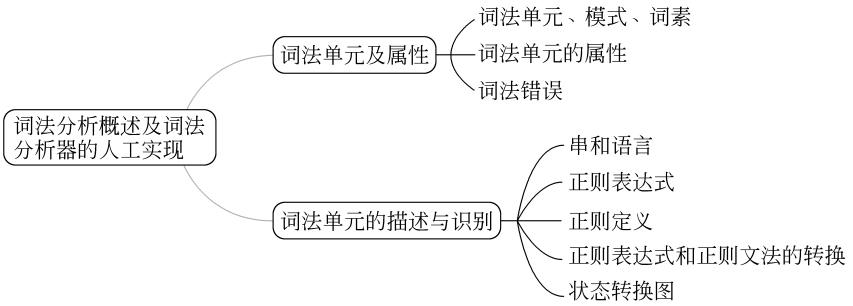
第 3 章　词法分析概述及词法分析器的人工实现



词法分析是编译的第一阶段。词法分析器的主要任务是读入源程序的输入字符，将它们组成词素，生成并输出一个词法单元序列，使每个词法单元对应于一个词素。这个词法单元序列被输出到语法分析器进行语法分析。词法分析器通常还要和符号表进行交互。当词法分析器发现了一个标识符的词素时，它要将这个词素添加到符号表中。在某些情况下，词法分析器会从符号表中读取有关标识符种类的信息，以确定向语法分析器传送哪个词法单元。

本章主要讨论如何构建一个词法分析器。如果要人工实现词法分析器，首先要建立每个词法单元的词法结构图或其他描述。然后，可以编写代码识别输入中出现的每个词素，并返回已经识别的词法单元的有关信息。而正则表达式就是一种可以方便地描述词素模式的方法，本章会具体介绍它的定义。而作为构造词法分析器的一个中间步骤，本章还会将正则表达式表示的模式转换成具有特定风格的流图，称为状态转换图。

3.1　词法单元及属性

3.1.1　词法单元、模式、词素

在讨论词法分析时，常使用 3 个相关但有区别的术语：词法单元、模式和词素。

词法单元由一个词法单元名和一个可选的属性值组成。词法单元名是一个表示某种词法单位的抽象符号，例如一个特定的关键字，或者代表一个标识符的输入字符序列。词法单元名是由语法分析器处理的输入符号。在后面的内容中，通常使用黑体字给出词法单元名，并使用词法单元名引用一个词法单元。

模式描述了一个词法单元的词素可能具有的形式。当词法单元是一个关键字时，它的模式就是组成这个关键字的字符序列。对于标识符和其他词法单元，模式是一个更加复杂的结构，它可以和很多符号串匹配。例如，在 C 语言中，标识符的模式可以定义为以字母或下画线开头，后面跟若干字母、数字或下画线的字符串；整数常量的模式可以定义为由一个

或多个数字组成的字符串。

词素是源程序中的一个字符序列，它和某个词法单元的模式匹配，并被词法分析器识别为该词法单元的一个实例。例如，在C语言中，源程序中的字符串int可以被识别为一个关键字词法单元，字符串int就是对应的词素。

例3.1 表3.1给出了常见的词法单元、非正式描述和词素示例。下面说明上述概念在实际中是如何应用的。在C语言printf("Total=%d\n",score)中，printf和score都是和词法单元id的模式匹配的词素，而"Total=%d\n"则是一个和literal匹配的词素。

表3.1 常见的词法单元、非正式描述和词素示例

词法单元	非正式描述	词素示例
if	字符i、f	if
else	字符e、l、s、e	else
comparison	<或>或<=或>=或==或!=	<=,!=
id	字母开头的字母数字串	pi,score,D2
number	任何数字常量	3.1415,0,6.02
literal	在两个"之间，除"以外的任何字符	"Hello world"

3.1.2 词法单元的属性

如果有多个词素可以和一个模式匹配，那么词法分析器必须向编译器的后续阶段提供有关被匹配词素的附加信息。例如，0和1都能和词法单元number的模式匹配，但是对于代码生成器而言，至关重要的是知道在源程序中找到了哪个词素。又如，在C语言中，数字常量可以是十进制数、十六进制数或八进制数。因此，当词法分析器遇到一个数字时，需要确定它是哪种类型的数字，以便后续阶段可以正确地处理它。因此，在很多情况下，词法分析器不仅向语法分析器返回一个词法单元名，而且会返回一个描述该词法单元的词素的属性值。词法单元名将影响语法分析过程中的决定，而这个属性则会影响语法分析之后对这个词法单元的翻译。

假设一个词法单元至多有一个相关的属性值，当然这个属性值可能是一个组合了多种信息的结构化数据。一般来说，和一个标识符有关的信息，例如它的词素、类型、第一次出现的位置(在发出一个有关该标识符的错误消息时需要使用这个信息)，都保存在符号表中。因此，一个标识符的属性值是一个指向符号表中该标识符对应条目的指针。

例3.2 FORTRAN语句

E=M * C**2

中的词法单元名和相关的属性值可写成如下的名字-属性对序列：

<id,指向符号表中E的条目的指针>

<assign_on>

<id,指向符号表中M的条目的指针>

<mult_op>

<id,指向符号表中C的条目的指针>

<exp_op>

<number,整数值 2>

注意：在某些名字-属性对中，特别是运算符、标点符号和关键字的名字-属性对中，不需要属性值。在本例中，词法单元 number 有一个整数属性值；而在实践中，编译器将保存一个代表该常量的字符串，并将一个指向该字符串的指针作为 number 的属性值。

3.1.3 词法错误

如果没有其他组件的帮助，词法分析器很难发现源代码中的错误。例如，当词法分析器在 C 程序片段

fi(a==f(x))…

中第一次遇到 fi 时，它无法指出 fi 究竟是关键字 if 的误写还是一个未声明的函数标识符。由于 fi 是标识符 id 的一个合法词素，因此词法分析器必须向语法分析器返回这个 id 词法单元，而让编译器的另一个阶段(在这个例子里是语法分析器)去处理这个因为字母颠倒而引起的错误。

然而，假设出现所有词法单元的模式都无法和剩余输入的某个前缀相匹配的情况，此时词法分析器就不能继续处理输入。此时，最简单的错误恢复策略是“恐慌模式”恢复。即，从剩余的输入中不断删除字符，直到词法分析器能够在剩余输入的开头发现一个正确的词法单元为止。这个恢复技术可能会给语法分析器带来混乱。但是在交互计算环境中，这个技术已经足够了。

可能采取的其他错误恢复动作如下：

(1) 从剩余的输入中删除一个字符。

(2) 向剩余的输入中插入一个遗漏的字符。

(3) 用一个字符替换另一个字符。

(4) 交换两个相邻的字符。

这些变换可以在试图修复错误输入时进行。最简单的策略是看一下是否可以通过一次变换将剩余输入的某个前缀变成一个合法的词素。这种策略还是有道理的，因为在实践中，大多数词法错误只涉及一个字符。另外一种更加通用的改正策略是计算出最少需要多少次变换才能够把一个源程序转换成为一个只包含合法词素的程序。但是在实践中发现这种方法的代价太大，不值得使用。

3.2 输入缓冲

在讨论词法分析器如何识别输入流中的词素之前，应该先讨论几种可以加速源程序读入的方法。源程序读入虽然看似简单，但实际上对编译器的性能和效率至关重要。由于词法分析器常常需要进行超前搜索，查看一个词素之后的若干字符才能够确定是否找到了正确的词素，而一旦发现超前搜索得到的字符不属于当前的词素，则会产生回退的操作，因此源程序读入时的效率直接影响整个编译器的速度。

在实践中，为了确定是否到达了标识符的末尾，通常需要至少向前查看一个字符。例如，只有读取到一个非字母或数字的字符之后才能确定已经到达一个标识符的末尾，因此这

个字符不是 id 的词素的一部分。此外，在 C 语言中，像－、＝或<这样的单字符运算符也有可能是->、＝＝或<＝这样的双字符运算符的开始字符。因此，为了安全地处理向前查看多个符号的问题，本章将介绍一种双缓冲区方案。这种方案可以在不丢失输入信息的情况下安全地处理向前查看多个符号的问题。同时还可以使用哨兵标记节约用于检查缓冲区末端的时间。

假设源程序存储在磁盘上，每读取一个字符就需要访问一次磁盘，尽管操作系统能够在一定程度上降低这种开销，但这依然会导致效率降低。因此人们开发出一些特殊的缓冲技术以减少这种影响，也就是设置适当的缓冲区(buffer)。如图 3.1 所示，词法分析器首先按照缓冲区的大小将一部分源程序预先读入缓冲区，这个缓冲区被称为输入缓冲区。当需要读取下一个字符时，可以直接从缓冲区中完成读取。直到缓冲区中的所有字符都已经被识别过程处理完毕，再一次性地从磁盘读入下一段源程序的字符流。这样就可以避免频繁地访问磁盘，提高读入效率。

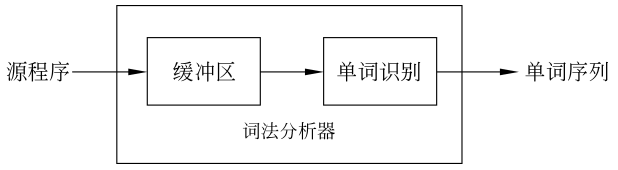


图 3.1　词法分析器

总的来说，缓冲技术是加速源程序读入的重要手段之一，可以通过设置适当的缓冲区减少访问磁盘的次数，提高读入效率。同时，也需要考虑如何处理向前查看多个符号的问题，可以使用双缓冲区方案解决这个问题。此外，还可以采用哨兵标记等方法优化词法分析器的设计和实现。但需要注意的是，虽然优化词法分析器的性能和效率是很重要的，但也不能忽视其正确性和稳定性，应该进行充分的测试和验证。

3.2.1　缓冲区对

由于在编译一个大型源程序时需要处理大量的字符，处理这些字符需要很多的时间，因此开发了一些特殊的缓冲技术以减少用于处理单个输入字符的时间开销。一种重要的机制就是利用一对交替读入的缓冲区，如图 3.2 所示。

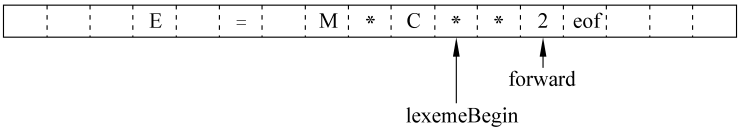


图 3.2　一对输入缓冲区

每个缓冲区的容量都是 N 个字符，通常 N 是一个磁盘块的大小，如 4096B。可以使用系统读取命令一次将 N 个字符读入缓冲区中，而不是每读入一个字符调用一次系统读取命令。如果输入文件中的剩余字符不足 N 个，那么就会有一个特殊字符(用 eof 表示)来标记源文件的结束。这个特殊字符不同于任何可能出现在源程序中的字符。

程序为输入维护了两个指针：

(1) lexemeBegin 指针。该指针指向当前词素的开始处。当前正试图确定这个词素的结尾。

(2) forward 指针。它一直向前扫描,直到发现某个模式被匹配为止。做出这个决定所依据的策略将在本章后面讨论。

一旦确定了下一个词素,forward 指针将指向该词素结尾的字符。词法分析器将这个词素作为某个返回给语法分析器的词法单元的属性值记录下来。然后使 lexemeBegin 指针指向刚刚找到的词素之后的第一个字符。在图 3.2 中可以看到,forward 指针已经越过下一个词素**(FORTRAN 的指数运算符)。在处理完这个词素后,它将会被左移一个位置。

将 forward 指针前移要求首先检查是否已经到达某个缓冲区的末尾。如果是,则必须将 N 个新字符读到另一个缓冲区中,且将 forward 指针指向这个新载入字符的缓冲区的头部。只要从不需要越过实际的词素向前看很远,以至于这个词素的长度加上向前看的距离大于 N,就不会在识别这个词素之前覆盖这个尚在缓冲区中的词素。

3.2.2 哨兵标记

如果采用 3.2.1 节中描述的方案,那么在每次向前移动 forward 指针时都必须检查是否到达了缓冲区的末尾。若是,那么必须加载另一个缓冲区。因此,每读入一个字符,需要做两次测试,一次是检查是否到达缓冲区的末尾,另一次是确定读入的字符是什么(后者可能是一个多路分支选择语句)。如果扩展每个缓冲区,使它们在末尾包含一个哨兵(sentinel)标记,就可以把对缓冲区末尾的测试和对当前字符的测试合二为一。这个哨兵字符必须是一个不会在源程序中出现的特殊字符,一个自然的选择就是字符 eof。

图 3.3 显示的缓冲区安排与图 3.2 一致,只是加入了哨兵标记。请注意,eof 仍然可以用来标记整个输入的结尾。任何不是出现在某个缓冲区末尾的 eof 都表示到达了输入的结尾。图 3.4 给出了带有哨兵标记的 forward 指针移动算法。请注意,在大部分情况下只需要进行一次测试就可以根据 forward 所指向的字符完成多路分支跳转。只有当确实处于缓冲区末尾或输入的结尾时,才需要进行更多的测试。

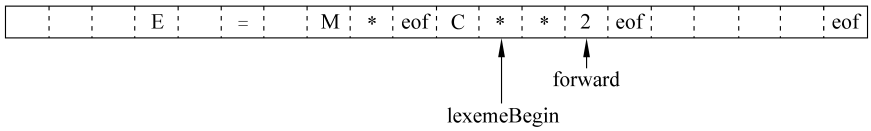


图 3.3 各个缓冲区末尾的哨兵标记

```
Switch(*forward ++){
        case eof:
             if(forward在第一个缓冲区末尾){
                          装载第二个缓冲区;
                         forward=第二个缓冲区的开头;
             }
             else if(forward在第二个缓冲区末尾){
                          装载第一个缓冲区;
                          forward=第一个缓冲区的开头;
             }
             else /*缓冲区内部的eof标记输入结束*/
                          终止词法分析
             break;
        其他字符的情况
}
```

图 3.4 带有哨兵标记的 forward 指针移动算法

3.3 词法单元的描述与识别

3.3.1 串和语言

在词法分析中，字母表(alphabet)和串是非常重要的概念。字母表是编译器所处理的符号集合，符号的典型例子包括字母、数字和标点符号。在计算机科学中，经常使用二进制字母表，即由两个符号 0 和 1 组成的字母表，用于表示数字和计算机指令等信息。此外，ASCII 和 Unicode 等字母表也被广泛使用，用于表示字符和文本等信息。字母表的大小通常被称为字母表的基数(radix)。在实践中，字母表的大小通常是 2 的幂，这样可以方便地进行位操作和处理二进制数据。

某个字母表上的一个串(string)是该字母表中符号的一个有穷序列，它是编译器中的重要概念。在编译器中，输入的源代码就是一个串，它由各种符号组成。在词法分析阶段，词法分析器会将输入的源代码串分割成一个个词法单元，这些词法单元代表了源代码中的各种语言结构，如关键字、标识符、运算符等。在语言理论中，术语“句子”和“字”常常被当作“串”的同义词。串 s 的长度是指该串中符号的个数，通常记作 $|s|$，例如，banana 是一个长度为 6 的串。空串(empty string)是指长度为 0 的串，通常用希腊字母 ε 表示。空串在编译器中也是非常重要的，它可以表示语法规则中可选的部分或空产生式。

在处理串时，可以使用各种数据结构和算法。例如，串匹配算法可以用来判断一个串是否包含另一个子串，这在编译器中经常用来判断标识符是否被正确定义。另外，正则表达式和有限状态自动机等形式化工具也可以用来描述和处理串。

表 3.2 给出了与串相关的常用术语。

表 3.2 与串相关的常用术语

术　语	说　明
串 s 的前缀(prefix)	从 s 的尾部删除 0 个或多个符号后得到的串。例如，ban、banana 和 ε 是 banana 的前缀
串 s 的后缀(suffix)	从 s 的开始处删除 0 个或多个符号后得到的串。例如，nana、banana 和 ε 是 banana 的后缀
串 s 的子串(substring)	删除 s 的某个前缀和某个后缀之后得到的串。例如，bnana、nan 和 ε 是 banana 的子串
串 s 的真前缀、真后缀、真子串	分别是 s 的既不等于 ε 也不等于 s 本身的前缀、后缀和子串
串 s 的子序列(subsequence)	从 s 中删除 0 个或多个符号后得到的串，这些被删除的符号可能不相邻。例如，baan 是 banana 的一个子序列

在词法分析中，语言是一个重要的概念，因为编译器需要识别输入的源代码是否符合某个给定的语言规范。如果源代码不符合语言规范，编译器需要给出错误提示，帮助程序员及时修复错误。因此，理解和掌握语言相关的概念对于编译器的设计和实现都是至关重要的。

一个语言是由某个给定字母表上的一个任意的可数的串集合组成的。这个定义非常宽泛，根据这个定义，空集 $\varnothing$ 和仅包含空串的集合 $\{\varepsilon\}$ 都是语言。所有语法正确的 C 程序的集合以及所有语法正确的英语句子的集合也都是语言，虽然后两种语言难以精确地描述。注

意,这个定义并没有要求语言中的串一定具有某种含义。例如,由 a 和 b 组成的所有字符串的集合是一个语言,但它们并没有明确的含义。而英语句子的集合虽然有明确的含义,但是它们很难用形式化的方式描述。

如果 x 和 y 是串,那么 x 和 y 的连接(concatenation)(记作 xy)是把 y 附加到 x 后面形成的串。例如,如果 $x=$dog 且 $y=$house,那么 $xy=$doghouse。空串是连接运算的单位元,也就是说,对于任何串 s 都有 $s\varepsilon=\varepsilon s=s$。

如果把两个串的连接看成这两个串的乘积,可以定义串的指数运算如下:定义 s^0 为 ε,并且对于 $i>0$,s^i 为 $s^{i-1}s$。因为 $\varepsilon s=s$,由此可知 $s^1=s$,$s^2=ss$,$s^3=sss$,依此类推。

在词法分析中,最重要的语言上的运算是并、连接和闭包运算。表 3.3 给出了这些运算的正式定义。并运算是常见的集合运算。语言就是以各种可能的方式从第一个语言中任取一个串,再从第二个语言中任取一个串,然后将它们连接后得到的所有串的集合。一个语言 L 的 Kleene 闭包(closure)记为 L^*,就是将 L 连接 0 次或多次后得到的串集。注意,L^0,即将 L 连接 0 次得到的集合,被定义为$\{\varepsilon\}$,并且 L 被归纳地定义为 $L^{i-1}L$。最后,L 的正闭包(记为 L^+)和 Kleene 闭包基本相同,但是不包含 L。也就是说,除非 ε 属于 L,否则 ε 不属于 L^+。

表 3.3 语言上的运算的定义

运　　算	定义和表示
L 和 M 的并	$L\cup M=\{s \mid s$ 属于 L 或者 s 属于 $M\}$
L 和 M 的连接	$LM=\{st \mid s$ 属于 L 且 t 属于 $M\}$
L 的 Kleene 闭包	$L^*=\bigcup_{i=0}^{\infty} L^i$
L 的正闭包	$L^+=\bigcup_{i=1}^{\infty} L^i$

例 3.3 令 L 表示字母的集合{A,B,…,Z,a,b,…,z},令 D 表示数码的集合{0,1,…,9}。可以用两种不同但等价的方式考虑 L 和 D:一种方式是将 L 看成由大小写字母组成的字母表,将 D 看成由 10 个数码组成的字母表;另一种方式是将 L 和 D 看成语言,它们的所有串的长度都为 1。下面是一些根据表 3.2 中的运算符用 L 和 D 构造得到的新语言:

(1) $L\cup D$ 是字母和数码的集合——严格地讲,这个语言包含 62 个长度为 1 的串,每个串是一个字母或一个数码。

(2) LD 是包含 520 个长度为 2 的串的集合,每个串都由一个字母和一个数码组成。

(3) L^4 是所有由 4 个字母构成的串的集合。

(4) L^* 是所有由字母构成的串的集合,包括空串 E。

(5) $L(L\cup D)^*$ 是所有以字母开头的、由字母和数码组成的串的集合。

(6) D^+ 是由一个或多个数码构成的串的集合。

3.3.2 正则表达式

假设要描述 C 语言的所有合法标识符的集合。它差不多就是例 3.3 的第 5 项所定义的语言,唯一的不同是 C 语言的标识符中可以包括下画线。

在例 3.3 中,可以首先给出字母和数码集合的名字,然后使用并、连接和闭包等运算符

描述标识符，这种处理方法非常有用。因此，人们常常使用一种称为正则表达式（也可称为正则式或正规式）的表示方法描述语言。正则表达式可以描述所有通过对某个字母表上的符号应用这些运算符而得到的语言。在这种表示法中，如果使用 letter_表示任一字母或下画线，用 digit_表示数码，那么可以使用如下的正则表达式描述对应于 C 语言标识符的语言：

$$\text{letter\_}(\text{letter\_}\,|\,\text{digit\_})^*$$

上式中的竖线表示并运算，括号用于把子表达式组合在一起，星号表示 0 个或多个括号中表达式的连接，将 letter_和表达式的其余部分并列表示连接运算。

正则表达式可以由较小的正则表达式按照如下规则递归地构建。每个正则表达式 r 表示一个语言 $L(r)$，这个语言也是根据 r 的子表达式所表示的语言递归地定义的。下面的规则定义了某个字母表上的正则表达式以及这些表达式所表示的语言。

1. 归纳基础

如下两个规则构成了归纳基础：

（1）ε 是一个正则表达式，$L(\varepsilon)=\{\varepsilon\}$，即该语言只包含空串。

（2）如果 α 是字母表上的一个符号，那么 α 是一个正则表达式，并且 $L(\alpha)=\{\alpha\}$。也就是说，这个语言仅包含一个长度为 1 的符号串 α。

2. 归纳步骤

由小的正则表达式构造较大的正则表达式的步骤有 4 个。假定 r 和 s 都是正则表达式，分别表示语言 $L(r)$和 $L(s)$，那么：

（1）$(r)|(s)$是一个正则表达式，表示语言 $L(r)\cup L(s)$。

（2）$(r)(s)$是一个正则表达式，表示语言 $L(r)L(s)$。

（3）$(r)^*$ 是一个正则表达式，表示语言$(L(r))^*$。

（4）(r)是一个正则表达式，表示语言 $L(r)$。最后这个步骤是说在表达式的两边加上括号并不影响表达式所表示的语言。

按照上面的定义，正则表达式经常会包含一些不必要的括号。如果采用如下的约定，就可以去掉一些括号：

（1）闭包运算具有最高的优先级，并且是左结合的。

（2）连接运算具有次高的优先级，并且也是左结合的。

（3）并运算的优先级最低，并且也是左结合的。

例如，可以根据这个约定将$(a)|((b)^*(c))$改写成 $a|b^*c$。这两个表达式都表示同样的串集合，其中的元素要么是单个 a，要么是由 0 个或多个 b 后面再跟一个 c 组成的串。

例 3.4 令字母表 $\Sigma=\{a,b\}$，那么：

（1）正则表达式 $a|b$ 表示集合$\{a,b\}$。

（2）正则表达式$(a|b)(a|b)$表示$\{aa,bb,ab,ba\}$，即由 a 和 b 构成的所有长度为 2 的串集合。表示同样集合的另一个正则表达式是 $aa|bb|ab|ba$。

（3）正则表达式 a^* 表示仅由字母 a 构成的所有串的集合，包括空串。

（4）正则表达式$(a|b)^*$ 表示由 a 和 b 构成的所有串的集合，包括空串。

如果两个正则表达式 r 和 s 表示同样的语言，就说 r 和 s 等价，写作 $r=s$。例如$(a|b)=(b|a)$。

正则表达式遵守一些代数定律，它们可用于正则表达式的等价变换，表 3.4 列出了正则

表达式 r、s 和 t 遵守的部分代数定律。

表 3.4 正则表达式 r、s 和 t 遵守的部分代数定律

定 律	描 述	定 律	描 述
$r\|s=s\|r$	并是可交换的	$\varepsilon r=r, r\varepsilon=r$	ε 是连接的恒等元素
$r\|(s\|t)=(r\|s)\|t$	并是可结合的	$r^*=(r\|\varepsilon)^*$	ε 肯定出现在一个闭包中
$(rs)t=r(st)$	连接是可结合的	$r^{**}=r^*$	* 是幂等的
$r(s\|t)=rs\|rt, (s\|t)r=sr\|tr$	连接对并是可分配的		

3.3.3 正则定义

为方便表示，希望给某些正则表达式命名，并在以后的正则表达式中像使用符号一样使用这些名字。如果字母表 Σ 是基本符号的集合，那么一个正则定义(regular definition)是具有如下形式的定义序列：

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

⋮

$d_n \rightarrow r_n$

其中：

- 每个 d_i 都是一个新符号，它们都不在 Σ 中，并且各不相同。
- 每个 r_i 是字母表 $\Sigma \cup \{d_1, d_2, \cdots, d_{i-1}\}$ 上的正则表达式。

限制 r_i 中只含有 Σ 中的符号和在它之前定义的各个 d_j，因此避免了递归定义的问题，并且可以为每个 r_i 构造出只包含 Σ 中符号的正则表达式。可以首先将 r_2(它只能使用 d_1)中的 d_1 替换为 r_1，然后将 r_3 中的 d_1 和 d_2 替换为 r_1 和(替换后的)r_2，依此类推。最后将 r_n 中的 $d_i(i=1,2,\cdots,n-1)$ 替换为 r_i 替换后的版本，在这些版本中都只包含 Σ 中的符号。

例 3.5 C 语言的标识符是由字母、数字和下画线组成的串。下面是 C 语言标识符对应的语言的一个正则定义。

letter→A|B|⋯|Z|a|b|⋯|z|_

digit→0|1|⋯|9

id→letter_(letter|digit)*

例 3.6 (整型或浮点型)无符号数是形如 5280、0.01234、6.336E4 或 1.89E−4 的串。下面的正则定义给出了这类符号串的精确归约：

digit→0|1|⋯|9

digits→digit digit*

optionFraction→.digit|ε

optionExponent→(E(+|−|ε)digits)|ε

number→digits optionFraction optionExponent

在这个定义中，optionFraction 要么是空串，要么是小数点后跟一个或多个数码。optionExponent 如果不是空串，就是字母 E 后跟一个可选的+或−，再跟一个或多个数码。

请注意，小数点后至少要跟一个数码，所以 number 和 1.不匹配，但和 1.0 匹配。

3.3.4 正则文法和正则式的等价性

一个正则语言可以由正则文法定义，也可以由正则表达式定义，对任意一个正则文法，存在一个定义同一个语言的正则表达式；反之，对每个正则表达式，存在一个生成同一个语言的正则文法。有些正则语言很容易用文法定义，有些正则语言更容易用正则表达式定义。本节介绍两者间的转换，从结构上建立它们的等价性。

1. 将正则表达式转换成正则文法

将 E 上的一个正则表达式 r 转换成文法 $G=(V_N,V_T,S,P)$。令 $V_T=\Sigma$，确定产生式和 V_T 的元素的方法如下：

选择一个非终结符 S 生成类似产生式的形式：$S\rightarrow r$，并将 S 定为 G 的识别符号。为表述方便，将 $S\rightarrow r$ 称作正则表达式产生式，因为在→的右部中含有“.”“*”或“|”等正则表达式符号，不是 V 中的符号。

若 x 和 y 都是正则表达式，则将形如 $A\rightarrow xy$ 的正则表达式产生式重写成 $A\rightarrow xB$ 和 $B\rightarrow y$ 两个产生式，其中 B 是新选择的非终结符，即 $B\in V_N$。

将形如 $A\rightarrow x^* y$ 的正则表达式产生式重写为

$A\rightarrow xB$

$A\rightarrow y$

$B\rightarrow xB$

$B\rightarrow y$

其中 B 为一个新的非终结符。

将形如 $A\rightarrow x|y$ 的正则表达式产生式重写为

$A\rightarrow x$

$A\rightarrow y$

不断利用上述规则进行变换，直到每个产生式都符合正则文法的形式。

例 3.7 将 $r=a(a|d)^*$ 转换成相应的正则文法。

令 S 是文法的开始符，首先形成 $S\rightarrow a(a|d)^*$，然后形成 $S\rightarrow aA$ 和 $A\rightarrow(a|d)^*$，再经过变换形成

$S\rightarrow aA$，$A\rightarrow(a|d)B$

$A\rightarrow\varepsilon$，$B\rightarrow(a|d)B$

$B\rightarrow\varepsilon$

进而变换为全部符合正则文法的形式：

$S\rightarrow aA$

$B\rightarrow aB$

$A\rightarrow aB$

$B\rightarrow dB$

$A\rightarrow dB$

$B\rightarrow\varepsilon$

$A\rightarrow\varepsilon$

2. 将正则文法转换成正则表达式

这一转换过程基本上是上面的过程的逆过程，最后只剩下一个开始符定义的正则表达式。其转换规则如表 3.5 所示。

表 3.5 正则文法到正则表达式的转换规则

序 号	正 则 文 法	正则表达式
规则 1	$A\rightarrow xB, B\rightarrow y$	$A=xy$
规则 2	$A\rightarrow xA \mid y$	$A=x^* y$
规则 3	$A\rightarrow x, A\rightarrow y$	$A=x \mid y$

例 3.8 文法 $G[s]$如下：

$S\rightarrow aA$

$S\rightarrow a$

$A\rightarrow aA$

$A\rightarrow dA$

$A\rightarrow a$

$A\rightarrow d$

首先有

$S=aA \mid a$

$A=(aA \mid dA) \mid (a \mid d)$

再将 A 的正则表达式变换为 $A=(a \mid d)A \mid (a \mid d)$，又变换为 $A=(a \mid d)^* (a \mid d)$，再将 A 的右部代入 S 的正则表达式得

$S=a(a \mid d)^* (a \mid d) \mid a$

再利用正则表达式的代数变换可依次得到

$S=a(a \mid d)^* (a \mid d) \mid \varepsilon$

$S=a(a \mid d)^*$

$a(a \mid d)^*$ 即为所求。

3.3.5 状态转换图

词法分析器的主要任务是将源代码分解为一个个词素。在识别词素的过程中，词法分析器必须根据词法单元的模式匹配源程序中的字符流，这是一项非常关键的工作。为了更高效地进行词法分析，首先将模式转换成具有特定风格的流图，称为状态转换图(transition diagram)。在本节中，将用人工方式将正则表达式表示的模式转换为状态转换图。

状态转换图有一组被称为状态(state)的结点或圆圈。词法分析器在扫描输入串的过程中寻找和某个模式匹配的词素，而状态转换图中的每个状态代表一个可能在这个过程中出现的情况。可以将一个状态看作对已经看到的位于 lexemeBegin 指针和 forward 指针之间的字符的总结，它包含了在进行词法分析时需要的全部信息。

状态转换图中的边(edge)从图的一个状态指向另一个状态。每条边的标号包含了一个或多个符号。如果处于某个状态 s，并且下一个输入符号是 a，就会寻找一条从 s 离开且标号为 a 的边(该边的标号中可能还包括其他符号)。如果找到了这样的一条边，就将

forward 指针前移，并进入状态转换图中该边所指的状态。这里假设所有状态转换图都是确定的，这意味着对于任何一个给定的状态和任何一个给定的符号，最多只有一条从该状态离开的边的标号包含该符号。这个限制保证了词法分析器能够高效地处理输入流，并且减小了出现错误的可能性。

一些关于状态转换图的重要约定如下：

(1) 某些状态称为接受状态或最终状态。这些状态表明已经找到了一个词素，虽然实际的词素可能并不包括 lexemeBegin 指针和 forward 指针之间的所有字符。这里用双层的圈表示一个接受状态，并且如果该状态要执行一个动作——通常是向语法分析器返回一个词法单元和相关属性值——将把这个动作附加到该接受状态上。

(2) 如果需要将 forward 指针回退一个位置(即相应的词素并不包含那个在最后一步到达接受状态的符号)，那么将在该接受状态的附近加上一个 * 。这些例子都不需要将 forward 指针回退多个位置，但万一出现这种情况，需要为接受状态附加相应数目的 * 。

(3) 有一个状态被指定为开始状态，也称初始状态，该状态由一条没有出发结点的、标号为 start 的边指明。在读入任何输入符号之前，状态转换图总是位于它的开始状态。

(4) 状态转换图通常按照状态编号的顺序排列。当状态转换图非常复杂时，按照状态编号的顺序排列可以使状态转换图更加易于阅读和理解。

(5) 状态转换图的边上可以标注多个符号，这些符号可以是字母、数字、特殊字符或者任何能够组成词素的字符。

在实际编写词法分析器时，通常使用工具生成状态转换图，而不是人工编写。这些工具可以根据给定的正则表达式自动生成状态转换图。然而，了解人工编写状态转换图的方法有助于更好地理解词法分析器的内部工作原理，并且更好地理解工具生成的状态转换图。

例 3.9 图 3.5 给出了能够识别所有与词法单元 relop 匹配的词素的状态转换图。从初始状态 0 开始。如果看到的第一个输入符号是<，那么在所有与 relop 模式匹配的词素中，只能选择<、<>或<=。因此进入状态 1 并查看下一个字符。如果这个字符是=，则识别出

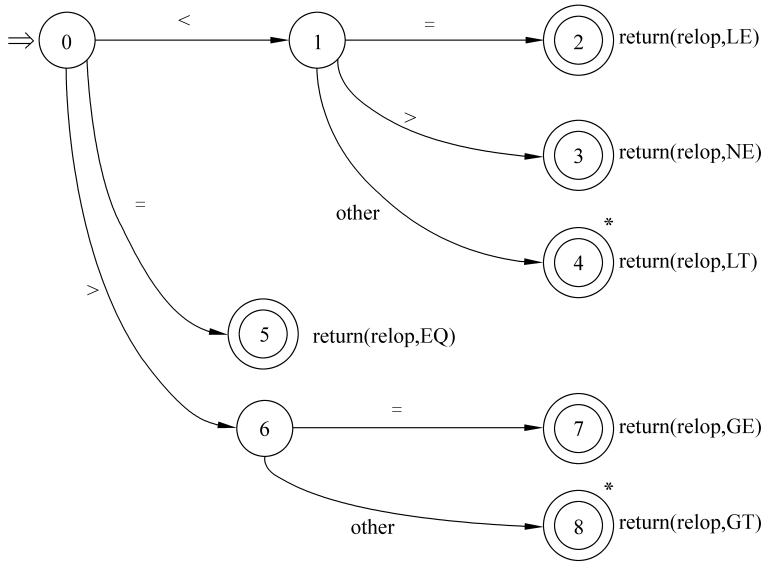


图 3.5 词法单元 relop 的状态转换图

词素<=，进入状态 2 并返回属性值为 LE 的 relop 词法单元。其中的符号常量 LE 代表了这个具体的比较运算符。如果在状态 1 时看到的下一个字符是>，那么就会得到词素<>，从而进入状态 3 并返回一个词法单元，表明已经找到一个不等运算符。而对于其他字符，识别得到的词素是<，则进入状态 4 并向语法分析器返回这个信息。请注意，状态 4 有一个 * ，说明必须将输入回退一个位置。

如果在状态 0 时看到的第一个字符是=，那么这个字符必定是要识别的词素。立即从状态 5 返回这个信息。其余的可能性是第一个字符为>的情况。那么应该进入状态 6，并根据下一个字符确定词素是>=（如果看到下一个字符为=）还是>（对于任何其他字符）。注意，如果在状态 0 时看到的是不同于<、=或>的字符，就不可能看到一个与 relop 匹配的词素，因此这个状态转换图将不会被使用。

识别关键字及标识符时有一个问题要解决。通常，像 if 或 then 这样的关键字是被保留的，因此，虽然它们看起来很像标识符，但它们不是标识符。因此，尽管通常使用如图 3.6 所示的状态转换图寻找标识符的词素，但这个图也可以识别出连续使用的例子中的关键字 if、then 及 else。

可以使用两种方法处理那些看起来很像标识符的关键字：

（1）初始化时就将各个关键字填入符号表中。符号表条目的某个字段会指明这些串并不是普通的标识符，并指出它们所代表的词法单元。假设图 3.6 中使用了这种方法。当找到一个标识符时，如果该标识符尚未出现在符号表中，就会调用 installID 函数将此标识符放入符号表中，并返回一个指针，指向这个刚找到的词素所对应的符号表条目。当然，任何在词法分析时不在符号表中的标识符都不可能是一个关键字，因此它的词法单元是 id。函数 getToken 查看对应刚找到的词素的符号表条目，并根据符号表中的信息返回该词素所代表的词法单元名——要么是 id，要么是一个在初始化时就被加入符号表中的关键字词法单元。

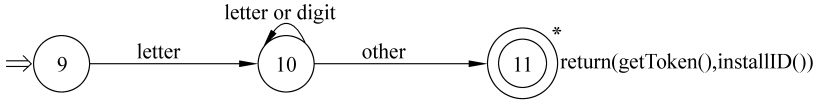


图 3.6　id 和关键字的状态转换图

（2）为每个关键字建立单独的状态转换图。图 3.7 是关键字 then 的一个例子。请注意，这样的状态转换图包含的状态表示看到该关键字的各个后续字母后的情况，最后是一个非字母或数字的测试，也就是检查后面是否为某个不可能成为标识符一部分的字符。有必要检查该标识符是否结束，否则在碰到像 thenextvalue 这样以 then 为前缀的 id 词法单元时，可能会错误地返回词法单元 then。如果采用这个方法，必须设定词法单元之间的优先级，使得当一个词素同时匹配 id 的模式和关键字的模式时，优先识别关键字词法单元，而不是 id 词法单元。这个例子中并没有使用这个方法，这也是没有对图 3.7 中的状态进行编号的原因。

在图 3.6 中可以看到，id 的状态转换图有一个简单的结构。由状态 9 开始，它检查被识别的词素是否以一个字母开头，如果是则进入状态 10。只要接下来的输入包含字母或数

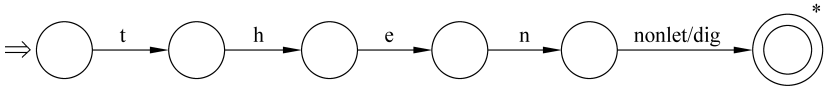


图 3.7　关键字 then 的状态转换图

码，就一直停留在状态 10。当第一次遇到不是字母或数码的其他任何字符时，便转入状态 11 并接受刚刚找到的词素。因为最后一个字符并不是标识符的一部分，所以必须将输入回退一个位置，并且如上面所讨论的那样，将已经找到的词素加入符号表中，并判断它究竟是一个关键字还是一个真正的标识符。

图 3.8 显示了词法单元 number 的状态转换图，它是本节到目前出现的最复杂的状态转换图。从状态 12 开始，如果看到一个数码，就转入状态 13。在该状态，可以读入任意数量的其他数码。然而，如果看到了一个不是数码、小数点和 E 的其他字符，就得到了一个整数形式的数字，如 123。这种情形在进入状态 20 时进行处理，在该状态返回词法单元 number 以及一个指向常量表条目的指针，刚刚找到的词素便放在这个常量表条目中。这些机制并没有在这个状态转换图中显示出来，但它们和处理标识符的方法相似。

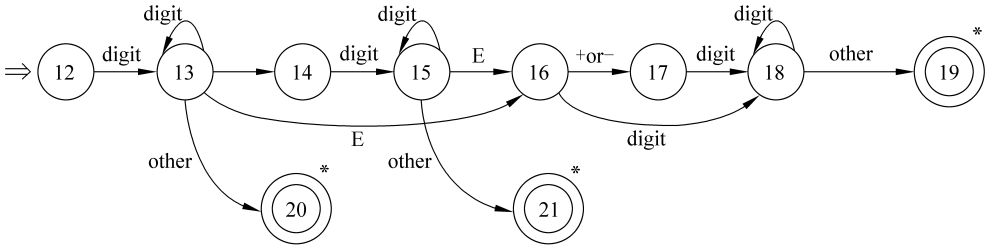


图 3.8　number 的状态转换图

如果在状态 13 看到的是一个小数点，那么就看到一个可选的小数部分。于是，进入状态 14，并寻找一个或多个更多的数码，状态 15 就被用于此目的。如果看到一个 E，那么就看到了一个可选的指数部分，它的识别任务由状态 16～19 完成。如果在状态 15 看到了一个不是 E 和数码的其他字符，那么就到达了小数部分的结尾，这个数字没有指数部分，将通过状态 21 返回刚刚找到的词素。

最后一个状态转换图如图 3.9 所示，它用于识别空白符。在该状态转换图中，寻找一个或多个空白符，在图 3.9 中用 delim 表示。典型的空白符有空格、制表符、换行符，还有可能包括那些根据语言设计不可能出现在任何词法单元中的字符。

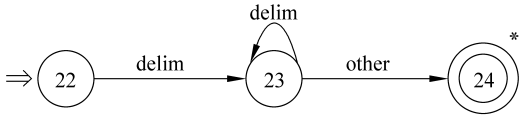


图 3.9　空白符的状态转换图

注意：在状态 24 中找到了一个连续的空白符组成的块，且后面还跟随一个非空白符。将输入回退到这个非空白符的开头，但并不向语法分析器返回任何词法单元；相反，必须在这个空白符之后再次启动词法分析过程。

小结

本章主要介绍了词法分析的相关概念，以下是主要内容：

- 词法单元、模式和词素。词法单元由一个词法单元名和一个可选的属性值组成。模式描述了一个词法单元的词素可能具有的形式。词素是源程序中的一个字符序列，它和某个词法单元的模式匹配，并被词法分析器识别为该词法单元的一个实例。
- 词法单元的属性。一个词法单元的属性值是一个指向符号表中该词法单元对应条目的指针。
- 字母表、串和语言。字母表是一个有限的符号集合。某个字母表上的一个串是该字母表中符号的一个有穷序列。语言是某个给定字母表上一个任意的可数的串集合。
- 正则表达式。它可以描述所有通过对某个字母表上的符号应用规定的运算符得到的语言。正则表达式可以由较小的正则表达式按照如下规则递归地构建：每个正则表达式 r 表示一个语言 $L(r)$，这个语言也是根据 r 的子正则表达式所表示的语言递归地定义的。
- 正则文法和正则表达式的等价性。一个正则语言可以由正则文法定义，也可以由正则表达式定义，对任意一个正则文法，存在一个定义同一个语言的正则表达式；反之，对每个正则表达式，存在一个生成同一个语言的正则文法。即两者可以相互转换。
- 状态转换图。作为构造词法分析器的一个中间步骤，首先将模式转换成具有特定风格的流图，称为状态转换图。状态转换图有一组被称为状态的结点或圆圈。状态图中的边从图的一个状态指向另一个状态。

习题 3

3.1 简述词法单元、模式和词素的含义。

3.2 简述串的前缀、后缀、子串、真子串和子序列的含义。

3.3 简述正则表达式和正则定义的含义。

3.4 将下面的 C++ 程序划分成正确的词素序列。

```
float limitedSquare(x)
{
    float x;
    /* returns x-squared,but never more than 100 */
    return(x<=-10.0|x>=10.0)?100:x * x;
}
```

哪些词素应该有相关联的词法值？应该具有什么值？

3.5 像 HTML 或 XML 之类的标记语言不同于传统的程序设计语言。它们要么包含很多标记(如 HTML)，要么使用由用户自定义的标记集合(如 XML)，而且标记还可以带有参数。请指出如何把如下的 HTML 文档划分成适当的词素序列。

```
Here is a photo of <B>my house</B>;
```

```
<P><IMG SRC "house.gif"><BR>
See <A HREF "porePix.html">More Pictures</A>if you
liked that one.<P>
```

哪些词素应该具有相关联的词法值？应该具有什么样的值？

3.6 说明在一个长度为 n 的字符串中分别有多少个前缀、后缀、真前缀、子串和子序列。

3.7 描述下列正则表达式定义的语言：

(1) $a(a \mid b)^* a$

(2) $((\varepsilon \mid a)b^*)^*$

(3) $(a \mid b)^* a(a \mid b)(a \mid b)$

(4) $a^* ba^* ba^* ba^*$

3.8 很多语言都是大小写敏感的(case sensitive)，因此这些语言的关键字只能有一种写法，描述这些关键字的词素的正则表达式就很简单。但是，像 SQL 这样的语言是大小写不敏感的(case insensitive)，一个关键字既可以大写，也可以小写，还可以大小写混用。因此，SQL 中的关键字 SELECT 可以写成 select、Select 或 sElEcT。请描述出如何用正则表达式表示大小写不敏感的语言中的关键字。给出描述 SQL 中的关键字 select 的表达式，以说明你的思想。

3.9 写出下列语言的正则定义：

(1) 包含 5 个元音的所有小写字母串，这些串中的元音按顺序出现。

(2) 所有由按词典递增序排列的小写字母组成的串。

(3) 注释，即/*和*/之间的串，且串中没有不在双引号(")中的*/。

(4) 所有不重复的数码组成的串。

(5) 所有最多只有一个重复数码的串。

(6) 所有由偶数个 a 和奇数个 b 组成的串。

(7) 以非正式方式表示的国际象棋的步法的集合，如 p-k4 或 kbpxgn。

(8) 所有由 a 和 b 组成且不含子串 abb 的串。

(9) 所有由 a 和 b 组成且不含子序列 abb 的串。

3.10 SQL 支持一种不成熟的模式描述方式，其中有两个具有特殊含义的字符；下画线(_)表示任意一个字符；百分号(%)表示包含 0 个或多个字符的串。此外，程序员还可以将任意一个字符(例如 e)定义为转义字符。那么，在_、%或者另一个 e 之前加上一个 e，就使得这个字符只表示它的字面值。假设已经知道哪个字符是转义字符，说明如何将任意 SQL 模式表示为一个正则表达式。

3.11 正则表达式 $r\{m,n\}$ 和模式 r 的 $m \sim n$ 次重复出现相匹配。例如，$a\{1,5\}$ 和由 1～5 个 a 组成的串匹配。证明：对于每一个包含这种形式的重复运算符的正则表达式，都存在一个等价的不包含重复运算符的正则表达式。

3.12 为下列的字符集合写出对应的字符类。

(1) 英文字母的前 10 个字母(从 a 到 j)，包括大写和小写。

(2) 所有小写辅音字母的集合。

(3) 十六进制中的数码(对大于 9 的数码，自己决定大写或小写)。

(4) 可以出现在一个合法的英语句子后面的字符集(例如感叹号)。

3.13 设有正则文法 G：

$A \rightarrow aB \mid bB$

$B \rightarrow aC \mid a \mid b$

$C \rightarrow aB$

试给出该文法对应的正则表达式。

3.14 给出下述文法对应的正则表达式：

$S \rightarrow 0A \mid 1B$

$A \rightarrow 1S \mid 1$

$B \rightarrow 0S \mid 0$

3.15 将 $R=(a \mid b)(aa)^*(a \mid b)$转换成相应的正则文法。

3.16 给出题 3.7 中各个正则表达式所描述的语言的状态转换图。

3.17 给出识别题 3.9 中各个正则表达式所描述的语言的状态转换图。

3.18 为正则文法 $G[W]$

$W \rightarrow Ua \mid a$

$U \rightarrow Va \mid Ub \mid c$

$V \rightarrow Wc \mid b$

画出相应的状态转换图。由运行状态转换图识别输入符号串 *cbacabba* 是否是该文法的句子。

拓展阅读：正则表达式的扩展与汉语词法分析

一、正则表达式的扩展

自从 Kleene 在 20 世纪 50 年代提出了带有基本运算符并、连接和闭包的正则表达式之后，已经出现了很多种针对正则表达式的扩展，它们被用来增强正则表达式描述串模式的能力。在这里，将介绍一些最早出现在像 Lex 这样的 UNIX 实用程序中的扩展表示法。这些扩展表示法在词法分析器的归约中非常有用。

(1) 一个或多个实例。单目后缀运算符＋表示一个正则表达式及其语言的正闭包。也就是说，如果 r 是一个正则表达式，那么$(r)^+$就表示语言$(L(r))^+$。运算符$^+$和*具有同样的优先级和结合性。两个有用的代数定律 $r^*=r^+ \mid \varepsilon$ 和 $r^+=rr^*=r^*r$ 说明了闭包和正闭包之间的关系。

(2) 零个或一个实例。单目后缀运算符?的意思是“零个或一个出现”。也就是说 r?等价于 $r \mid \varepsilon$，换句话说，$L(r?)=L(r) \cup \{\varepsilon\}$。运算符?与运算符$^+$和*具有同样的优先级和结合性。

(3) 字符类。一个正则表达式 $a_1 \mid a_2 \mid \cdots \mid a_n$(其中 a_i 是字母表中的各个符号)可以缩写为$[a_1a_2\cdots a_n]$。更重要的是，当 $a_1, a_2, \cdots, a_n$ 形成一个逻辑上连续的序列时，例如连续的大写字母、小写字母或数码时，可以把它们表示成 $a_1\text{-}a_n$。也就是说，只写出第一个和最后一个符号，中间用连字符隔开。因此，$[abc]$是 $a \mid b \mid c$ 的缩写，$[a\text{-}z]$是 $a \mid b \mid \cdots \mid z$ 的缩写。

例 3.10 根据这些缩写表示法，可以将例 3.5 中的正则定义改写为

letter$\rightarrow [A\text{-}Za\text{-}z]$

digit→[0-9]

id→letter_(letter|digit)*

例 3.6 的正则定义可以改写为

digit→[0-9]

digits→digit+

number→digits(.digits)? (E{+−}? digits)?

二、汉语词法分析

因为不同语言各自的特性，词法分析具体做法是不同的，以英语和汉语为例作为对比：英语用空格隔开，无须分词，用词形态变化来表示语法关系。汉语词与词紧密相连，没有明显的分界标志，词形态变化少，靠词序或虚词来表示。对于汉语词法分析而言，以句子“警察正在详细调查事故原因”为例，其分析流程如图 3.10 所示。

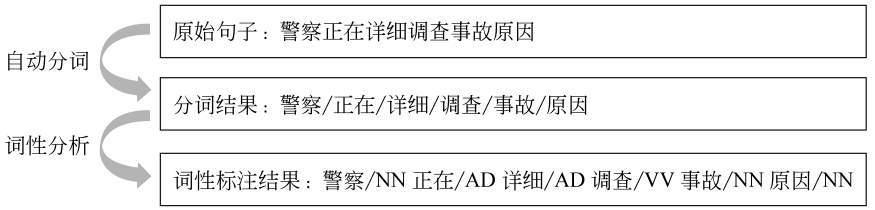


图 3.10　汉语词法分析流程

从上面的例子可以看出汉语词法分析包括两个主要任务：自动分词和词性标注。

1. 自动分词的 3 个问题

自动分词是将输入的汉字串切成词串。

自动分词面临 3 个问题：歧义问题、未登录词问题和分词标准问题。

1）歧义问题

歧义指的是切分歧义，即对同一个待切分字符串存在多个分词结果。歧义分为交集型歧义、组合型歧义和混合歧义。

（1）交集型歧义。字符串 abc 既可以切分成 a/bc，也可以切分成 ab/c。其中，a、bc、ab、c 是词。举个例子：“白天鹅”可以切分成“白天/鹅”和“白/天鹅”，“研究生命”可以切分成“研究/生命”和“研究生/命”。至于具体要取哪一种分词结果，需要根据上下文推断。也许对于人来说，这些歧义很好分辨；但是对计算机而言，这是一个很重要的问题。针对交集型歧义，研究者提出链长这一概念：交集型切分歧义所拥有的交集串的个数称为链长。例如，“中国产品质量”的交集串集合为{国，产，品，质}，链长为 4；“部分居民生活水平”的交集串集合为{分，居，民，生，活，水}，链长为 6。

（2）组合型歧义。ab 为词，而 a 和 b 在句子中又可分别单独成词。例如，“门把手弄坏了”切分为“门/把手/弄/坏/了”和“门/把/手/弄/坏/了”，“把手”本身是一个词，分开之后可以分别成词。

（3）混合歧义。以上两种情况通过嵌套、交叉组合等而产生的歧义。例如，“这篇文章写得太平淡了”，其中“太平”是组合型歧义，“太平淡”是交集型歧义。

通过上面的例子可以看出，歧义问题在汉语中是十分常见的。

2）未登录词问题

未登录词问题是指句子中出现词典中没有收录过的人名、地名、机构名、专业术语、译名、新术语等。该问题在文本中的出现频度远远高于歧义问题。未登录词类型如下：

- 实体名称，包括汉语人名（张三、李四）、汉语地名（黄山、韩村）、机构名（外贸部、国际卫生组织）。
- 数字、日期、货币等。
- 商标字号（可口可乐、同仁堂）。
- 专业术语（万维网、贝叶斯算法）。
- 缩略语（五讲四美、计生办）。
- 新词语（美刀、卡拉 OK）。

未登录词问题是分词错误的主要来源。

3）分词标准问题

对于“汉语中什么是词”这个问题，不仅普通人有认识上的偏差，即使是语言专家，在这个问题上依然有不小的差异。缺乏统一的分词规范和标准这种问题也反映在分词语料库上，不同语料库的数据无法直接拿过来混合训练。

2. 自动分词方法

接下来介绍进行自动分词的技术方法，基本方法有机械分词法、语义分词法、基于统计的分词法和人工智能分词法。

1）机械分词法

机械分词法按照一定的策略将待匹配的字符串和一个已建立好的充分大的词典中的词进行匹配，若找到某个词条，则说明匹配成功，识别了该词。基于词典的分词方法在传统分词方法中是应用最广泛、分词速度最快的一类，实现相对简单。这类方法主要有以下两种：

（1）最大匹配法。基本思想：先建立一个最长词条字数为 L 的词典，然后按正向（或逆向）取句子前（或后）L 个字查词典，如查不到，则去掉最后一个字继续查，一直到查到一个词为止。最大匹配法以及其改进方案是基于词典和规则的。其优点是实现简单，算法运行速度快。其缺点是严重依赖词典，无法很好地处理分词歧义和未登录词。

（2）最少切分法。基本思想：假设待切分字符串为 $S=c_1c_2\cdots c_n$，其中 c_i 为单个字，串长为 $n(n\geqslant 1)$。建立一个结点数为 $n+1$ 的切分有向无环图 G，若 $w=c_ic_{i+1}\cdots c_j(0<i<j\leqslant n)$ 是一个词，则在结点 v_{i-1} 和 v_j 之间建立有向边。从产生的所有路径中选择路径最短的（词数最少的）作为最终分词结果。该种方法的优点在于需要的语言资源（词表）不多。但是，它对许多歧义字段难以区分；当最短路径有多条时，选择最终的输出结果缺乏应有的标准；另外，字符串长度较大和选取的最短路径数增大时，长度相同的路径数急剧增加，选择正确结果的困难越来越大。

2）语义分词法

语义分词法引入了语义分析，对自然语言自身的语言信息进行了更多的处理，如扩充转移网络法、知识分词语义分析法、邻接约束法、综合匹配法、后缀分词法等。扩充转移网络法是一种普遍应用于数据库自然语言查询中的语法分析方法，它主要由递归网络加一个测试集以及一组寄存器组成。分析句子时，测试条件（检查弧上所标识的语法成分条件及其他相关测试条件）以确定是否与一条弧匹配，测试结果为真才允许通过该弧，寄存器则用来保存

被分析单词(或短语)的有关特性及分析过程的中间结果。可见,扩充转移网络法的实现需要建立一个语法知识库,用于作为弧间状态迁移的测试条件。这也是语义分词法的复杂之处。语法知识库的建立在提高了分词的精度的同时也加大了实现的难度。相对于机械分词法而言,语义分析法的切分深度更进了一步。

3) 基于统计的分词法

词是固定的字的组合,在文本中相邻的字同时出现的次数越多,越有可能是一个词,因此,计算上下文中相邻的字出现的联合概率,可以判断字成词的概率。通过对语料中相邻共现的各个字的组合频度进行统计,计算它们的互现信息。互现信息体现了汉字之间结合关系的紧密程度,当紧密程度高于某一个阈值时,可判定这几个字构成一个词。这种方法的优点是不受待处理文本领域的限制,不需要专门的词典。统计分词法以概率论为理论基础,将上下文中汉字组合串的出现抽象成随机过程,随机过程的参数可以通过大规模语料库训练得到。基于统计的分词方法中常见的模型有 N 元统计模型、隐马尔可夫模型、条件随机场模型、神经网络模型及最大熵模型等。

4) 人工智能分词法

人工智能是对信息进行智能化处理的一种模式。人工智能分词法主要有两种:

(1) 基于心理学的符号处理方法。专家系统模拟人脑的功能,构造推理网络,经过符号转换,从而进行解释性处理。专家系统应用到分词中就有了专家系统分词法。它将自动分词过程看作知识推理过程,力求从结构和功能上将分词过程和实现分词所依赖的汉语语法知识、句法知识以及部分语义知识分离,需要考虑知识表示、知识库的逻辑结构与知识库的维护。这种方法的不足在于其串行处理机制,学习能力低,对于外界最新的信息反应滞后。

(2) 基于生理学的模拟方法。旨在模拟人脑的神经系统机构的运作机制,是以非线性并行处理为主流的一种非逻辑的信息处理方法。最常见的就是深度学习神经网络方法,在分词任务中将句子输入神经网络,通过自学习和训练修改内部权值达到正确的分词结果。该方法最大的特点是知识获取快,这也是神经网络方法的一大特色,并行性、分布性和联结性的网络结构为神经网络的知识获取提供了良好的环境,并通过样本学习和训练自我更新。但神经网络的知识分布在整个系统内部,对用户而言是黑箱,而且它对于结论不能作出合理的解释。该方法在实践环节中涉及知识库的组织和神经网络推理机制的建立。

3. 词性标注

词性标注是确定每个词的词性并加以标注。

与英文相比,中文词性标注主要有以下几个难点:

(1) 缺乏直接判断的依据。汉语是一种缺乏词形态变化的语言,词的类别不能直接从词的形态变化上判别。

(2) 常用词兼类现象严重。在对现代汉语常用词的收取统计中,兼类词(即指一个词有两种或两种以上的词性,又称同词异类)所占的比例高达 22.5%,且越是常用的词,不同的用法越多。由于兼类使用程度高,兼类现象涉及汉语中大部分词类,因而造成在汉语文本中词类歧义排除的任务量巨大。

(3) 研究者主观原因造成的困难。由于语言学界在词性划分的目的、标准等问题上还存在分歧,导致目前还没有一个被广泛认可的汉语词类划分标准。不同机构对词类划分的粒度和标记符号都不统一。词类划分标准和标记符号集的差异以及分词规范的含混性给中