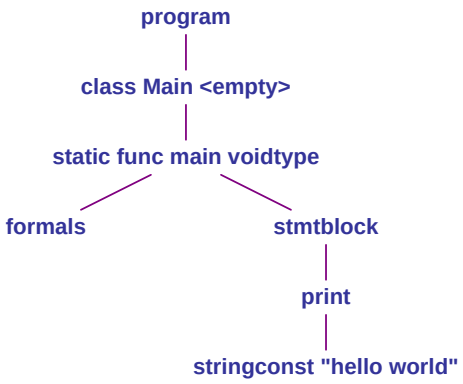

图 3　抽象语法树

实验代码框架中定义好了我们要用到的各种抽象语法树结点对应的数据结构，见 decaf.tree 包中的类。在第一阶段动手实现之前应熟悉这些数据结构。

4.3 阶段一（A）实验内容

阶段一（A）实验的重点是掌握 Lex 和 Yacc 的用法，体会使用编译器自动构造工具的好处，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 的核心是利用正规表达式给出词法规则，而利用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 EBNF 的扩展形式给出的。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，其要领在实验文档中有所提示（部分可参考 4.2 节所列举的上下文无关文法片断）。另外，还要注意的是，Lex 中的正规表达式形式上要比我们在“形式语言与自动机”课程中学习的形式（只含三种运算）丰富得多。

在第 5 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，我们要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，大家只需要完善实验框架中的部分内容。然而，建议大家在开始前先读懂这个框架。特别要关注 AST 的数据结构以及 Lex 和 Yacc 的联用等。

5. 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。

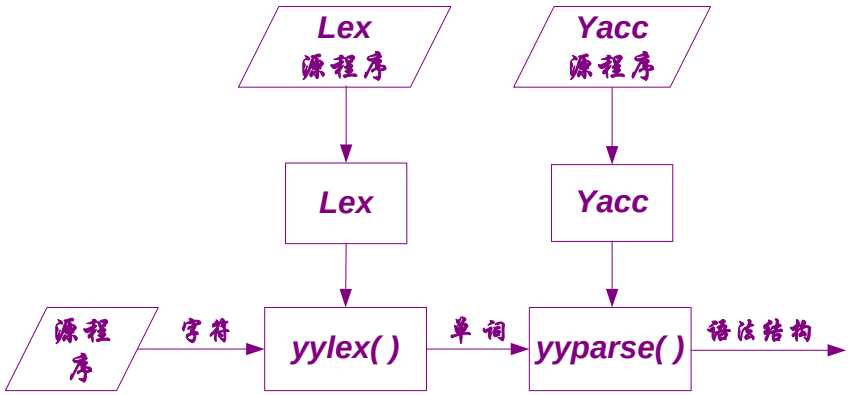

图 4　Lex 和 Yacc 工具的使用

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。

5.1 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。

5.1.1 *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。

5.1.2 *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 5.1.1。

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 5.1.1。“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。

5.1.3 *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')
%%
```

5.1.4 与 Yacc 的接口约定

Lex 的一个主要应用方面是与 *yacc*（参见 5.2 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*/  }
%%
```

5.2 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。

5.2.1 *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。

5.2.1.1 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;
%}
```

5.2.1.2 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
```

5.2.1.3 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。

5.2.1.4 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。

5.2.2 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 5.1.4 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
```

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. Jflex, http://jflex.de/ .
4. BYACC/J, http://byaccj.sourceforge.net/ .

5. MIPS SPIM, University of Wisconsin-Madison, http://pages.cs.wisc.edu/~larus/spim.html

课后作业

1. 阅读有关 Lex & Yacc 的技术文档。
2. 掌握 Jflex & BYACC/J 的使用。