第3章 词法分析

本章我们主要讨论如何构建一个词法分析器。如果要手动地实现词法分析器，首先建立起每个词法单元的词法结构图或其他描述会有所帮助。然后，我们可以编写代码来识别输入中出现的每个词素，并返回识别到的词法单元的有关信息。

我们也可以通过如下方式自动生成一个词法分析器：向一个*词法分析器生成工具*(lexical-analyzer generator)描述出词素的模式，然后将这些模式编译为具有词法分析器功能的代码。这种方法使得修改词法分析器的工作变得更加简单，因为我们只需改写那些受到影响的模式，无需改写整个程序。这种方法还加快了词法分析器的实现速度，因为程序员只需要在很高的模式层次上描述软件，就可以依赖生成工具来生成详细的代码。我们将在3.5节中介绍一个名为Lex的词法分析器生成工具(它的一个最新的变体称为Flex)。

在介绍词法分析器生成工具之前，我们先介绍正则表达式。正则表达式是一种可以很方便地描述词素模式的方法。我们将介绍如何对正则表达式进行转换：首先转换为不确定有穷自动机，然后再转换为确定有穷自动机。后两种表示方法可以作为一个“驱动程序”的输入。这个驱动程序就是一段模拟这些自动机的代码，它使用这些自动机来确定下一个词法单元。这个驱动程序以及对自动机的规约形成了词法分析器的核心部分。

3.1 词法分析器的作用

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，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。

这种交互过程在图3-1中给出。通常，交互是由语法分析器调用词法分析器来实现的。图中的命令 *getNextToken* 所指示的调用使得词法分析器从它的输入中不断读取字符，直到它识别出下一个词素为止。词法分析器根据这个词素生成下一个词法单元并返回给语法分析器。

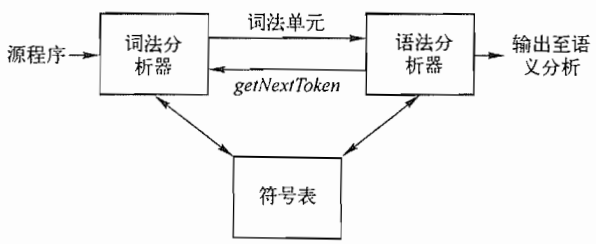


图3-1 词法分析器与语法分析器之间的交互

词法分析器在编译器中负责读取源程序，因此它还会完成一些识别词素之外的其他任务。任务之一是过滤掉源程序中的注释和空白(空格、换行符、制表符以及在输入中用于分隔词法单元的其他字符)；另一个任务是将编译器生成的错误消息与源程序的位置联系起来。例如，词法

分析器可以负责记录遇到的换行符的个数，以便给每个出错消息赋予一个行号。在某些编译器中，词法分析器会建立源程序的一个拷贝，并将出错消息插入到适当位置。如果源程序使用了一个宏预处理器，则宏的扩展也可以由词法分析器完成。

有时，词法分析器可以分成两个级联的处理阶段：

1）*扫描阶段*主要负责完成一些不需要生成词法单元的简单处理，比如删除注释和将多个连续的空白字符压缩成一个字符。

2）*词法分析阶段*是较为复杂的部分，它处理扫描阶段的输出并生成词法单元。

3.1.1 词法分析及语法分析

把编译过程的分析部分划分为词法分析和语法分析阶段有如下几个原因：

1）最重要的考虑是简化编译器的设计。将词法分析和语法分析分离通常使我们至少可以简化其中的一项任务。例如，如果一个语法分析器必须把空白符和注释当作语法单元进行处理，那么它就会比那些假设空白和注释已经被词法分析器过滤掉的处理器复杂得多。如果我们正在设计一个新的语言，将词法和语法分开考虑有助于我们得到一个更加清晰的语言设计方案。

2）提高编译器的效率。把词法分析器独立出来使我们能够使用专用于词法分析任务、不进行语法分析的技术。此外，我们可以使用专门的用于读取输入字符的缓冲技术来显著提高编译器的速度。

3）增强编译器的可移植性。输入设备相关的特殊性可以被限制在词法分析器中。

3.1.2 词法单元、模式和词素

在讨论词法分析时，我们使用三个相关但有区别的术语：

- *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，比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。词法单元名字是由语法分析器处理的输入符号。在后面的内容中，我们通常使用黑体字给出词法单元名。我们将使用词法单元的名字来引用一个词法单元。
- *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。
- *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 3.1 图 3-2 给出了一些常见的词法单元、非正式描述的词法单元的模式，并给出了一些示例词素。下面说明上述概念在实际中是如何应用的。在 C 语句

```
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。 □

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	3.14159, 0, 6.02e23
literal	在两个 "之间，除" 以外的任何字符	"core dumped"

图 3-2 词法单元的例子

在很多程序设计语言中，下面的类别覆盖了大部分或所有的词法单元：

1）每个关键字有一个词法单元。一个关键字的模式就是该关键字本身。

2）表示运算符的词法单元。它可以表示单个运算符，也可以像图 3-2 中的 comparison 那样，表示一类运算符。

3）一个表示所有标识符的词法单元。

4）一个或多个表示常量的词法单元，比如数字和字面值字符串。

5）每一个标点符号有一个词法单元，比如左右括号、逗号和分号。

3.1.3 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。因此，在很多情况下，词法分析器不仅仅向语法分析器返回一个词法单元名字，还会返回一个描述该词法单元的词素的属性值。词法单元的名字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。

我们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。最重要的例子是词法单元 **id**，我们通常会将很多信息和它关联。一般来说，和一个标识符有关的信息——例如它的词素、类型、它第一次出现的位置（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。

识别词法单元时的棘手问题

如果给定一个描述了某词法单元的词素的模式，在与之匹配的词素出现在输入中时识别出匹配的词素是相对简单的。然而，在某些程序设计语言中，要判断是否识别到一个和某词法单元匹配的词素并不是一件轻而易举的事。下面的例子来自 Fortran 语言的固定格式（fixed-format）程序。Fortran 90 中仍然支持固定格式。在语句

```
DO 5 I = 1.25
```

中，在我们看到 1 后的小数点之前，我们并不能确定 DO5I 是第一个词素，即一个标识符词法单元的实例。注意，在 Fortran 语言的固定格式中，空格是被忽略的（这是一种过时的惯例）。假如我们看到的是一个逗号，而不是小数点，那么我们就得到了一个 do 语句

```
DO 5 I = 1,25
```

在这个语句中，第一个词素是关键字 DO。

例 3.2 Fortran 语句

```
E = M * C ** 2
```

中的词法单元名字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_op>
<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，在某些对中，特别是运算符、标点符号和关键字的对中，不需要有属性值。在这个例子中，

词法单元 **number** 有一个整数属性值。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。 □

3.1.4 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。比如，当词法分析器在 C 程序片断

```
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。当出现这种情况时，最简单的错误恢复策略是“恐慌模式”恢复。我们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包括：

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) 用一个字符来替换另一个字符。

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高，不值得使用。

3.1.5 3.1 节的练习

练习 3.1.1： 根据 3.1.2 节中的讨论，将下面的 C++ 程序

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

划分成正确的词素序列。哪些词素应该有相关联的词法值？应该具有什么值？

练习 3.1.2： 像 HTML 或 XML 之类的标记语言不同于传统的程序设计语言，它们要么包含有很多标点符号(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

```
Here is a photo of <B>my house</B>;
<P><IMG SRC = "house.gif"><BR>
See <A HREF = "morePix.html">More Pictures</A> if you
liked that one.<P>
```

划分成适当的词素序列。哪些词素应该具有相关联的词法值？应该具有什么样的值？

3.2 输入缓冲

在讨论如何识别输入流中的词素之前，我们首先讨论几种可以加快源程序读入速度的方法。源程序读入虽然简单，却很重要。由于我们常常需要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，因此这个任务变得有些困难。在 3.1 节的“识别词法单元时的棘手问题”中给出了一个极端的例子。但是在实践中，很多情况下我们的确需要至少向前看一个字符。比如，我们只有读取到一个非字母或数字的字符之后才能确定我们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。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，我们将介绍一种双缓冲区方案，这种方案能够安全地处理向前看多个符号的问题。然后我们将考虑一种改进方法。这种方法使用“哨兵标记”来节约用于检查缓冲区末端的时间。

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来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两个交替读入的缓冲区，如图 3-3 所示。

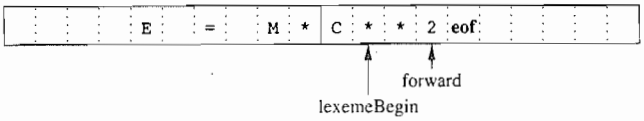


图 3-3 使用一对输入缓冲区

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 4096 字节。我们可以使用系统读取命令一次将 N 个字符读入到缓冲区中，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。当前我们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的其余部分中讨论。

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 3-3 中，我们看到，`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` 指针前移要求我们首先检查是否已经到达某个缓冲区的末尾。如果是，我们必须将 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。只要我们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我们向前看的距离大于 N，我们就决不会在识别这个词素之前覆盖掉这个尚在缓冲区中的词素。

3.2.2 哨兵标记

如果我们采用上一节中描述的方案，那么在每次向前移动 `forward` 指针时，我们都必须检查是否到达了缓冲区的末尾。若是，那么我们必须加载另一个缓冲区。因此每读入一个字符，我们需要做两次测试：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）。如果我们扩展每个缓冲区，使它们在末尾包含一个“哨兵”（sentinel）字符，我们就可以把对缓冲区末端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 3-4 显示的缓冲区安排与图 3-3 一致，只是加入了“哨兵标志”字符。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 3-5 总结了前移 forward 指针的算法。请注意，我们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。只有当我们确实处于缓冲区末尾或输入末尾时，才需要进行更多的测试。

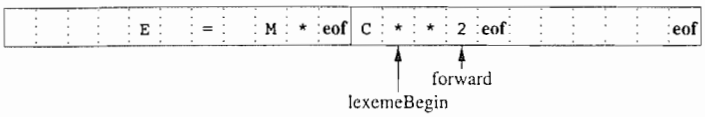


图 3-4 各个缓冲区末端的“哨兵标记”

```
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 3-5 带有哨兵标记的 forward 指针移动算法

3.3 词法单元的规约

正则表达式是一种用来描述词素模式的重要表示方法。虽然正则表达式不能表达出所有可能的模式，但是它们可以高效地描述在处理词法单元时要用到的模式类型。在这一节中，我们将研究正则表达式的形式化表示方法。在 3.5 节中，我们将看到如何将这些表达式运用到词法分析器生成工具中。然后，3.7 节显示了如何将正则表达式转换成能够识别所描述的词法单元的自动机，并由此建立一个词法分析器。

我们会不会用完缓冲区空间?

在大多数现代程序设计语言中，词素很短，向前看一到两个字符就能够确定一个词素，所以数千字节大小的缓冲区就已经足够了。使用 3.2.1 节中介绍的双缓冲区方案肯定没问题。但是仍然存在一些风险。比如，如果字符串包含很多行，那么我们就有可能面临单个词素的长度超过 N 的情况。为了避免长字符串引起的问题，我们可以把它们看作不同组成部分的连接，每个组成部分对应于该字符串的一行。比如，在 Java 语言中，人们习惯于将一个字符串写成多个部分，每个部分占一行，并在每个部分的结尾加上运算符 +，将它们连接起来。

当需要向前看任意多个字符时，就会出现一个更加严重的问题。比如，像 PL/I 这样的语言没有将关键字作为保留字来处理，也就是说，你可以使用一个和某个关键字（比如 DECLARE）同名的标识符。当词法分析器处理以 DECLARE(ARG1,ARG2,……开头的 PL/I 程序的文本时，它不能确定 DECLARE 究竟是一个关键字（此时后面的 ARG1 等是被声明的变量），还是一个带有参数的过程名。因为这个原因，大多数现代程序设计语言都保留关键字。然而，如果不保留关键字，我们可以把像 DECLARE 这样的关键字当作一个二义性的标识符，由语法分析器来解决这个问题。此时语法分析器就需要在符号表中查询有关信息。

3.3.1 串和语言

字母表（alphabet）是一个有限的符号集合。符号的典型例子包括字母、数位和标点符号。集合{0, 1}是*二进制字母表*（binary alphabet）。ASCII 是字母表的一个重要例子，它被用于很多软件系统中。Unicode 包含了大约 100000 个来自世界各地的字符，它是字母表的另一个重要例子。

实现多路分支

我们也许会认为图 3-5 的算法中的 switch 需要执行很多步，而且将 eof 分支放在开头也不是明智的选择。但事实上，我们按照什么顺序列出针对各个字符的 case 并不重要。在实践中，可以用一个以字符为下标的地址数组来存放对应于各个 case 的指令地址，并根据此数组中找到的目标地址一次完成跳转。

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。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，通常记作 $|s|$，是指 s 中符号出现的次数。例如，banana 是一个长度为 6 的串。*空串*（empty string）是长度为 0 的串，用 ϵ 表示。

语言（language）是某个给定字母表上一个任意的可数的串集合。这个定义非常宽泛。根据这个定义，像空集 $\emptyset$ 和仅包含空串的集合{ϵ}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。定义串的“含义”的方法将在第 5 章中讨论。

串的各部分的术语

下面是一些与串相关的常用术语：

1）串 s 的*前缀*（prefix）是从 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。

2）串 s 的*后缀*（suffix）是从 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。

3）串 s 的*子串*（substring）是删除 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。

4）串 s 的*真*（true）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。

5）串 s 的*子序列*（subsequence）是从 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。

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 后面而形成的

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，$s\epsilon = \epsilon s = s$。

如果把两个串的连接看成是这两个串的“乘积”，我们可以定义串的“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 $\epsilon s = s$，由此可知 $s^1 = s$，$s^2 = ss$，$s^3 = sss$，依此类推。

3.3.2 语言上的运算

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。图 3-6 给出了这些运算的正式定义。并运算是常见的集合运算。语言的连接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{ϵ}，并且 L^i 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^0。也就是说，除非 ϵ 属于 L，否则 ϵ 不属于 L^+。

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	$L^* = \cup_{i=0}^{\infty} L^i$
L的正闭包	$L^* = \cup_{i=1}^{\infty} L^i$

图 3-6 语言上的运算的定义

例 3.3 令 L 表示字母的集合{A, B, …, Z, a, b, …, z}，令 D 表示数位的集合{0, 1, …, 9}。我们可以用两种不同但等价的方式来考虑 L 和 D。一种方法是将 L 看成是大、小写字母组成的字母表，将 D 看成是 10 个数位组成的字母表。另一种方法是将 L 和 D 看作语言，它们的所有串的长度都为一。下面是一些根据图 3-6 中的运算符从 L 和 D 构造得到的新语言：

1) $L \cup D$ 是字母和数位的集合——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位。

2) LD 是包含 520 个长度为 2 的串的集合，每个串都是一个字母跟一个数位。

3) L^4 是所有由四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 ϵ。

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

6) D^+ 是由一个或多个数位构成的串的集合。

□

3.3.3 正则表达式

假设我们要描述 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 的标识符中可以包括下划线。

在例 3.3 中，我们可以首先给出字母和数位集合的名字，然后使用并、连接和闭包这些运算符来描述标识符。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的表示方法来描述语言。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 _来表示任一字母或下划线，用 digit _来表示数位，那么可以使用如下的正则表达式来描述对应于 C 语言标识符的语言：

$$letter\_(letter\_ \mid digit)^*$$

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“零个或多个”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 Σ 上的正则表达式以及这些表达式所表示的语言。

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，$L(\epsilon) = \{\epsilon\}$，即该语言只包含空串。

2）如果 a 是 Σ 上的一个符号，那么 **a** 是一个正则表达式，并且 $L(\mathbf{a}) = \{a\}$。也就是说，这个语言仅包含一个长度为 1 的符号串 a。请注意，根据惯例，我们通常用斜体表示符号，粗体表示它们所对应的正则表达式。[⊖]

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四个部分。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 $L(r)$。最后这个规则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。如果我们采用如下的约定，就可以丢掉一些括号：

1）一元运算符 * 具有最高的优先级，并且是左结合的。

2）连接具有次高的优先级，它也是左结合的。

3）| 的优先级最低，并且也是左结合的。

例如，我们可以根据这个约定将 $(\mathbf{a})|((\mathbf{b})^*(\mathbf{c}))$ 改写为 $\mathbf{a}|\mathbf{b}^*\mathbf{c}$。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 $\Sigma = \{a, b\}$。

1）正则表达式 $\mathbf{a}|\mathbf{b}$ 表示语言 $\{\mathbf{a}, \mathbf{b}\}$。

2）正则表达式 $(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$ 表示语言 $\{aa, ab, ba, bb\}$，即在字母表 Σ 上长度为 2 的所有串的集合。可表示同样语言的另一个正则表达式是 $\mathbf{aa}|\mathbf{ab}|\mathbf{ba}|\mathbf{bb}$。

3）正则表达式 $\mathbf{a}^*$ 表示所有由零个或多个 a 组成的串的集合，即 $\{\epsilon, a, aa, aaa, \cdots\}$。

4）正则表达式 $(\mathbf{a}|\mathbf{b})^*$ 表示由零个或多个 a 或 b 的实例构成的串的集合，即由 a 和 b 构成的所有串的集合 $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \cdots\}$。另一个表示相同语言的正则表达式是 $(\mathbf{a}^*\mathbf{b}^*)^*$。

5）正则表达式 $\mathbf{a}|\mathbf{a}^*\mathbf{b}$ 表示语言 $\{a, b, ab, aab, aaab, \cdots\}$，也就是串 a 和以 b 结尾的零个或多个 a 组成的串的集合。 □

可以用一个正则表达式定义的语言叫做正则集合（regular set）。如果两个正则表达式 r 和 s 表示同样的语言，则称 r 和 s 等价（equivalent），记作 $r = s$。例如，$(\mathbf{a}|\mathbf{b}) = (\mathbf{b}|\mathbf{a})$。正则表达式遵守一些代数定律，每个定律都断言两个具有不同形式的表达式等价。图 3-7 给出了一些对于任意正则表达式 r、s 和 t 都成立的代数定律。

⊖ 然而，当讨论 ASCII 字符集中的特定字符时，我们通常将使用电传字体同时表示字符和它的正则表达式。

定律	描述
$r\|s = s\|r$	|是可以交换的
$r\|(s\|t) = (r\|s)\|t$	|是可结合的
$r(st) = (rs)t$	连接是可结合的
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	连接对|是可分配的
$\epsilon r = r\epsilon = r$	ϵ是连接的单位元
$r^* = (r\|\epsilon)^*$	闭包中一定包含ϵ
$r^{**} = r^*$	*具有幂等性

图 3-7 正则表达式的代数定律

3.3.4 正则定义

为方便表示，我们可能希望给某些正则表达式命名，并在之后的正则表达式中像使用符号一样使用这些名字。如果 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：

$$\begin{aligned} d_1 &\rightarrow r_1 \\ d_2 &\rightarrow r_2 \\ &\cdots \\ d_n &\rightarrow r_n \end{aligned}$$

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。

我们限制每个 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，并且我们可以为每个 r_i 构造出只包含 Σ 中符号的正则表达式。我们可以首先将 r_2(它不能使用 d_1 之外的任何 d)中的 d_1 替换为 r_1，然后再将 r_3 中的 d_1 和 d_2 替换为 r_1 和(替换之后的)r_2，依此类推。最后，我们将 r_n 中的 $d_i(i=1, 2, \cdots, n-1)$ 替换为 r_i 的经替换后的版本，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、数字和下划线组成的串。下面是 C 标识符对应的语言的一个正则定义。我们将按照惯例用斜体字来表示正则定义中定义的符号。

$$\begin{aligned} \textit{letter\_} &\rightarrow \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\ \textit{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ \textit{id} &\rightarrow \textit{letter\_} \ (\ \textit{letter\_} \mid \textit{digit}\ )^* \end{aligned}$$

□

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规约：

$$\begin{aligned} \textit{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ \textit{digits} &\rightarrow \textit{digit}\ \textit{digit}^* \\ \textit{optionalFraction} &\rightarrow \texttt{.}\ \textit{digits} \mid \epsilon \\ \textit{optionalExponent} &\rightarrow (\ \texttt{E}\ (\ \texttt{+} \mid \texttt{-} \mid \epsilon\ )\ \textit{digits}\ ) \mid \epsilon \\ \textit{number} &\rightarrow \textit{digits}\ \textit{optionalFraction}\ \textit{optionalExponent} \end{aligned}$$

在这个定义中，*optionalFraction* 要么是空串，要么是小数点后再跟一个或多个数位。*optionalExponent* 如果不是空串，就是字母 E 后跟随一个可选的 + 号或 - 号，再跟上一个或多个数位。请注意，小数点后至少要跟一个数位，所以 *number* 和 `1.` 不匹配，但和 `1.0` 匹配。 □

3.3.5 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 Kleene 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，我们介绍的一些最早出现在像 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规约中非常有用。本章的参考文献中包含了一个对当今仍在使用的正则表达式变体的讨论。

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 + 和运算符 * 具有同样的优先级和结合性。两个有用的代数定律 $r^* = r^+ | \epsilon$ 和 $r^+ = rr^* = r^* r$ 说明了 Kleene 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 $r|\epsilon$，换句话说，$L(r?) = L(r) \cup \{\epsilon\}$。运算符？与运算符 + 和运算符 * 具有同样的优先级和结合性。

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

例 3.7 根据这些缩写表示法，我们可以将例 3.5 中的正则定义改写为：

$$\begin{aligned} letter\_ &\rightarrow [\texttt{A-Za-z\_}] \\ digit &\rightarrow [\texttt{0-9}] \\ id &\rightarrow letter\_ \ (\ letter\_ \mid digit\ )^* \end{aligned}$$

例 3.6 的正则定义可以简化为：

$$\begin{aligned} digit &\rightarrow [\texttt{0-9}] \\ digits &\rightarrow digit^+ \\ number &\rightarrow digits\ (\ .\ digits)?\ (\ \texttt{E}\ [\texttt{+-}]?\ digits\ )? \end{aligned}$$

□

3.3.6 3.3 节的练习

练习 3.3.1：对于下列各个语言，查询语言使用手册以确定：(i)形成各语言的输入字母表的字符集分别是什么(不包括那些只能出现在字符串或注释中的字符)？(ii)各语言的数字常量的词法形式是什么？(iii)各语言的标识符的词法形式是什么？

(1) C (2) C++ (3) C# (4) Fortran (5) Java (6) Lisp (7) SQL

! **练习 3.3.2**：试描述下列正则表达式定义的语言：

1) **a**(**a**|**b**)* **a**

2) ((ϵ|**a**)**b***)*

3) (**a**|**b**)* **a**(**a**|**b**)(**a**|**b**)

4) **a*** **ba*** **ba*** **ba***

!! 5) (**aa**|**bb**)* ((**ab**|**ba**)(**aa**|**bb**)* (**ab**|**ba**)(**aa**|**bb**)*)*

练习 3.3.3：试说明在一个长度为 n 的字符串中，分别有多少个

1) 前缀

2) 后缀

3) 真前缀

! 4) 子串

！5）子序列

练习3.3.4：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来表示大小写不敏感的语言中的关键字。给出描述SQL语言中的关键字“select”的表达式，以说明你的思想。

！**练习3.3.5**：试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）所有不重复的数位组成的串。提示：首先尝试解决只含有少量数位(比如{0，1，2})的数位串。

！！5）所有最多只有一个重复数位的串。

！！6）所有由偶数个a和奇数个b构成的串。

7）以非正式方式表示的国际象棋的步法的集合，如p-k4或kbp×qn。

！！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。

练习3.3.6：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~j)，包括大写和小写。

2）所有小写的辅音字母的集合。

3）十六进制中的“数位”(对大于9的数位，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(比如感叹号)。

从下面开始直到练习3.3.10(含)讨论了来自Lex的正则表达式的扩展表示方法(我们将在3.5节中讨论这个词法分析器生成工具)。这些扩展表示方法在图3-8中列出。

表达式	匹配	例子
c	单个非运算符字符c	a
\c	字符c的字面值	*
"s"	串s的字面值	"**"
.	除换行符以外的任何字符	a.*b
^	一行的开始	^abc
$	行的结尾	abc$
[s]	字符串s中的任何一个字符	[abc]
[^s]	不在串s中的任何一个字符	[^abc]
r*	和r匹配的零个或多个串连接成的串	a*
r+	和r匹配的一个或多个串连接成的串	a+
r?	零个或一个r	a?
$r\{m,n\}$	最少m个，最多n个r的重复出现	a{1,5}
r_1r_2	r_1后加上r_2	ab
$r_1 \mid r_2$	r_1或r_2	a|b
(r)	与r相同	(a|b)
r_1/r_2	后面跟有r_2时的r_1	abc/123

图3-8 Lex的正则表达式

练习 3.3.7：请注意这些正则表达式中的下列字符(称为运算符字符)都具有特殊的含义：

\ " . ^ $ [ ] * + ? { } | /

如果想要使得这些特殊字符在一个串中表示它们自身，就必须取消它们的特殊含义。我们将它们放在一个长度大于等于 1 且加上双引号的串中就可以取消特殊含义。例如，正则表达式“**”和字符串 ** 匹配。我们也可以在一个运算符字符前加一个反斜线，得到这个字符的字面含义。那么，正则表达式 * * 也和串 ** 匹配。请写出一个和字符串"\匹配的正则表达式。

练习 3.3.8：在 Lex 中，*补集字符类*(complemented character class)代表该字符类中列出的字符之外的所有字符。我们将^放在开头来表示一个补集字符类。除非^在该字符类内列出，否则这个字符不在被取补的字符类中。因此，[^A-Za-z]匹配所有不是大小写字母的字符，[^\^]匹配除^(以及换行符，因为它不在任何字符类中)之外的任何字符。试证明：对于每个带有补集字符类的正则表达式，都存在一个等价的不含补集字符类的正则表达式。

！练习 3.3.9：正则表达式 $r\{m, n\}$ 和模式 r 的 m 到 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。试证明：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。

！练习 3.3.10：运算符^匹配一行的最左端，$匹配一行的最右端。运算符^也被用作补集字符类的首字符，但是通过上下文总是能够确定它的含义。例如，^[^aeiou]*$匹配任何一个不包含小写元音字符的行。

1）你怎样判断^到底表示哪一个意思？

2）是否总是能够将一个包括^和$运算符的正则表达式替换为一个等价的不包含这些运算符的正则表达式？

！练习 3.3.11：UNIX 的 shell 命令 sh 在文件名表达式中使用图 3-9 中的运算符来描述文件名的集合。例如，文件名表达式 `*.o` 和所有以 `.o` 结束的文件名匹配；`sort1.?` 和所有形如 `sort1.c` 的文件名匹配，其中 c 可以是任何字符。试问如何使用只包含并、连接和闭包运算符的正则表达式来表示 sh 文件名表达式？

表达式	匹配	例子
's'	串 *s* 的字面值	'\'
\c	字符 *c* 的字面值	\'
*	任何串	*.o
?	任何字符	sort1.?
[*s*]	*s* 中的确任何字符	sort1.[cso]

图 3-9 shell 命令 sh 使用的文件名表达式

！练习 3.3.12：SQL 语言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：下划线(_)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(比如 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。假设我们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。

3.4 词法单元的识别

上一节介绍了如何使用正则表达式来表示一个模式。现在，我们必须学习如何根据各个需要识别的词法单元的模式来构造出一段代码。这段代码能够检查输入字符串，并在输入的前缀

中找出一个和某个模式匹配的词素。我们的讨论将围绕下面的例子展开。

例 3.8 图 3-10 的文法片段描述了分支语句和条件表达式的一种简单形式。这个语法和 Pascal 语言的语法类似，它的 **then** 关键字显式地出现在条件表达式的后面。对于 **relop**，我们使用 Pascal 或 SQL 语言中的比较运算符，其中 = 表示“相等”，< > 表示“不相等”，因为它们呈现了一种有意思的词素结构。

在考虑词法分析器时，文法的终结符号，包括 **if**、**then**、**else**、**relop**、**id** 及 **number**，都是词法单元的名字。这些词法单元的模式使用图 3-11 中的正则定义来描述。其中 *id* 和 *number* 的模式和我们之前在例 3.7 中看到的模式类似。

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	ϵ
expr	→	*term* **relop** *term*
	|	*term*
term	→	**id**
	|	**number**

图 3-10 分支语句的文法

digit	→	[0-9]
digits	→	$digit^+$
number	→	*digits* (. *digits*)? (E [+-]? *digits*)?
letter	→	[A-Za-z]
id	→	*letter* (*letter* | *digit*)*
if	→	if
then	→	then
else	→	else
relop	→	< | > | <= | >= | = | <>

图 3-11 例 3.8 中词法单元的模式

对这个语言，词法分析器将识别关键字 *if*、*then*、*else* 以及和 *relop*、*id* 和 *num* 的模式匹配的词素。为了简化问题，我们做出如下的常见假设：关键字也是保留字。也就是说，它们不是标识符，虽然它们的词素和标识符的模式匹配。

此外，我们还让词法分析器负责消除空白符，方法是让它识别如下定义的“词法单元”*ws*。

$$ws \rightarrow (\ \textbf{blank} \mid \textbf{tab} \mid \textbf{newline}\ )^+$$

这里，**blank**、**tab** 及 **newline** 是用于表示具有同样名字的 ASCII 字符的抽象符号。词法单元 *ws* 同其他的词法单元的不同之处在于：当我们识别到 *ws* 时，我们并不将它返回给语法分析器，而是从这个空白之后的字符开始继续进行词法分析。返回给语法分析器的是下一个词法单元。

图 3-12 总结了词法分析器的目标。对于各个词素或词素的集合，该表显示了应该将哪个词法单元名返回给语法分析器，以及按照 3.1.3 节中的介绍，应该返回什么属性值。请注意，对于其中的 6 个关系运算符，符号常量 LT、LE 等被当作属性值返回，其目的是指明我们发现的是词法单元 **relop** 的哪个实例。找到的运算符将影响编译器输出的代码。 □

词素	词法单元名字	属性值
Any *ws*	–	–
if	**if**	–
then	**then**	–
else	**else**	–
Any *id*	**id**	指向符号表条目的指针
Any *number*	**number**	指向符号表条目的指针
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

图 3-12 词法单元、它们的模式以及属性值

3.4.1 状态转换图

作为构造词法分析器的一个中间步骤，我们首先将模式转换成具有特定风格的流图，称为“状态转换图”。在本节中，我们用手工方式将正则表达式表示的模式转化为状态转换图，在 3.6 节中，我们将看到可以使用自动化的方法根据一组正则表达式集合构造出状态转换图。

状态转换图(transition diagram)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。我们可以将一个状态看作是对我们已经看到的位于 *lexemeBegin* 指针和 *forward* 指针之间的字符的总结，它包含了我们在进行词法分析时需要的全部信息。

状态图中的*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。如果我们处于某个状态 s，并且下一个输入符号是 a，我们就会寻找一条从 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)。如果我们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。我们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。从 3.5 节开始，我们将放松对确定性的要求，令词法分析器的设计者更加容易完成任务，但同时提高了对实现者的技巧要求。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。我们用双层的圈来表示一个接受状态，并且如果该状态要执行一个动作的话——通常是向语法分析器返回一个词法单元和相关属性值——我们将把这个动作附加到该接受状态上。

2) 另外，如果需要将 *forward* 回退一个位置(即相应的词素并不包含那个在最后一步使我们到达接受状态的符号)，那么我们将在该接受状态的附近加上一个 * 。我们的例子都不需要将 *forward* 指针回退多个位置，但万一出现这种情况，我们将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。

例 3.9 图 3-13 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。我们从初始状态 0 开始。如果我们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，我们只能选择 < 、<> 或 <= 。因此我们进入状态 1 并查看下一个字符。如果这个字符是 =，我们识别出词素 <=，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，下一个字符是 >，那么我们就会得到词素 <>，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 <，我们进入状态 4 并向语法分析器返回这个信息。请注意，状态 4 有一个 * 号，说明我们必须将输入回退一个位置。

另一方面，如果在状态 0 时我们看到的第一个字符是 =，那么这个字符必定是要识别的词素。我们立即从状态 5 返回这个信息。其余的可能性是第一个字符为 > 的情况。那么我们应该进入状态 6，并根据下一字符确定词素是 >= (如果我们看到下一个字符为 =)还是 > (对于任何其他字符)。注意，如果在状态 0 时我们看到的是不同于 < 、= 或 > 的字符，我们就不可能看到一个 **relop** 的词素，因此这个状态转换图将不会被使用。 □

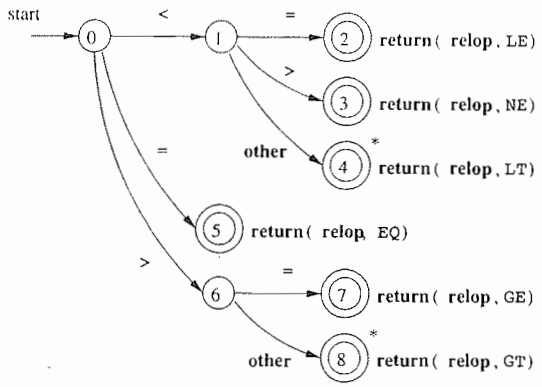


图 3-13 词法单元 **relop** 的状态转换图

3.4.2 保留字和标识符的识别

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（在我们正在使用的例子中就是如此），因此虽然它们看起来很像标识符，但它们不是标识符。因此，尽管我们通常使用如图 3-14 所示的状态转换图来寻找标识符的词素，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`。

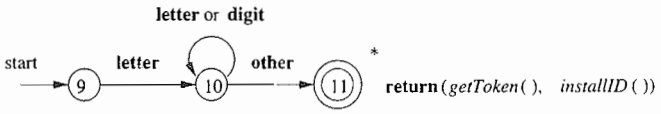


图 3-14 **id** 和关键字的状态转换图

我们可以使用两种方法来处理那些看起来很像标识符的保留字：

1）初始化时就将各个保留字填入符号表中。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。我们已经假设图 3-14 中使用了这种方法。当我们找到一个标识符时，如果该标识符尚未出现在符号表中，就会调用 *installID* 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保留字，因此它的词法单元是 **id**。函数 *getToken* 查看对应于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到符号表中的关键字词法单元。

2）为每个关键字建立单独的状态转换图。图 3-15 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，否则在碰到词素像 `thenextvalue` 这样以 `then` 为前缀的 **id** 词法单元时，我们可能会错误地返回词法单元 `then`。如果采用这个方法，我们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，优先识别保留字词法单元，而不是 **id** 词法单元。我们并没有在例子中使用这个方法，这也是我们没有对图 3-15 中的状态进行编号的原因。

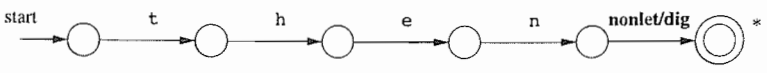


图 3-15 假想的关键字 `then` 的状态转换图

3.4.3 完成我们的例子

我们在图 3-14 中看到，**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，如果是的话进入状态 10。只要接下来的输入包含字母或数位，我们就一直停留在状态 10。当我们第一次遇到不是字母或数位的其他任何字符时，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，我们必须将输入回退一个位置，并且如 3.4.2 节所讨论的那样，我们将已经找到的标识符加入到符号表中，并判断我们得到的究竟是一个关键字还是一个真正的标识符。

图 3-16 显示了词法单元 **number** 的状态转换图，它是我们至今为止看到的最复杂的状态转换图。从状态 12 开始，如果我们看到一个数位，就转入状态 13。在该状态，我们可以读入任意数量的其他数位。然而，如果我们看到了一个不是数位、不是小数点，也不是 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，我们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转换图中显示出来，但它们和我们处理标识符的方法相似。

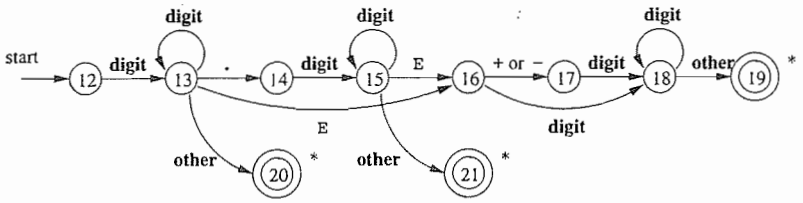


图 3-16 无符号数字的状态转换图

如果我们在状态 13 看到的是一个小数点，那么我们就看到一个“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位，状态 15 就被用于此目的。如果我们看到一个 E，那么我们就看到了一个“可选的指数部分”，它的识别任务由状态 16 ~ 19 完成。如果我们在状态 15 看到的是不同于 E 和数位的其他字符，那么我们就到达了小数部分的结尾，这个数字没有指数部分，我们将通过状态 21 返回刚刚找到的词素。

最后一个状态转换图显示在图 3-17 中，它用于识别空白符。在该图中，我们寻找一个或多个空白字符，在图中用 **delim** 表示。典型的空白字符有空格、制表符、换行符，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。

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

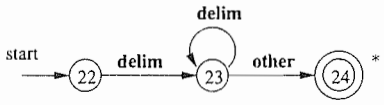


图 3-17 空白符的状态转换图

3.4.4 基于状态转换图的词法分析器的体系结构

有几种方法可以根据一组状态转换图构造出一个词法分析器。不管整体的策略是什么，每个状态总是对应于一段代码。我们可以想象有一个变量 `state` 保存了一个状态转换图的当前状态的编号。有一个 switch 语句根据 `state` 的值将我们转到对应于各个可能状态的相应代码段，我们可以在那里找到该状态需要执行的动作。一个状态的代码本身常常也是一条 switch 语句或多路分支语句。这个语句读入并检查下一个输入字符，由此确定下一个状态。

例 3.10 在图 3-18 中，我们可以看到 getRelop()方法的一个概述。它是一个 C++函数，其任务是模拟图 3-13 中的状态转换图，并返回一个 TOKEN 类型的对象。该对象由一个词法单元名(在该例中必定是 **relop**)和一个属性值(在该例中是 6 个比较运算符之一的编码)组成。函数 getRelop()首先创建一个新的对象 retToken，并将该对象的第一个分量初始化为 RELOP，即词法单元 **relop** 的编码。

在 case 0 中，我们可以看到一个典型的状态行为。函数 nextChar( )从输入中获取下一个字符，并将它赋给局部变量 c。然后我们检查 c 是否为我们期望找到的三个字符，并在每种情况下根据图 3-13 所示的状态转换图完成状态转换。例如，如果下一输入字符是 =，那么就转换到状态 5。

如果下一个输入字符不是某个比较运算符的首字符，getRelop()就会调用函数 fail()。函数 fail()的具体操作依赖于词法分析器的全局错误恢复策略。它应该将 forward 指针重置为 lexemeBegin 的值，使得我们可以使用另一个状态转换图从尚未处理的输入部分的真实开始位置开始识别。然后，它还需要将变量 state 的值改为另一状态转换图的初始状态，该转换图将寻找另一个词法单元。在另一种情况下，如果所有的转换图都已经用过，则 fail()可以启动一个错误纠正步骤，按照 3.1.4 节中讨论的方法来纠正输入并找到一个词素。

在图 3-18 中，我们还展示了状态 8 的行为。由于状态 8 带有一个 * 号，我们必须将输入指针回退一个位置(也就是把 c 放回输入流)。该任务由函数 retract()完成。因为状态 8 代表了对词素 > 的识别，我们把返回对象中的第二个分量设置成 GT，即这个运算符的编码。我们假设这个分量的名字是 attribute。 □

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

图 3-18 **relop** 的转换图的概要实现

为了在适当的地方模拟适当的状态转换图，我们考虑几种将如图 3-18 所示的代码集成到整个词法分析器中的方法。

1) 我们可以让词法分析器顺序地尝试各个词法单元的状态转换图。然后，在每次调用例 3.10 中的函数 fail()时，它重置 forward 指针并启动下一个状态转换图。这个方法使我们可以像图 3-15 中所建议的那样，为各个关键字使用各自的状态转换图。我们只需要在使用 **id** 的状

态转换图之前使用这些关键字的转换图，就可以使得关键字被识别为保留字。

2）我们可以“并行地”运行各个状态转换图，将下一个输入字符提供给所有的状态转换图，并使得每个状态转换图作出它应该执行的转换。如果我们采用这个策略，就必须谨慎地解决如下的问题：一个状态转换图已经找到了一个与它的模式相匹配的词素，但另外的一个或多个状态转换图仍然可以继续处理输入。解决这个问题的常见策略是取最长的和某个模式相匹配的输入前缀。举例来说，该规则让我们识别出标识符 **thenext** 而不是关键字 **then**，识别出 -> 而不是 - 。

3）有一个更好的方法，也是我们将在下面各节中采用的方法，就是将所有的状态转换图合并为一个图。我们允许合并后的状态转换图尽量读取输入，直到不存在下一个状态为止；然后像上面的2 中讨论的那样取最长的和某个模式匹配的最长词素。在我们的例子中，进行这种合并很简单，因为没有两个词法单元以相同的字符开头。也就是说，根据第一个字符就可以知道我们正在寻找的是哪个词法单元。因此，我们可以直接将状态 0、9、12 及 22 合并成一个开始状态，并保持其他转换不变。但一般而言，正如我们不久将看到的那样，合并几个词法单元的状态转换图的问题会更加复杂。

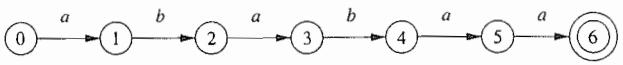
3.4.5 3.4 节的练习

练习 3.4.1：给出识别练习 3.3.2 中各个正则表达式所描述的语言的状态转换图。

练习 3.4.2：给出识别练习 3.3.5 中各个正则表达式所描述的语言的状态转换图。

从下面的练习开始到练习 3.4.12 介绍了 Aho-Corasick 算法。该算法可以在文本串中识别一组关键字，所需时间和文本长度以及所有关键字的总长度成正比。该算法使用了一种称为“trie”的特殊形式的状态转换图。trie 是一个树型结构的状态转换图，从一个结点到它的各个子结点的边上有不同的标号。Trie 的叶子结点表示识别到的关键字。

Knuth、Morris 和 Pratt 提出了一种在文本串中识别单个关键字 $b_1b_2\cdots b_n$ 的算法。这里的 trie 是一个包含了从 0 ~ n 共 $n+1$ 个状态的状态转换图。状态 0 是初始状态，状态 n 表示接受，也就是发现关键字的情形。从 0 到 $n-1$ 之间的任意一个状态 s 出发，存在一个标号为 b_{s+1} 的到达状态 $s+1$ 的转换。例如，关键字 ababaa 的 trie 树为：



为了快速处理文本串并在这些串中搜索一个关键字，针对关键字 $b_1b_2\cdots b_n$ 以及该关键字中的位置 s（对应于关键字的 trie 中的状态 s）定义*失效函数* $f(s)$，该函数的计算方法如图 3-19 所示。

该函数的目标是使得 $b_1b_2\cdots b_{f(s)}$ 是最长的既是 $b_1b_2\cdots b_s$ 的真前缀又是 $b_1b_2\cdots b_s$ 的后缀的子串。$f(s)$ 之所以重要，原因在于如果我们试图用一个文本串匹配 $b_1b_2\cdots b_n$，并且我们已经匹配了前 s 个位置，但此时匹配失败（也就是说文本串的下一个位置并不是 b_{s+1}），那么 $f(s)$ 就是可能和以我们的当前位置为结尾的文本串相匹配的最长的 $b_1b_2\cdots b_n$ 的前缀。当然，文本串的下一个字符必须是 $b_{f(s)+1}$，否则仍然有问题，必须考虑一个更短的前缀，即 $b_{f(f(s))}$。

看一个例子，根据 ababaa 构造的 trie 的失效函数是：

s	1	2	3	4	5	6
$f(s)$	0	0	1	2	3	1

例如，状态 3 和 1 分别表示前缀 aba 以及 a。因为 a 是最长的既是 aba 的真前缀，同时也是 aba 的后缀的串，因此 $f(3)=1$。同样，因为最长的既是 ab 的真前缀又是它的后缀的字符串是空串，因此 $f(2)=0$。

练习 3.4.3：构造下列串的失效函数。

1) abababaab

2) aaaaaa

3) abbaabb

```
1)  t = 0;
2)  f(1) = 0;
3)  for (s = 1; s < n; s + +) {
4)          while (t > 0 && b_{s+1} != b_{t+1}) t = f(t);
5)          if (b_{s+1} == b_{t+1}) {
6)                  t = t + 1;
7)                  f(s + 1) = t;
            }
8)          else f(s + 1) = 0;
    }
```

图 3-19 计算关键字 $b_1b_2\cdots b_n$ 的失效函数的算法

! **练习 3.4.4**：对 s 进行归纳，证明图 3-19 的算法正确地计算出了失效函数。

!! **练习 3.4.5**：说明图 3-19 中第 4 行的赋值语句 $t=f(t)$ 最多被执行 n 次。进而说明整个算法的时间复杂度是 $O(n)$，其中 n 是关键字的长度。

计算得到关键字 $b_1b_2\cdots b_n$ 的失效函数之后，我们就可以在 $O(m)$ 时间内扫描字符串 $a_1a_2\cdots a_m$ 以判断该关键字是否出现在其中。图 3-20 中所展示的算法使关键字沿着被匹配字符串滑动，不断尝试将关键字的下一个字符与被匹配字符串的下一个字符匹配，逐步推进。如果在匹配了 s 个字符后无法继续匹配，那么该算法将关键字“向右滑动”$s-f(s)$ 个位置，也就是认为只有该关键字的前 $f(s)$ 个字符和被匹配字符串匹配。

练习 3.4.6：应用 KMP 算法判断关键字 ababaa 是否为下面字符串的子串：

1) abababaab

2) abababbaa

!! **练习 3.4.7**：说明图 3-20 中的算法可以正确地指出输入关键字是否为一个给定字符串的子串。*提示*：对 i 进行归纳。说明对于所有的 i，在第四行运行后 s 的值是那些既是 $a_1a_2\cdots a_i$ 的后缀又是该关键字的前缀的字符串中最长字符串的长度。

```
1)  s = 0;
2)  for (i = 1; i ⩽ m; i++) {
3)          while (s > 0 && a_i != b_{s+1}) s = f(s);
4)          if (a_i == b_{s+1}) s = s + 1;
5)          if (s == n) return "yes";
    }
6)  return "no";
```

图 3-20 KMP 算法在 $O(m+n)$ 时间内检测字符串 $a_1a_2\cdots a_m$ 中是否包含单个关键字 $b_1b_2\cdots b_n$

!! **练习 3.4.8**：假设已经计算得到函数 f 且它的值存储在一个以 s 为下标的数组中，说明图 3-20 中算法的时间复杂度为 $O(m+n)$。

练习 3.4.9：*Fibonacci* 字符串的定义如下：

1) s_1 = b。

2) s_2 = a。

3) 当 $k>2$ 时，$s_k=s_{k-1}s_{k-2}$。

例如，s_3 = ab，s_4 = aba，s_5 = abaab。

1）s_n 的长度是多少？

2）构造 s_6 的失效函数。

3）构造 s_7 的失效函数。

!! 4）说明任何 s_n 的失效函数都可以被表示为：$f(1)=f(2)=0$，且对于 $2<j\leqslant|s_n|$，$f(j)=j-|s_{k-1}|$，其中 k 是使得 $|s_k|\leqslant j+1$ 的最大的整数。

!! 5）在 KMP 算法中，当我们试图确定关键字 s_k 是否出现在字符串 s_{k+1} 中时，最多会连续多少次调用失效函数？

Aho 和 Corasick 对 KMP 算法进行了推广，使它可以在一个文本串中识别一个关键字集合中的任何关键字。在这种情况下，trie 是一棵真正的树，从其根结点开始就会出现分支。如果一个字符串是某个关键字的前缀（不一定是真前缀），那么在 trie 中就有一个和该字符串对应的状态。串 $b_1b_2\cdots b_{k-1}$ 对应的状态是串 $b_1b_2\cdots b_k$ 对应的状态的父结点。如果一个状态对应于某个完整的关键字，那么该状态就是接受状态。例如，图 3-21 显示了对应于关键字 he、she、his 和 hers 的 trie 树。

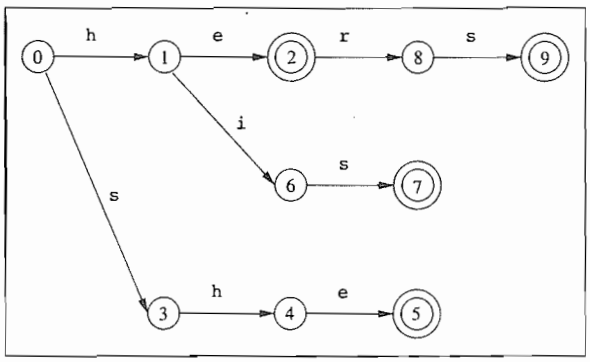


图 3-21 关键字 he、she、his 和 hers 的 trie 树

通用 trie 树的失效函数的定义如下。假设 s 是对应于串 $b_1b_2\cdots b_n$ 的状态，那么状态 $f(s)$ 对应于最长的、既是串 $b_1b_2\cdots b_n$ 的后缀又是某个关键字的前缀的字符串。例如，图 3-21 中 trie 树的失效函数为：

s	1	2	3	4	5	6	7	8	9
$f(s)$	0	0	0	1	2	0	3	0	3

! **练习 3.4.10**：修改图 3-19 中的算法，使它可以计算通用 trie 树的失效函数。提示：主要的不同在于，在图 3-19 的第 4、5 行上，我们不能简单地测试 b_{s+1} 和 b_{t+1} 是否相等。从任何一个状态出发，都可能存在多个在不同字符上的转换。比如在图 3-21 中，存在从状态 1 出发、分别在字符 e 和 i 上的两个转换。这些转换都可能进入代表了最长的既是后缀又是前缀的字符串的状态。

练习 3.4.11：为下面的关键字集合构造 trie 以及失效函数。

1）aaa、abaaa 和 ababaaa。

2）all、fall、fatal、llama 和 lame。

3）pipe、pet、item、temper 和 perpetual。

! **练习 3.4.12**：说明练习 3.4.10 中所设计的算法的运行时间和所有关键字长度的总和成线

性关系。

3.5 词法分析器生成工具 Lex

在本节中，我们将介绍一个名为 Lex 的工具。在最近的实现中它也称为 Flex。它支持使用正则表达式来描述各个词法单元的模式，由此给出一个词法分析器的规约。Lex 工具的输入表示方法称为 Lex 语言(Lex language)，而工具本身则称为 Lex 编译器(Lex compiler)。在它的核心部分，Lex 编译器将输入的模式转换成一个状态转换图，并生成相应的实现代码，并存放到文件 lex.yy.c 中。这些代码模拟了状态转换图。如何将正则表达式翻译为状态转换图是下一节讨论的主题，这里我们只学习 Lex 语言。

3.5.1 Lex 的使用

Lex 的使用方法如图 3-22 所示。首先，用 Lex 语言写出一个输入文件，描述将要生成的词法分析器。在图中这个输入文件称为 lex.l。然后，Lex 编译器将 lex.l 转换成 C 语言程序，存放该程序的文件名总是 lex.yy.c。最后，文件 lex.yy.c 总是被 C 编译器编译为一个名为 a.out 的文件。C 编译器的输出就是一个读取输入字符流并生成词法单元流的可运行的词法分析器。

编译后的 C 程序，在图 3-22 中被称为 a.out，通常是一个被语法分析器调用的子例程，这个子例程返回一个整数值，即可能出现的某个词法单元名的编码。而词法单元的属性值，不管它是一个数字编码，还是一个指向符号表的指针，或者什么都没有，都保存在全局变量 yylval 中[⊖]。这个变量由词法分析器和语法分析器共享。这么做可以同时返回一个词法单元名字和一个属性值。

3.5.2 Lex 程序的结构

一个 Lex 程序具有如下形式：

```
声明部分
%%
转换规则
%%
辅助函数
```

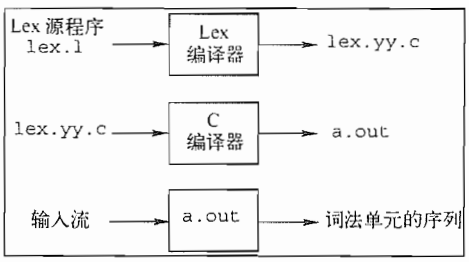


图 3-22 用 Lex 创建一个词法分析器

声明部分包括变量和明示常量(manifest constant，被声明的表示一个常数的标识符，如一个词法单元的名字)的声明和 3.3.4 节中描述的正则定义。

Lex 程序的每个转换规则具有如下形式：

模式 { 动作 }

其中，每个模式是一个正则表达式，它可以使用声明部分中给出的正则定义。动作部分是代码片段。虽然人们已经创建了很多能使用其他语言的 Lex 的变体，但这些代码片段通常是用 C 语言编写的。

Lex 程序的第三个部分包含各个动作需要使用的所有辅助函数。还有一种方法是将这些函数单独编译，并与词法分析器的代码一起装载。

由 Lex 创建的词法分析器和语法分析器按照如下方式协同工作。当词法分析器被语法分析器调用时，词法分析器开始从余下的输入中逐个读取字符，直到它发现了最长的与某个模式 P_i

⊖ 顺便说一下，在 yylval 和 lex.yy.c 中出现的 yy 指的是我们将在 4.9 节中讨论的语法分析器生成工具 yacc，它一般和 Lex 一起使用。

匹配的前缀。然后，词法分析器执行相关的动作 A_i。通常 A_i 会将控制返回给语法分析器。然而，如果它不返回控制(比如 P_i 描述的是空白符或注释)，那么词法分析器就继续寻找其他的词素，直到某个动作将控制返回给语法分析器为止。词法分析器只向语法分析器返回一个值，即词法单元名。但在需要时可以利用共享的整型变量 yylval 传递有关这个词素的附加信息。

例 3.11 图 3-23 是一个 Lex 程序，它能够识别图 3-12 中的各个词法单元，并返回找到的词法单元。观察这段代码可以发现 Lex 的很多重要特点。

我们在声明部分看到一对特殊的括号：%{和%}。出现在括号内的所有内容都被直接复制到文件 lex.yy.c 中。它们不会被当作正则定义处理。我们一般将明示常量的定义放置在该括号内，并利用 C 语言的#define 语句给每个明示常量赋予一个唯一的整数编码。在我们的例子中，我们在一个注释中列出了 LT、IF 等明示常量，但没有显示它们被赋予哪些特定的整数。⊖

在声明部分还包含一个正则定义的序列。这些定义使用了 3.3.5 节中描述的正则表达式的扩展表示方法。那些将在后面的定义中或某个转换规则的模式中使用的正则定义用花括号括起来。例如，*delim* 被定义为表示一个包含了空格、制表符及换行符的字符类的缩写。后两个字符分别用反斜线再跟上 *t* 及 *n* 来表示。这个表示法和 UNIX 命令使用的方法相同。于是，*ws* 通过正则表达式{delim}+定义为一个或多个分隔符组成的序列。

注意，在 *id* 和 *number* 的定义中，圆括号是用于分组的元符号，并不代表圆括号自身。相反，在 *number* 定义中的 E 代表其自身。如果我们希望 Lex 的某个元符号(比如括号、+、*或? 等)表示其自身，我们可以在它们前面加上一个反斜线。例如，我们在 *number* 的定义中看到的\. 就表示小数点本身。在它前面加上反斜线的原因是，和在 UNIX 正则表达式中一样，该字符在 Lex 中是一个代表“任一字符”的元符号。

在辅助函数部分，我们可以看到这样两个函数：installID()和 installNum()。和位于%{…%}中的声明部分一样，出现在辅助部分中的所有内容都被直接复制到文件 lex.yy.c 中。虽然它们位于转换规则部分之后，但这些函数可以在规则部分的动作定义中使用。

最后，让我们看一下图 3-23 的中间部分的一些模式和规则。首先，在第一部分中定义的标识符 *ws* 有一个相关的空动作。如果我们发现了一个空白符，我们并不把它返回给语法分析器，而是继续寻找另一个词素。第二词法单元有一个简单的正则表达式模式 if。如果我们在输入中看到两个字母 if，并且 if 之后没有跟随其他字母或数位(如果有的话，词法分析器会去寻找一个和 **id** 模式匹配的最长输入前缀)，然后词法分析器从输入中读入这两个字符，并返回词法单元名 IF，也就是明示常量 IF 所代表的整数值。关键字 then 和 else 的处理方法与此类似。

第五个词法单元的模式由 *id* 定义。注意，虽然像 if 这样的关键字既和这个模式匹配，也和之前的一个模式匹配，但是当最长匹配前缀和多个模式匹配时，Lex 总是选择最先被列出的模式。当 *id* 被匹配时，相应的处理动作分为三步：

1) 调用函数 installID()将找到的词素放入符号表中。

2) 该函数返回一个指向符号表的指针。这个指针被放到全局变量 yylval 中，并可被语法分析器或编译器的某个后续组件使用。注意，函数 installID()可以使用以下两个由 Lex 生成的、由词法分析器自动赋值的变量：

- yytext 是一个指向词素开头的指针，与图 3-3 中的 lexemeBegin 类似。

⊖ 如果 Lex 同 Yacc 一起使用，那么明示常量通常会在 Yacc 程序中定义，并在 Lex 程序中不加定义就使用它们。因为 lex.yy.c 是和 Yacc 的输出一起编译的，因而这些常量在 Lex 程序的动作中也是可用的。

- yyleng 存放刚找到的词素的长度。

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); return(NUMBER);}
"<"       {yylval = LT; return(RELOP);}
"<="      {yylval = LE; return(RELOP);}
"="       {yylval = EQ; return(RELOP);}
"<>"      {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="      {yylval = GE; return(RELOP);}

%%

int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

图 3-23 识别图 3-12 中的词法单元的 Lex 程序

3）将词法单元名 ID 返回到语法分析器。

当一个词素与模式 *number* 匹配时，执行的处理与此类似，它使用辅助函数 installNum() 完成处理。 □

3.5.3 Lex 中的冲突解决

前面我们已经间接提到了 Lex 解决冲突的两个规则。当输入的多个前缀与一个或多个模式匹配时，Lex 用如下规则选择正确的词素：

1）总是选择最长的前缀。

2）如果最长的可能前缀与多个模式匹配，总是选择在 Lex 程序中先被列出的模式。

例 3.12 第一个规则告诉我们，要持续读入字母和数位，寻找最长的由这些字符组成的前缀并将它们组合成为一个标识符。它也告诉我们应该将 <= 看成是一个词素，而不是将 < 看作一个词

素、再将 = 看作下一个词素。如果我们在 Lex 程序中将关键字的模式置于 **id** 的模式之前，那么第二个规则将使得关键字成为保留字。例如，如果 then 被确定为和某个模式匹配的最长输入前缀，并且如图 3-23 所示，模式 then 被置于{id}之前，那么返回的词法单元将是 THEN，而不是 ID。 □

3.5.4 向前看运算符

Lex 自动地向前读入一个字符，它会读取到形成被选词素的全部字符之后的那个字符，然后再回退输入，使得只有词素本身从输入中消耗掉。但是在某些时候，我们希望仅当词素的后面跟随特定的其他字符时，这个词素才能和某个特定的模式相匹配。在这种情况下，我们可以在模式中用斜线来指明该模式中和词素实际匹配的部分的结尾，斜线/之后的内容表示一个附加的模式，只有附加模式和输入匹配之后，我们才可以确定已经看到了要寻找的词法单元的词素，但是和第二个模式匹配的字符并不是这个词素的一部分。

例 3.13 在 Fortran 和一些其他语言中，关键字并不是保留字。这种情形会产生一些问题，比如下面的语句

```
IF(I, J) =3
```

其中，IF 是一个数组的名字，而不是关键字。与这条语句形成对比的是下面形式的语句：

```
IF ( condition ) THEN ···
```

在这里，IF 是一个关键字。幸运的是，我们可以确定关键字 IF 后面总是跟着一个左括号，然后是一些可能包含在括号中的文本，即条件表达式，接着是一个右括号和一个字母。那么，我们可以为关键字 IF 写出如下的 Lex 规则：

```
IF /\( . * \) {letter}
```

这条规则是说和这个词素匹配的模式仅仅是两个字母 IF。斜线表示后面会有一个附加的模式，但是这个模式并不和词素匹配。在这个附加模式中，第一个字符是左括号。由于左括号是 Lex 的一个元符号，因此我们必须在它的前面加上一个反斜线，说明它表示的是其字面含义。其中的 . * 与“任何不包含换行符的字符串”匹配。请注意，点号是一个 Lex 的元符号，表示“除换行符外的任何字符”。接下来是一个右括号，同样也加一个反斜线使得该字符表示其字面含义。该附加模式的最后是符号 *letter*，该符号是一个正则定义，表示代表所有字母的字符类。

注意，为了使该模式简单可靠，我们必须对输入进行预处理，消除其中的空白符。在该模式中，我们既没有考虑到空白符，也不能处理条件表达式跨行的情形，因为点号不能和一个换行符匹配。

例如，假设该模式被用来匹配下面的输入前缀：

```
IF(A<(B+C)*D)THEN···
```

前两个字符和 IF 匹配，下一字符和\(匹配，接下来的九个字符和 . * 匹配，再接下来的两个字符分别和\) 及 *letter* 匹配。请注意，第一个右括号(在 C 的后面)后面跟的不是一个字母，这个事实与问题不相关，因为我们只需要找到某种方式将输入与模式相匹配。最后我们得出结论，字符 IF 组成一个词素，并且它们是词法单元 **if** 的一个实例。 □

3.5.5 3.5 节的练习

练习 3.5.1：描述如何对图 3-23 中的 Lex 程序作出如下修改：

1) 增加关键字 while。

2) 将比较运算符转变成 C 语言中的同类运算符。

3) 允许把下划线当作一个附加的字母。

! 4)增加一个新的具有词法单元 STRING 的模式。该模式由一个双引号(")、任意字符串以

及结尾处的一个双引号组成。但是，如果一个双引号出现在上述串中，那么它的前面必须加上一个反斜线(\)进行转义处理，因此在该字符串中的反斜线将用双反斜线表示。这个词法单元的词法值是去掉了双引号的字符串，并且其中用于转义的反斜线已经被删除。识别得到的字符串将被存放到一个字符串表中。

练习 3.5.2：编写一个 Lex 程序。该程序拷贝一个文件，并将文件中每个非空的空白符序列替换为单个空格。

练习 3.5.3：编写一个 Lex 程序。该程序拷贝一个 C 程序，并将程序中关键字 float 的每个实例替换成 double。

! **练习 3.5.4**：编写一个 Lex 程序。该程序把一个文件改变成为“Pig latin”文。明确地讲，假设该文件是一个用空白符分隔开的单词(即字母串)序列。每当你遇到一个单词时：

1) 如果第一个字母是辅音字母，则将它移到单词的结尾，并加上 ay。

2) 如果第一个字母是元音字母，则只在单词的结尾加上 ay。

所有非字母的字符不加处理直接拷贝到输出。

! **练习 3.5.5**：在 SQL 中，关键字和标识符都是大小写不敏感的。编写一个 Lex 程序，该程序识别(大小写字母任意组合的)关键字 SELECT、FROM 和 WHERE 以及词法单元 ID。考虑到这个练习的目的，你可以把 ID 看成是任何以一个字母开头、由字母和数位组成的字符串。你不必将标识符存放到一个符号表中，但需要指出这里的“install”函数与图 3-23 中用于描述大小写敏感标识符的函数有何不同。

3.6 有穷自动机

现在，我们将揭示 Lex 是如何将它的输入程序变成一个词法分析器的。转换的核心是被称为*有穷自动机*(finite automata)的表示方法。这些自动机在本质上是与状态转换图类似的图，但有如下几点不同：

1) 有穷自动机是*识别器*(recognizer)，它们只能对每个可能的输入串简单地回答“是”或“否”。

2) 有穷自动机分为两类：

① *不确定的有穷自动机*(Nondeterministic Finite Automata，NFA)对其边上的标号没有任何限制。一个符号标记离开同一状态的多条边，并且空串 ϵ 也可以作为标号。

② 对于每个状态及自动机输入字母表中的每个符号，*确定的有穷自动机*(Deterministic Finite Automata，DFA)有且只有一条离开该状态、以该符号为标号的边。

确定的和不确定的有穷自动机能识别的语言的集合是相同的。事实上，这些语言的集合正好是能够用正则表达式描述的语言的集合。这个集合中的语言称为*正则语言*(regular language)[⊖]。

3.6.1 不确定的有穷自动机

一个不确定的有穷自动机(NFA)由以下几个部分组成：

1) 一个有穷的状态集合 S。

⊖ 这里有个小问题：按照我们的定义，正则表达不能描述空的语言，因为我们在实践中从不会想到使用这样的模式。但是，有穷自动机可以定义空语言。在理论研究中，$\emptyset$ 被视为一个额外的正则表达式，这个表达式的用途就是定义空语言。

2）一个输入符号集合 Σ，即*输入字母表*(input alphabet)。我们假设代表空串的 ϵ 不是 Σ 中的元素。

3）一个*转换函数*(transition function)，它为每个状态和 Σ∪{ϵ} 中的每个符号都给出了相应的*后继状态*(next state)的集合。

4）S 中的一个状态 s_0 被指定为*开始状态*，或者说*初始状态*。

5）S 的一个子集 F 被指定为*接受状态*(或者说*终止状态*的)集合。

不管是 NFA 还是 DFA，我们都可以将它表示为一张*转换图*(transition graph)。图中的结点是状态，带有标号的边表示自动机的转换函数。从状态 s 到状态 t 存在一条标号为 a 的边当且仅当状态 t 是状态 s 在输入 a 上的后继状态之一。这个图与状态转换图十分相似，但是：

① 同一个符号可以标记从同一状态出发到达多个目标状态的多条边。

② 一条边的标号不仅可以是输入字母表中的符号，也可以是空符号串 ϵ。

例 3.14 图 3-24 给出了一个能够识别正则表达式(**a**|**b**)* **abb** 的语言的 NFA 的转换图。这个抽象的例子描述了所有由 a 和 b 组成的、以字符串 abb 结尾的字符串。这个例子将贯穿本节。虽然它很抽象，但是实际上它与一些具有实际意义的语言的正则表达式相似。例如，描述所有其名字以 .o 结尾的文件的表达式是 **any*** .o，其中 **any** 表示任何可打印字符。

沿用状态转换图中的惯例，状态 3 的双圈表明该状态是接受状态。请注意，从状态 0 到达接受状态的所有路径都是先在状态 0 上运行一段时间，然后从输入中读取 abb，分别进入状态 1、2 和 3。因此能够到达接受状态的所有字符串都是以 abb 结尾的。 □

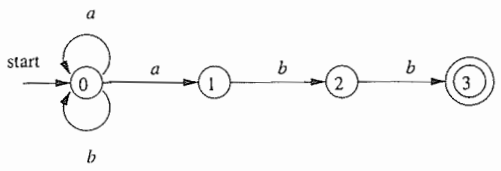


图 3-24 一个不确定有穷自动机

3.6.2 转换表

我们也可以将一个 NFA 表示为一张*转换表*(transition table)，表的各行对应于状态，各列对应于输入符号和 ϵ。对应于一个给定状态和给定输入的条目是将 NFA 的转换函数应用于这些参数后得到的值。如果转换函数没有给出对应于某个状态 - 输入对的信息，我们就把 Ø 放入相应的表项中。

例 3.15 图 3-25 显示了与图 3-24 的 NFA 对应的转换表。 □

转换表的优点是我们能够很容易地确定和一个给定状态和一个输入符号相对应的转换。它的缺点是：如果输入字母表很大，且大多数状态在大多数输入字符上没有转换的时候，转换表需要占用大量空间。 □

3.6.3 自动机中输入字符串的接受

状态	a	b	ϵ
0	{0,1}	{0}	Ø
1	Ø	{2}	Ø
2	Ø	{3}	Ø
3	Ø	Ø	Ø

图 3-25 对应于图 3-24 的 NFA 的转换表

一个 NFA *接受*(accept)输入字符串 x，当且仅当对应的转换图中存在一条从开始状态到某个接受状态的路径，使得该路径中各条边上的标号组成符号串 x。注意，路径中的 ϵ 标号将被忽略，因为空串不会影响到根据路径构建得到的符号串。

例 3.16 图 3-24 的 NFA 接受符号串 $aabb$，因为存在如下从状态 0 到达状态 3 的标号序列为 $aabb$ 的路径：

$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

请注意，可能还存在多条具有相同标号序列、但是到达不同状态的路径。例如下面的路径

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

是另一条从状态 0 出发、标号序列同样为 *aabb* 的路径。这条路径最后仍回到状态 0，但状态 0 不是接受状态。然而，请记住，只要存在某条其标号序列为某符号串的路径能够从开始状态到达某个接受状态，NFA 就接受这个符号串。存在某些到达非接受状态的路径并不会影响这个结论。 □

由一个 NFA 定义(或接受)的语言是从开始状态到某个接受状态的所有路径上的标号串的集合。前面提到过，图 3-24 中的 NFA 定义的语言和正则表达式(**a**|**b**)***abb** 定义的语言相同，即所有来自字母表{*a*, *b*}且以串 *abb* 结尾的串的集合。我们可以用 $L(A)$ 表示自动机 A 接受的语言。

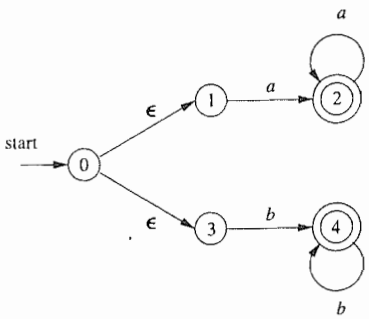


图 3-26 接受 **aa***|**bb*** 的 NFA

例 3.17 图 3-26 是一个接受 L(**aa***|**bb***)的 NFA。因为存在如下的路径：

$$0 \xrightarrow{\epsilon} 1 \xrightarrow{a} 2 \xrightarrow{a} 2 \xrightarrow{a} 2$$

字符串 *aaa* 被这个 NFA 接受。请注意，路径中的 ϵ 标号在连接时“消失”了，因此这条路径的标号是 *aaa*。 □

3.6.4 确定的有穷自动机

确定的有穷自动机(简称 DFA)是不确定有穷自动机的一个特例，其中：

1) 没有输入 ϵ 之上的转换动作。

2) 对每个状态 s 和每个输入符号 a，有且只有一条标号为 a 的边离开 s。

如果我们使用转换表来表示一个 DFA，那么表中的每个表项就是一个状态。因此，我们可以不使用花括号，直接写出这个状态，因为花括号只是用来说明表项的内容是一个集合。

NFA 抽象地表示了用来识别某个语言中的串的算法，而相应的 DFA 则是一个简单具体的识别串的算法。在构造词法分析器的时候，我们真正实现或模拟的是 DFA。幸运的是，每个正则表达式和每个 DFA 都可以被转变成为一个接受相同语言的 DFA。下边的算法说明了如何将 DFA 用于串的识别。

算法 3.18 模拟一个 DFA。

输入： 一个以文件结束符 **eof** 结尾的字符串 x。DFA D 的开始状态为 s_0，接受状态集为 F，转换函数为 *move*。

输出： 如果 D 接受 x，则回答“yes”，否则回答“no”。

方法： 把图 3-27 中的算法应用于输入字符串 x。函数 *move*(s, c)给出了从状态 s 出发，标号为 c 的边所到达的状态。函数 *nextchar* 返回输入串 x 的下一个字符。 □

```
s = s0;
c = nextChar();
while ( c != eof ) {
        s = move(s,c);
        c = nextChar();
}
if ( s在F中 ) return "yes";
else return "no";
```

图 3-27 模拟一个 DFA

例 3.19 图 3-28 显示的是一个 DFA 的转换图。该 DFA 接受的语言与图 3-24 的 NFA 所接受的语言相同，都是(**a**|**b**)***abb**。给定输入串 *ababb*，这个 DFA 顺序进入状态序列 0、1、2、1、2、3，并返回“yes”。 □

3.6.5 3.6 节的练习

!练习 3.6.1：3.4 节的练习中的图 3-19 计算了 KMP 算法的失效函数。说明在已知失效函数的情况下，如何根据已知的关键字 $b_1b_2\cdots b_n$，构造出一个具有 $n+1$ 个状态的 DFA，该 DFA 可以识别语言 $.^*b_1b_2\cdots b_n$（其中，点代表任意字符）。更进一步，证明构造这个 DFA 的时间复杂度是 $O(n)$。

练习 3.6.2：为练习 3.3.5 中的每一个语言设计一个 DFA 或 NFA。

练习 3.6.3：找出图 3-29 所示的 NFA 中所有标号为 *aabb* 的路径。这个 NFA 接受 *aabb* 吗？

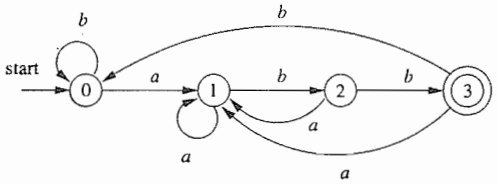


图 3-28 接受(a|b)*abb 的 DFA

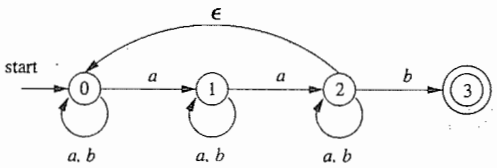


图 3-29 练习 3.6.3 的 NFA

练习 3.6.4：对于图 3-30 的 NFA，重复练习 3.6.3。

练习 3.6.5：给出如下练习中的 NFA 的转换表：

1）练习 3.6.3。

2）练习 3.6.4。

3）图 3-26。

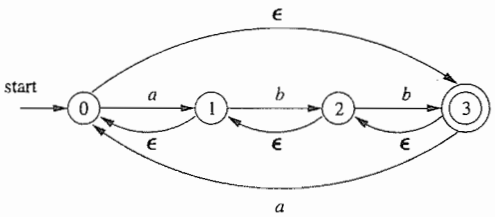


图 3-30 练习 3.6.4 的 NFA

3.7 从正则表达式到自动机

就像 3.5 节的内容所介绍的，正则表达式非常适合描述词法分析器和其他模式处理软件。然而那些软件的实现需要像算法 3-18 中那样来模拟 DFA 的执行，或者模拟 NFA 的执行。由于 NFA 对于一个输入符号可以选择不同的转换（如在图 3-24 中的状态 0 上输入为 a 时），它还可以执行输入 ϵ 上的转换（如在图 3-26 中的状态 0 上时），甚至可以选择是对 ϵ 或是对真实的输入符号执行转换，因此对 NFA 的模拟不如对 DFA 的模拟直接。于是，我们需要将一个 NFA 转换为一个识别相同语言的 DFA。

这一节我们将首先介绍如何把 NFA 转化为 DFA。然后，我们利用这种称为“子集构造法”的技术给出一个直接模拟 NFA 的算法。这个算法可用于那些将 NFA 转化到 DFA 比直接模拟 NFS 更加耗时的（非词法分析的）情形。接着，我们将说明如何把正则表达式转换为 NFA，在必要时可以根据这个 NFA 构造出一个 DFA。最后我们讨论了不同的正则表达式实现技术之间的时间－空间权衡问题，并说明如何为具体的应用选择合适的方法。

3.7.1 从 NFA 到 DFA 的转换

子集构造法的基本思想是让构造得到的 DFA 的每个状态对应于 NFA 的一个状态集合。DFA 在读入输入 $a_1a_2\cdots a_n$ 之后到达的状态对应于相应 NFA 从开始状态出发，沿着以 $a_1a_2\cdots a_n$ 为标号的路径能够到达的状态的集合。

DFA 的状态数有可能是 NFA 状态数的指数，在这种情况下，我们在试图实现这个 DFA 时会遇到困难。然而，基于自动机的词法分析方法的处理能力部分源于如下事实：对于一个真实的语言，它的 NFA 和 DFA 的状态数量大致相同，状态数量呈指数关系的情形尚未在实践中

出现过。

算法 3.20 由 NFA 构造 DFA 的子集构造(subset construction)算法。

输入：一个 NFA N。

输出：一个接受同样语言的 DFA D。

方法：我们的算法为 D 构造一个转换表 *Dtran*。D 的每个状态是一个 NFA 状态集合，我们将构造 *Dtran*，使得 D"并行地"模拟 N 在遇到一个给定输入串时可能执行的所有动作。我们面对的第一个问题是正确处理 N 的 ϵ 转换。在图 3-31 中我们可以看到一些函数的定义。这些函数描述了一些需要在这个算法中执行的 N 的状态集上的基本操作。请注意，s 表示 N 的单个状态，而 T 代表 N 的一个状态集。

操作	描述
ϵ-*closure*(s)	能够从NFA的状态 s 开始只通过 ϵ 转换到达的 NFA状态集合
ϵ-*closure*(T)	能够从T中某个NFA状态 s 开始只通过 ϵ 转换到达的NFA状态集合，即 $\cup_{s\in T}\ \epsilon$-*closure*(s)
move(T, a)	能够从 T 中某个状态 s 出发通过标号为 a 的转换到达的NFA状态的集合

图 3-31 NFA 状态集上的操作

我们必须找出当 N 读入了某个输入串之后可能位于的所有状态集合。首先，在读入第一个输入符号之前，N 可以位于集合 ϵ-*closure*(s_0) 中的任何状态上，其中 s_0 是 N 的开始状态。下面进行归纳。假定 N 在读入输入串 x 之后可以位于集合 T 中的状态上。如果下一个输入符号是 a，那么 N 可以立即移动到集合 *move*(T, a) 中的任何状态。然而，N 可以在读入 a 后再执行几个 ϵ 转换，因此 N 在读入 xa 之后可位于 ϵ-*closure*(*move*(T, a)) 中的任何状态上。根据这些思想，我们可以得到图 3-32 中显示的方法，该方法构造了 D 的状态集合 *Dstates* 和 D 的转换函数 *Dtran*。

```
一开始, ε-closure(s0)是 Dstates 中的唯一状态, 且它未加标记;
while (在 Dstates 中有一个未标记状态 T) {
    给T加上标记;
    for (每个输入符号 a) {
        U = ε-closure(move(T, a));
        if (U 不在Dstates中 )
            将U 加入到Dstates中, 且不加标记;
        Dtran[T, a] = U;
    }
}
```

图 3-32 子集构造法

D 的开始状态是 ϵ-*closure*(s_0)，D 的接受状态是所有至少包含了 N 的一个接受状态的状态集合。我们只需要说明如何对 NFA 的任何状态集合 T 计算 ϵ-*closure*(T)，就可以完整地描述子集构造法。这个计算过程显示在图 3-33 中。它是从一个状态集合开始的一次简单的图搜索过程，不过此时假设这个图中只存在标号为 ϵ 的边。 □

例 3.21 图 3-34 给出了另一个接受语言(**a**|**b**)***abb** 的 NFA。它正好是我们将在 3.7 节中根据这个正则表达式直接构造得到的 NFA。我们现在把算法 3.20 应用到图 3-34 中。

```
将T的所有状态压入stack中；
将 ε-closure(T) 初始化为T;
while ( stack非空) {
        将栈顶元素t弹出栈中;
        for (每个满足如下条件的u：从t出发有一个标号为ε的转换到达状态u)
                if ( u不在 ε-closure(T)中) {
                        将u加入到ε-closure(T)中;
                        将u压入栈中;
                }
}
```

图 3-33 计算 ϵ-*closure*(*T*)

等价 NFA 的开始状态 A 是 ϵ-*closure*(0)，即 $A = \{0, 1, 2, 4, 7\}$。A 中的状态就是能够从状态 0 出发，只经过标号为 ϵ 的路径到达的所有状态。请注意，因为路径可以不包含边，所以状态 0 也是可以从它自身出发经过标号为 ϵ 的路径到达的状态。

NFA 的输入字母表是 $\{a, b\}$。因此，我们的第一步是标记 A，并计算 $Dtran[A, a] = \epsilon$-*closure*(*move*(A, a)) 以及 $Dtran[A, b] = \epsilon$-*closure*(*move*(A, b))。在状态 0、1、2、4、7 中，只有 2 和 7 有 a 上的转换，分别到达状态 3 和 8，因此 *move*(A, a) = {3, 8}，同时 ϵ-*closure*({3, 8}) = {1, 2, 3, 4, 6, 7, 8}。因此我们有：

$$Dtran[A,a] = \epsilon\text{-}closure(move(A,a)) = \epsilon\text{-}closure(\{3,8\}) = \{1,2,3,4,6,7,8\}$$

我们称这个集合为 B，得到 $Dtran[A, a] = B$。

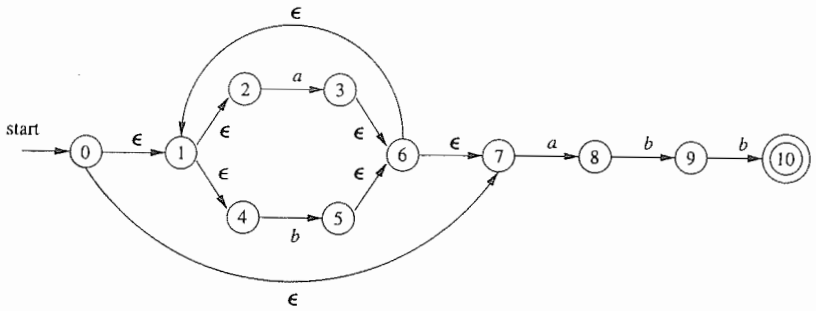


图 3-34 **(a|b)*abb** 对应的 NFA N

现在我们要计算 $Dtran[A, b]$。在 A 的状态中只有 4 有一个输入 b 上的转换，它到达状态 5，因此

$$Dtran[A,b] = \epsilon\text{-}closure(\{5\}) = \{1,2,4,6,7\}$$

我们称这个集合为 C，因此 $Dtran[A, b] = C$。

如果我们对未加标记的集合 B 和 C 继续这个处理过程，最终会使得这个 DFA 的所有状态都被加上标记。这个结论一定正确，因为 11 个 NFA 状态的集合只有 2^{11} 个子集。我们实际上构造出 5 个不同的 DFA 状态。这些状态、它们对应的 NFA 状态集以及 D 的转换表显示在图 3-35 中。D 的转换图如图 3-36 所示。状态 A 是 D 的开始状态，而包含 NFA 状态 10 的 E 状态是唯一的接受状态。

NFA 状态	DFA 状态	a	b
{0, 1, 2, 4, 7}	A	B	C
{1, 2, 3, 4, 6, 7, 8}	B	B	D
{1, 2, 4, 5, 6, 7}	C	B	C
{1, 2, 4, 5, 6, 7, 9}	D	B	E
{1, 2, 4, 5, 6, 7, 10}	E	B	C

图 3-35 DFA D 的转换表 *Dtran*

请注意，相比图 3-28 中接受相同语言(**a**|**b**)* **abb** 的 DFA，这个 DFA D 多了一个状态。D 的状态 A 和 C 具有同样的转换函数，因此可以被合并。我们将在 3.9.6 中讨论使一个 DFA 的状态个数最小化问题。 □

3.7.2 NFA 的模拟

许多文本编辑程序使用的策略是根据一个正则表达式构造出相应的 NFA，然后使用类似于 on-the-fly(即边构造边使用的)的子集构造法来模拟这个 NFA 的执行。这种模拟执行方法将在下面给出。

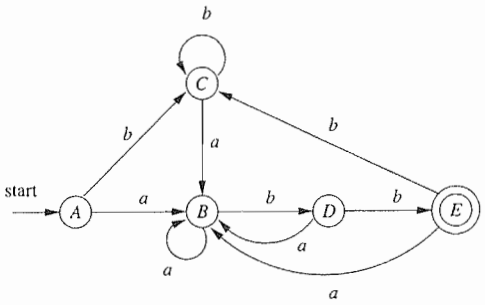


图 3-36 将子集构造法应用于图 3-34 的结果

算法 3.22 模拟一个 NFA 的执行。

输入：一个以文件结束符 **eof** 结尾的输入串 x。一个 NFA N，其开始状态为 s_0，接受状态集为 F，转换函数为 *move*。

输出：如果 N 接受 x 则返回“yes”，否则返回“no”。

方法：这个算法保存了一个当前状态的集合 S，即那些可以从 s_0 开始沿着标号为当前已读入输入部分的路径到达的状态的集合。如果 c 是函数 *nextChar*( )读到的下一个输入字符，那么我们首先计算 *move*(S, c)，然后使用 ϵ-*closure* 求出这个集合的闭包。该算法的思想如图 3-37 所示。 □

3.7.3 NFA 模拟的效率

```
1)  S = ε-closure(s0);
2)  c = nextChar();
3)  while ( c != eof ) {
4)          S = ε-closure(move(S, c));
5)          c = nextChar();
6)  }
7)  if ( S ∩ F != ∅ ) return "yes";
8)  else return "no";
```

图 3-37 模拟一个 NFA

如果精心实现，算法 3.22 可以相当高效。因为这些高效实现的思想可以用于许多涉及图搜索的算法。我们将更详细地介绍这个实现。我们需要的数据结构包括：

1）两个堆栈，其中每一个堆栈都存放了一个 NFA 状态集合。其中的一个堆栈 *oldStates* 存放“当前状态集合”，即图 3-37 的第 4 行中右边的 S 的值。另一个堆栈 *newStates* 存放了“下一个”状态集合，即第 4 行中左边的 S 的值。在我们运行第 3 行到第 6 行的循环时，中间的一个步骤没有在图 3-37 中列出，即把 *newStates* 的值转移到 *oldStates* 中去的步骤。

2）一个以 NFA 状态为下标的布尔数组 *alreadyOn*。它指示出哪个状态已经在 *newStates* 中。虽然这个数组存放的信息和栈中存放的信息相同，但查询 *alreadyOn*[s]要比在栈 *newStates* 中查询 s 快很多。我们同时保持两种表示方法的原因就是为了获得这个效率。

3）一个二维数组 *move*[s, a]，它保存这个 NFA 的转换表。这个表中的条目是状态的集合，它们用链表表示。

为了实现图 3-37 的第一行，我们需要将 *alreadyOn* 数组中的所有条目都设置为 FALSE，然后对于 ϵ-*closure*(s_0)中的每个状态 s，将 s 压入 *oldStates* 并设置 *alreadyOn*[s]为 TRUE。这个对状态 s 的操作以及图 3-37 第 4 行中的操作，都可以使用函数 *addState*(s)来实现。这个函数将 s 压入 *newStates*，将 *alreadyOn*[s]设置为 TRUE，并使用 *move*[s, ϵ]作为参数递归地调用自身，继续计算 ϵ-*closure*(s)的值。然而，为了避免重复工作，我们必须小

```
9)  addState(s) {
10)         将s压入栈newStates中;
11)         alreadyOn[s] = TRUE;
12)         for ( t on move[s, ε] )
13)                 if ( !alreadyOn[t] )
14)                         addState(t);
15) }
```

图 3-38 加入一个不在 *newStates* 中的新状态 s

心，不要对一个已经在栈 *newStates* 中的状态调用 *addState*。图 3-38 给出了这个函数的概要。

我们通过查看 *oldStates* 中的每个状态 s 来实现图 3-37 的第 4 行。我们首先找出状态集合 *move*[s, c]，其中 c 是下一个输入字符。对于那些不在 *newStates* 栈中的状态，我们应用函数 *addState*。注意，*addState* 还计算了一个状态的 ϵ-*closure* 值，并把其中的状态一起加入到 *newStates* 中(如果这些状态不存在的话)。这一系列处理步骤如图 3-39 所示。

假定一个 NFA N 有 n 个状态和 m 个转换，即 m 是离开各个状态的转换数的总和。如果不包括第 19 行中对 *addState* 的调用，在第 16 行到第 21 行的循环上花费的时间是 $O(n)$。也就是说，我们最多需要运行这个循环 n 遍，且如果不考虑调用 *addState* 所花费的时间，每一遍的工作量都是常数。对于第 22 行到第 26 行的循环，这个结论也成立。

```
16)  for (oldStates上的每个s) {
17)      for (move[s, c]中的每个t)
18)          if ( !alreadyOn[t] )
19)              addState(t);
20)      将s弹出oldStates栈 ;
21)  }

22)  for (newStates中的每个s) {
23)      将s弹出newStates栈 ;
24)      将s压入oldStates 栈 ;
25)      alreadyOn[s] = FALSE;
26)  }
```

图 3-39 图 3-37 中第 4 步的实现

在图 3-39 的一次执行中(即图 3-37 的第 4 行)，对于任意给定的状态最多只能调用 *addState* 一次。原因在于每次调用 *addState*(s) 时都会在图 3-38 的第 11 行上把 *alreadyOn*[s] 置为 TRUE。一旦 *alreadyOn*[s] 设为 TRUE，图 3-38 的第 13 行和图 3-39 的第 18 行就会禁止再次调用 *addState*(s)。

如果不考虑第 14 行中的递归调用所花费的时间，第 10 行、第 11 行对 *addState* 的一次调用所花的时间为 $O(1)$，第 12、13 行的时间取决于有多少 ϵ 转换离开 s。对于一个给定的状态，我们不知道这个数目是多少，但是我们知道最多只有 m 个离开各个状态的转换。因此，在图 3-39 中代码的一次执行中，在第 12 行和 13 行上用于调用 *addState* 的累计时间为 $O(m)$。花费在 *addState* 的其他步骤的累计时间为 $O(n)$，因为每一次调用的时间是一个常数，且最多只有 n 次调用。

因此我们可以得出如下结论，即只要实现方法得当，执行图 3-37 的第 4 行的时间是 $O(n+m)$。从第 3 行到第 6 行的 while 循环的其余部分在每次迭代时花费 $O(1)$ 时间。如果输入 x 的长度为 k，那么该循环的总工作量为 $O(k(n+m))$。图 3-37 的第 1 行的执行时间为 $O(n+m)$，因为它实际上就是图 3-39 中的各个步骤，只不过 *oldStates* 中只包含状态 s_0。第 2、7、8 行都花费 $O(1)$ 时间。因此，如果实现正确，算法 3.22 的运行时间为 $O(k(n+m))$。也就是说，该算法所需时间和输入串的长度和转换图的大小(结点数加上边数)的乘积成正比。

大 O 表示法

形如 $O(n)$ 的表达式是“最多某个常数乘以 n”的缩写。从技术上讲，我们说一个函数 $f(n)$ 是 $O(g(n))$ 的条件是存在常量 c 和 n_0 使得当 $n \geqslant n_0$ 时必然有 $f(n) \leqslant cg(n)$。这里的 $f(n)$ 可能是一个算法的某些步骤的运行时间。一个有用的写法是“$O(1)$”，它表示“某个常量”。使用大 O 表示法可以使得我们不需要过多地考虑使用什么样的运行时间单位来进行度量，而仍然可以表示一个算法的运行时间的增长速度。

3.7.4 从正则表达式构造 NFA

现在我们给出一个算法，它可以将任何正则表达式转变为接受相同语言的 NFA。这个算法是语法制导的，也就是说它沿着正则表达式的语法分析树自底向上递归地进行处理。对于每个子表达式，该算法构造一个只有一个接受状态的 NFA。

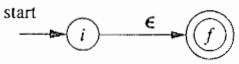
算法 3.23 将正则表达式转换为一个 NFA 的 McMaughton-Yamada-Thompson 算法。

输入：字母表 Σ 上的一个正则表达式 r。

输出：一个接受 $L(r)$ 的 NFA N。

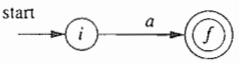
方法：首先对 r 进行语法分析，分解出组成它的子表达式。构造一个 NFA 的规则分为基本规则和归纳规则两组。基本规则处理不包含运算符的子表达式，而归纳规则根据一个给定表达式的直接子表达式的 NFA 构造出这个表达式的 NFA。

基本规则：对于表达式 ϵ，构造下面的 NFA。



这里，i 是一个新状态，也是这个 NFA 的开始状态；f 是另一个新状态，也是这个 NFA 的接受状态。

对于字母表 Σ 中的子表达式 a，构造下面的 NFA。



同样，i 和 f 都是新状态，分别是这个 NFA 的开始状态和接受状态。请注意，在这两个基本构造规则中，对于 ϵ 或某个 a 的作为 r 的子表达式的每次出现，我们都会使用新状态分别构造出一个独立的 NFA。

归纳规则：假设正则表达式 s 和 t 的 NFA 分别为 $N(s)$ 和 $N(t)$。

1）假设 $r=s|t$，r 的 NFA，即 $N(r)$，可以按照图 3-40 中的方式构造得到。这里 i 和 f 是新状态，分别是 $N(r)$ 的开始状态和接受状态。从 i 到 $N(s)$ 和 $N(t)$ 的开始状态各有一个 ϵ 转换，从 $N(s)$ 和 $N(t)$ 到接受状态 f 也各有一个 ϵ 转换。请注意，$N(s)$ 和 $N(t)$ 的接受状态在 $N(r)$ 中不是接受状态。因为从 i 到 f 的任何路径要么只通过 $N(s)$，要么只通过 $N(t)$，且离开 i 或进入 f 的 ϵ 转换都不会改变路径上的标号，因此我们可以判定 $N(r)$ 识别 $L(s)\cup L(t)$，也就是 $L(r)$。也就是说，图 3-40 中的 NFA 是一个正确的处理并运算符的构造。

2）假设 $r=st$，然后按照图 3-41 所示构造 $N(r)$。$N(s)$ 的开始状态变成了 $N(r)$ 的开始状态。$N(t)$ 的接受状态成为 $N(r)$ 的唯一接受状态。$N(s)$ 的接受状态和 $N(t)$ 的开始状态合并为一个状态，合并后的状态拥有原来进入和离开合并前的两个状态的全部转换。图 3-41 中一条从 i 到 f 的路径必须首先经过 $N(s)$，因此这条路径的标号以 $L(s)$ 中的某个串开始。然后，这条路径继续通过 $N(t)$，因此这条路径的标号以 $L(t)$ 中的某个串结束。就像我们很快要论证的，没有转换离开构造得到的接受状态，也没有转换进入开始状态，因此一个路径不可能在离开 $N(s)$ 后再次进入 $N(s)$。因此，$N(r)$ 恰好接受 $L(s)L(t)$，它是 $r=st$ 的一个正确的 NFA。

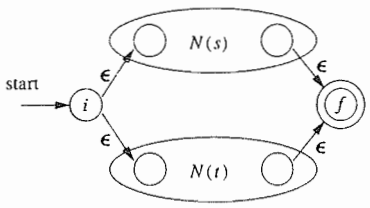


图 3-40　两个正则表达式的并的 NFA

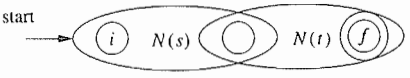


图 3-41　两个正则表达式的连接的 NFA

3）假设 $r=s^*$，然后为 r 构造出图 3-42 所示的 *NFA* $N(r)$。这里，i 和 f 是两个新状态，分别是 $N(r)$ 的开始状态和唯一的接受状态。要从 i 到达 f，我们可以沿着新引入的标号为 ϵ 的路径前进，这个路径对应于 $L(s)^0$ 中的一个串。我们也可以到达 $N(s)$ 的开始状态，然后经过该 NFA，再零次或多次从它的接受状态回到它的开始状态并重复上述过程。这些选项使得 $N(r)$ 可以接受

$L(s)^1$、$L(s)^2$ 等集合中的所有串，因此 $N(r)$ 识别的所有串的集合就是 $L(s)^*$。

4）最后，假设 $r=(s)$，那么 $L(r)=L(s)$，我们可以直接把 $N(s)$ 当作 $N(r)$。 □

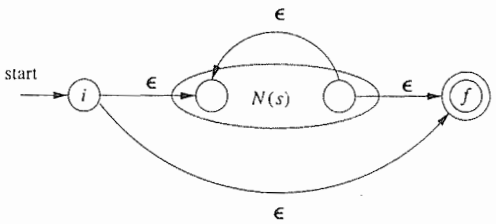


图 3-42 一个正则表达式的闭包的 NFA

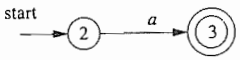
算法 3.23 中描述的方法包含了一些提示，说明为什么这个归纳性构造方法能够得到正确的解答。我们不会给出正式的正确性证明。但除了最重要的性质，即 $N(r)$ 接受语言 $L(r)$ 之外，我们还在下面列出一些由该算法构造得到的 NFA 所具有的性质。这些性质本身也很有趣，并且有助于正式证明这个方法的正确性。

1）$N(r)$ 的状态数最多为 r 中出现的运算符和运算分量的总数的 2 倍。得出这个上界的原因是算法的每一个构造步骤最多只引入两个新状态。

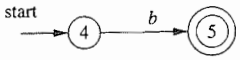
2）$N(r)$ 有且只有一个开始状态和一个接受状态。接受状态没有出边，开始状态没有入边。

3）$N(r)$ 中除接受状态之外的每个状态要么有一条其标号为 Σ 中符号的出边，要么有两条标号为 ϵ 的出边。

例 3.24 让我们用算法 3.23 为正则表达式 $r=(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$ 构造一个 NFA。图 3-43 显示了 r 的一棵语法分析树，这棵树和 2.2.3 节中构造的算术表达式的语法分析树相似。对于子表达式 r_1，即第一个 **a**，我们构造如下的 NFA：



我们在选择这个 NFA 中的状态编号时考虑了和接下来生成的 NFA 的状态编号之间的一致性。对 r_2 构造如下 NFA：



现在我们可以使用图 3-40 中的构造方法，将 $N(r_1)$ 和 $N(r_2)$ 合并，得到 $r_3=r_1|r_2$ 的 NFA。这个 NFA 显示在图3-44中。

子表达式 $r_4=(r_3)$ 的 NFA 和 r_3 的 NFA 相同。子表达式 $r_5=(r_3)^*$ 的 NFA 的构造如图 3-45 所示。我们使用图 3-42 所示的方法根据图 3-44 中的 NFA 构造出这个 NFA。

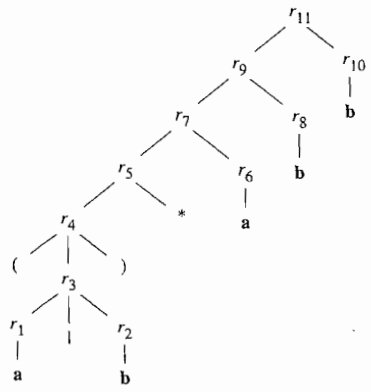


图 3-43 $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$ 的语法分析树

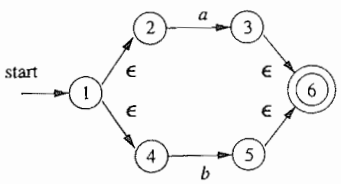
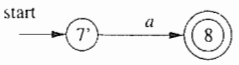


图 3-44 r_3 的 NFA

现在考虑 r_6，它是表达式中的另一个 **a**。我们再次对 a 使用基本构造法，但是必须使用新的状态。虽然 r_1 和 r_6 是相同的表达式，但这个构造方法不允许我们复用那个为 r_1 构造的 NFA。r_6 的 NFA 如下：



要得到 $r_7 = r_5 r_6$ 的 NFA，我们应用图 3-41 中的构造方法，将状态 7 和 7' 合并，得到如图3-46所示的 NFA。按照这个方法继续构造出两个分别名为 r_8 和 r_{10}、对应于子表达式 b 的新 NFA，最后构造出如图 3-34 所示的(**a**|**b**)* **abb** 的 NFA。 □

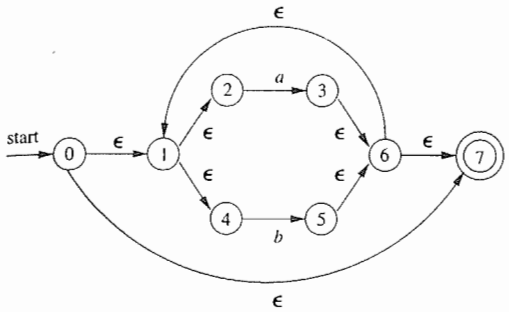


图 3-45 r_5 的 NFA

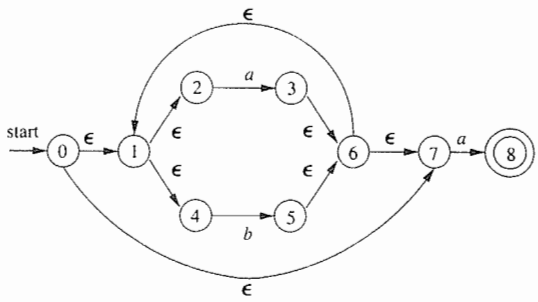


图 3-46 r_T 的 NFA

3.7.5 字符串处理算法的效率

我们看到，算法 3.18 能在 $O(|x|)$ 时间内处理字符串 x，而在 3.7.3 节中我们提到，要模拟一个 NFA 的运行所需的时间与 $|x|$ 和该 NFA 的转换图的大小的乘积成正比。很明显，用 DFA 来模拟比用 NFA 模拟更快，因此我们可能会怀疑模拟一个 NFA 到底有没有意义。

支持使用 NFA 模拟的论据之一是子集构造法在最坏的情况下可能会使状态个数呈指数增长。虽然原则上 DFA 的状态数不会影响算法 3.18 的运行时间，但是假如状态数大到一定程度，以至于转换表超过了主存容量时，那么真正的运行时间就必须加上磁盘读写时间，从而使运行时间显著增加。

例 3.25 考虑形如 $L_n = (\mathbf{a}|\mathbf{b})^* \mathbf{a}(\mathbf{a}|\mathbf{b})^{n-1}$ 的正则表达式所描述的语言族。也就是说，每个语言 L_n 包含了所有由 a 和 b 组成且从右端向左数第 n 个符号是 a 的串。很容易构造出一个具有 $n+1$ 个状态的 NFA。它在任何输入符号上都可以停留在其初始状态，但是当输入为 a 时也可以到达状态 1。在处于状态 1 时，它在任何输入符号上都会转到状态 2，以此类推，当到达状态 n 时它接受输入串。图 3-47 给出了这个 NFA。

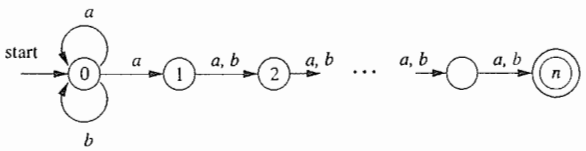


图 3-47 一个 NFA，它的状态数量远小于等价的最小 DFA 的状态数

然而，L_n 的任何一个 DFA 都至少有 2^n 个状态。我们不证明这个结论，只说明其基本思想。假设两个长度均为 n 的串到达 DFA 的同一个状态，必然存在一些位置使得两个串在这些位置上的符号不同(必然一个是 a 而另一个是 b)。我们考虑最后一个这样的位置。我们可以不断把相同的符号同时添加到这两个串的后面，直到它们的最后 $n-1$ 个位置上的符号串相同，但是倒数

第 n 个位置上的符号不同。那么这个 DFA 在处理这两个(经过扩展的)符号串时会到达同一个状态(因为根据假设，此 DFA 在处理未经扩展的两个串时到达同一个状态，而对这两个串的扩展方法相同——译者注)。此时这个 DFA 要么同时接受这两个符号串，要么都不接受这两个符号串。(注意这两个符号串的倒数第 n 个符号是不同的，它们应该有且只有一个串在这个语言中，由此得出矛盾。这说明任意两个长度为 n 的不同符号串应该到达不同的状态。而长度为 n 的符号串共有 2^n 个，也就是说至少要有 2^n 个状态——译者注。)幸运的是，如我们前面提到的，词法分析很少需要处理这种类型的模式，我们也不用担心会遇到状态数量出奇多的 DFA。 □

然而，词法分析器生成工具和其他字符串处理系统经常以正则表达式作为输入。我们面临着将正则表达式转换成 DFA 还是 NFA 的问题。转换成 DFA 的额外开销是在将算法 3.20 应用于转换得到的 NFA 而产生的开销(也可以将一个正则表达式直接转化为 DFA，但工作量实质上是一样的)。如果字符串处理器被频繁使用，比如词法分析器，那么转换到 DFA 时付出的任何代价都是值得的。然而在另一些字符串处理应用中，例如 grep，用户指定一个正则表达式，并在一个或多个文件中搜索这个表达式所描述的模式，那么跳过构造的 DFA 步骤直接模拟 NFA 可能更加高效。

现在我们考虑用算法 3.23 把正则表达式 r 转换成相应的 NFA 的代价。其关键步骤是构造 r 的语法分析树。在第 4 章中我们会看到几种可以在线性时间内构造语法分析树的方法，即在 $O(|r|)$ 时间内完成语法分析树的构造，其中 $|r|$ 表示 r 的大小，也就是 r 中运算符和运算分量的总和。我们也很容易发现每次应用算法 3.23 中的基本规则和归纳规则只需要常数时间，因此转换得到一个 NFA 所花费的全部时间是 $O(|r|)$。

此外，如我们在 3.7.4 节中观察到的，构造得到的 NFA 最多有 $2|r|$ 个状态和 $4|r|$ 个转换。也就是说，根据 3.7.3 节中的分析，可以得到 $n \leqslant 2|r|$ 和 $m \leqslant 4|r|$。因此，模拟这个 NFA 处理输入字符串 x 的过程所花费时间是 $O(|r| \times |x|)$。这个时间远远超过构造 NFA 所用的时间 $O(|r|)$。因此，我们得到，对于正则表达式 r 和字符串 x，能够在 $O(|r| \times |x|)$ 时间内判断 x 是否属于 $L(r)$。

子集构造法所花费的时间很大程度上取决于构造得到的 DFA 的状态数。首先注意在图 3-22 所示的子集结构法中，算法的关键步骤，即根据状态集 T 和输入符号 a 构建状态集 U 的过程与算法 3.22 的 NFA 模拟方法中根据旧状态集构造新状态集的过程类似。我们已经知道，如果实现得当，这个步骤所花的时间最多和 NFA 状态数与转换数之和成正比。

假设我们要从一个正则表达式 r 开始，并将它构造成一个 NFA。这个 NFA 最多有 $2|r|$ 个状态和 $4|r|$ 个转换，并且最多有 $2|r|$ 个输入符号。因此，对于每个构造得到的 DFA 状态，我们最多必须构造 $|r|$ 个新状态，构造每个新状态最多花费 $O(2|r|+4|r|)$ 时间。因此，构造一个有 s 个状态的 DFA 所用的时间为 $O(|r|^2 s)$。

在通常情况下，s 大约等于 $|r|$，上面的子集构造法需要的时间为 $O(|r|^3)$。然而，在如例 3.25 所示的最坏情况下，这个时间是 $O(|r|^2 2^{|r|})$。当我们需要构造一个识别器来指明一个或多个串 x 是否在一个给定的正则表达式 r 所定义的 $L(r)$ 中时，我们有多个选项。图 3-48 对这些选项作了总结。

自动机	初始开销	每个串的开销
NFA	$O(\|r\|)$	$O(\|r\| \times \|x\|)$
DFA typical case	$O(\|r\|^3)$	$O(\|x\|)$
DFA worst case	$O(\|r\|^2 2^{\|r\|})$	$O(\|x\|)$

图 3-48 识别一个正则表达式所表示的语言的不同方法所具有的初始开销和单个串的开销

如果处理各个字符串所花的时间多很多，比如我们构造词法分析器时面临的情况，我们显然倾向于使用 DFA。然而，在像 grep 这样的命令中，我们只会对一个符号串运行这个自动机。此时我们通常倾向于使用 NFA 方式。只有当 $|x|$ 接近 $|r|^3$ 的时候，我们才会考虑转换到 DFA。

还有一种混合策略可以做到对每个正则表达式 r 和输入串 x，它的效率总是和 DFA 和 NFA

方法中较好的一个差不多。这个策略从模拟 NFA 开始，但是在计算出各个状态集（也就是 DFA 的状态）和转换的同时把它们记录下来。在模拟中每次处理此 NFA 的当前状态集合和当前输入符号之前，首先查看我们是否已经计算了这个转换。如果是，就直接使用这个信息。

3.7.6 3.7 节的练习

练习 3.7.1：将下列图中的 NFA 转换为 DFA。

1）图 3-26

2）图 3-29

3）图 3-30

练习 3.7.2：用算法 3.22 模拟下列图中的 NFA 在处理输入 *aabb* 时的过程。

1）图 3-29

2）图 3-30

练习 3.7.3：使用算法 3.23 和 3.20 将下列正则表达式转换成 DFA。

1）$(\mathbf{a}|\mathbf{b})^*$

2）$(\mathbf{a}^*|\mathbf{b}^*)^*$

3）$((\epsilon|\mathbf{a})\mathbf{b}^*)^*$

4）$(\mathbf{a}|\mathbf{b})^*\mathbf{abb}(\mathbf{a}|\mathbf{b})^*$

3.8 词法分析器生成工具的设计

本节中我们将应用 3.7 节中介绍的技术，讨论像 Lex 这样的词法分析器生成工具的体系结构。我们将讨论两种分别基于 NFA 和 DFA 的方法，后者实质上就是 Lex 的实现方法。

3.8.1 生成的词法分析器的结构

图 3-49 概括了由 Lex 生成的词法分析器的体系结构。作为词法分析器的程序包含一个固定的模拟自动机的程序。现在我们暂时不规定这个自动机是确定的还是不确定的。词法分析器的其他部分是由 Lex 根据 Lex 程序创建的组件组成的。

这些组件包括：

1）表示自动机的一个转换表。

2）由 Lex 编译器从 Lex 程序中直接拷贝到输出文件的函数（见 3.5.2 节的讨论）。

3）输入程序定义的动作。这些动作是一些代码片段，将在适当的时候由自动机模拟器调用。

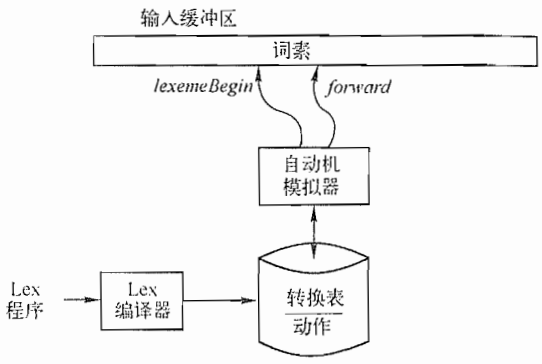


图 3-49 一个 Lex 程序被转变成由有限自动机模拟器使用的转换表和动作

在构建自动机时，我们首先用算法 3.23 把 Lex 程序中的每个正则表达式模式转换为一个 NFA。我们需要使用一个自动机来识别所有与 Lex 程序中的模式相匹配的词素，因此我们将这些 NFA 合并为一个 NFA。合并的方法是引入一个新的开始状态，从这个新开始状态到各个对应于模式 p_i 的 NFA Ni 的开始状态各有一个 ϵ 转换。构造方法如图 3-50 所示。

例 3.26 我们将使用如下所述的简单、抽象的例子来说明本节所要说明的思想：

a {模式 p_1 的动作 A_1}
abb {模式 p_2 的动作 A_2}
a * **b** + {模式 p_3 的动作 A_3}

请注意，上述三个模式之间存在我们在 3.5.3 节中讨论过的冲突。更明确地说，字符串 *abb* 同时满足第二个和第三个模式，但是我们将把它看作模式 p_2 的词素，因为在上面的 Lex 程序中首先列出的是模式 p_2。像 *aabbb*…这样的输入串有很多前缀都满足第三个模式，Lex 的规则是接受最长的前缀，因此我们不断读入 *b*，直到另一个 *a* 出现为止。此时我们报告识别的词素就是从第一个 *a* 开始的、包含了其后所有 *b* 的符号串。

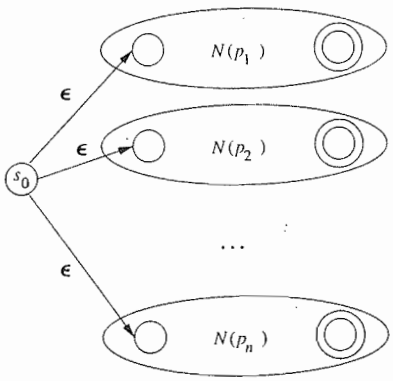


图 3-50 根据 Lex 程序构造得到的一个 NFA

图 3-51 列出了分别识别这三个模式的 NFA。其中第三个 NFA 是根据算法 3.23 的转换结果经简化得到的。然后，图 3-52 显示了通过加入一个新开始状态 0 和 3 个 ϵ 转换将这三个 NFA 合并后得到的单个 NFA。 □

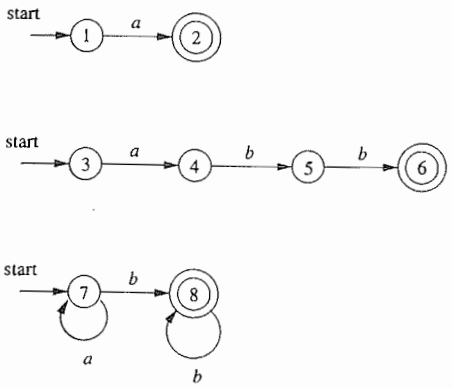


图 3-51 **a**、**abb** 和 **a** * **b** + 的 NFA

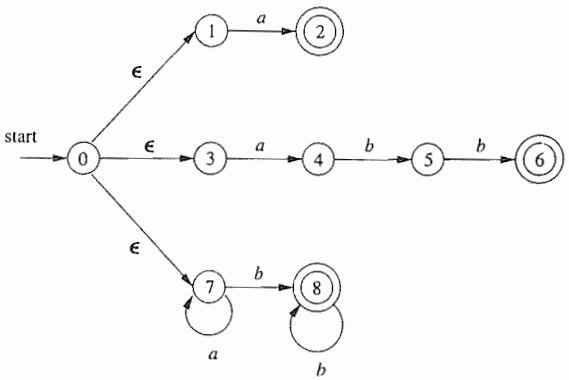


图 3-52 合并后的 NFA

3.8.2 基于 NFA 的模式匹配

如果词法分析器模拟了像一个图 3-52 所示的 NFA，那么它必须从它的输入中 *lexemeBegin* 所指的位置开始读取输入。当它在输入中向前移动 *forward* 指针时，它在每个位置上根据算法 3.22 计算当前的状态集。

在这个模拟 NFA 运行的过程中，最终会到达一个没有后续状态的输入点。那时，不可能有任何更长的输入前缀使得这个 NFA 到达某个接受状态，此后的状态集将一直为空。于是，我们

就可以判定最长前缀(与某个模式匹配的词素)是什么。

我们沿着状态集的顺序回头寻找，直到找到一个包含一个或多个接受状态的集合为止。如果集合中有多个接受状态，我们就选择和在 Lex 程序中位置最靠前的模式相关联的那个接受状态 p_i。我们将 *forward* 指针移回到词素末尾，同时执行与 p_i 相关联的动作 A_i。

例 3.27 假设我们有例 3.26 所示的模式，并且输入字符串以 *aaba* 开头。如果图 3.52 中的 NFA 从初始状态 0 的 ϵ-闭包，即{0, 1, 3, 7}，开始处理输入，那么它进入的状态集合的序列如图 3-53 所示。在读入第四个输入符号之后，我们处于一个空状态集中，因为在图 3-52 中没有在输入 *a* 上离开状态 8 的转换。

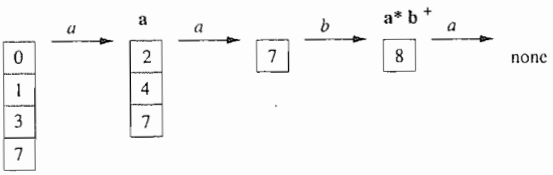


图 3-53 在处理输入 *aaba* 时进入的状态集的序列

因此，我们要向回寻找一个包含了某个接受状态的状态集。请注意，如图 3-53 所示，在读入 *a* 之后，我们所在的状态集包含状态 2，这表明模式 **a** 已经被匹配。然而在读入 *aab* 之后，我们在状态 8 中，这表明模式 $\mathbf{a}^*\mathbf{b}^+$ 被匹配；前缀 *aab* 是最长的使我们到达某个接受状态的前缀。因此我们选择 *aab* 作为被识别的词素，并且执行 A_3。这个动作应该包含一个返回语句，向语法分析器指明已经找到了一个模式为 $p_3 = \mathbf{a}^*\mathbf{b}^+$ 的词法单元。 □

3.8.3 词法分析器使用的 DFA

另一种体系结构和 Lex 的输出相似，它使用算法 3.20 中的子集构造法将表示所有模式的 NFA 转换为等价的 DFA。在 DFA 的每个状态中，如果该状态包含一个或多个 NFA 的接受状态，那么就要确定哪些模式的接受状态出现在此 DFA 状态中，并找出第一个这样的模式。然后将该模式作为这个 DFA 状态的输出。

例 3.28 使用子集构造法可以根据图 3-52 中的 NFA 构造得到一个 DFA。图 3-54 显示了这个 DFA 的一个转换图。图中的接受状态都用该状态所标识的模式作为标号。例如，状态{6, 8}有两个接受状态，分别对应于模式 **abb** 和 $\mathbf{a}^*\mathbf{b}^+$。由于前一个模式先被列出，因此该模式就是状态{6, 8}所关联的模式。 □

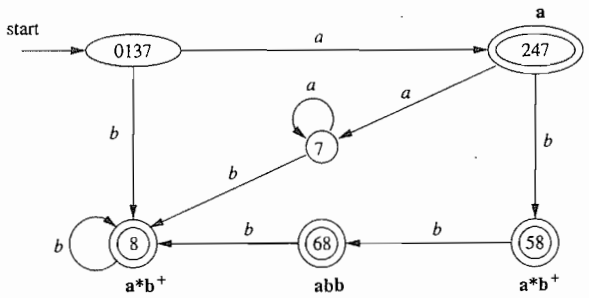


图 3-54 处理模式 **a**、**abb** 和 $\mathbf{a}^*\mathbf{b}^+$ 的 DFA 的转换图

在词法分析器中，我们使用 DFA 的方法与使用 NFA 的方法很相似。我们模拟这个 DFA 的运行，直到在某一点上没有后续状态为止(严格地说应该是下一个状态为 Ø，即对应于空的 NFA 状态集合的*死状态*)。此时，我们回头查找我们进入过的状态序列，一旦找到接受状态就执行与该

状态对应的模式相关联的动作。

例 3.29 假设图 3-54 中的 DFA 的输入为 *abba*。处理输入时进入过的状态序列为 0137、247、58、68。在读入最后一个 *a* 时，没有离开状态 68 的相应转换。因此，我们从后向前考察这个状态序列。在这个例子中，68 本身就是一个接受状态，对应于模式 p_2 = **abb**。 □

3.8.4 实现向前看运算符

回顾 3.5.4 节可知，Lex 模式 r_1/r_2 中的 Lex 向前看运算符/是必不可少的。因为有时为了正确地识别某个词法单元的实际词素，我们需要指明在这个词法单元的模式 r_1 之后必须跟着模式 r_2。在将模式 r_1/r_2 转化成 NFA 时，我们把/看成 ϵ，因此我们实际上不会在输入中查找/。然而，如果 NFA 发现输入缓冲区的一个前缀 xy 和这个正则表达式匹配时，这个词素的末尾并不在这个 NFA 进入接受状态的地方。实际上，这个末尾是在此 NFA 进入满足如下条件的状态 s 的地方：

1）s 在(假想的)/上有一个 ϵ 转换。

2）有一条从 NFA 的开始状态到状态 s(相应标号序列为 x)的路径。

3）有一条从状态 s 到 NFA 的接受状态(相应标号序列为 y)的路径。

4）在所有满足条件 1～3 的 xy 中，x 尽可能长。

如果这个 NFA 中只有一个在假想的/上的 ϵ 转换状态，那么就如例 3.30 所示，词素的末尾出现在最后一次进入该状态的地方。如果 NFA 在假想的/上有多个 ϵ 转换状态，那么如何寻找正确的状态 s 的问题就会变得困难得多。

例 3.30 图 3-55 的 NFA 识别例 3.13 中给出的 IF 模式。这个模式使用了向前看运算符。请注意，从状态 2 到状态 3 的 ϵ 转换就代表这个向前看运算符。状态 6 表明关键字 IF 的出现。然而，当进入状态 6 时，我们需要向回扫描到最晚出现的状态 2 才可以找到词素 IF。 □

DFA 中的死状态

从技术上讲，图 3-54 中的自动机并不是一个真正的 DFA。因为 DFA 中的每个状态在它的输入字母表中的每个符号上都有一个离开转换。这里我们省略了到达死状态 Ø 的转换，并且我们也省略了从这个死状态出发、在所有输入符号上到达其自身的转换。前面的 NFA 到 DFA 转换的例子中不存在从开始状态到达 Ø 的路径，但是图 3-52 中的 NFA 有这样的路径。

然而，当我们构造一个用于词法分析器的 DFA 时，重要的是，我们必须用不同的方式来处理死状态，因为我们必须知道什么时候已经不可能识别到更长的词素了。因此我们建议省略到达死状态的转换，并消除死状态本身。实际上这个问题要比看起来困难一些，因为一个 NFA 到 DFA 的构造过程可能会产生多个不可能到达接受状态的 DFA 状态。我们必须知道何时到达了一个这样的状态。3.9.6 节讨论了如何将这些状态合并为一个死状态，这使得识别这些状态变得容易。还要指出的是，如果我们使用算法 3.20 和 3.23 根据一个正则表达式构造出一个 DFA，那么得到在 DFA 中除 Ø 之外的所有状态都可到达某个接受状态。

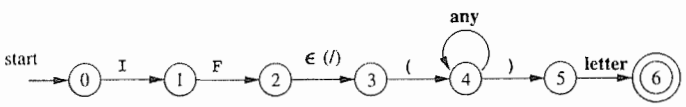


图 3-55 识别关键字 IF 的 NFA

3.8.5 3.8节的练习

练习3.8.1：假设我们有两个词法单元：(1)关键字 if，(2)标识符，它表示除 if 之外的所有由字母组成的串。请给出：

1）识别这些词法单元的 NFA。

2）识别这些词法单元的 DFA。

练习3.8.2：对如下的词法单元重复练习3.8.1：(1)关键字 while，(2)关键字 when，(3)标识符，它代表以字母开头、由字母和数位组成的字符串。

! **练习3.8.3**：假设我们修正 DFA 的定义，使得每个状态在每个输入符号上有零个或一个转换(而不是像标准的 DFA 定义中那样恰好有一个转换)。那么，有些正则表达式就可以具有相比按标准定义构造得到的 DFA 而言更小的“DFA”。给出这种正则表达式的一个例子。

!! **练习3.8.4**：设计一个算法来识别形如 r_1/r_2 的 Lex 向前看模式，其中 r_1 和 r_2 都是正则表达式。说明该算法如何处理如下输入：

1）$(abcd|abc)/d$

2）$(a|ab)/ba$

3）aa^*/a^*

3.9 基于 DFA 的模式匹配器的优化

我们将在本节中给出三个算法，这些算法用于实现和优化根据正则表达式构造得到的模式匹配器。

1）第一个算法可以用于 Lex 编译器，因为它不需构造中间的 NFA 就可以根据一个正则表达式直接构造得到 DFA。同时，得到的 DFA 的状态数也比通过 NFA 构造得到的 DFA 的状态数少。

2）第二个算法可以将任何 DFA 中具有相同未来行为的多个状态合并，从而使该 DFA 的状态数量减到最少。这个算法本身相当高效，它的时间复杂度仅有 $O(n\log n)$，其中 n 是被处理的 DFA 的状态数量。

3）第三个算法可以生成比标准二维表更加紧凑的转换表的表示方式。

3.9.1 NFA 的重要状态

在讨论如何根据一个正则表达式直接生成 DFA 之前，我们必须首先深入分析算法3.23构建 NFA 的过程，并考虑各种状态所扮演的角色。如果一个 NFA 状态有一个标号非 ϵ 的离开转换，那么我们称这个状态是*重要状态*(important state)。请注意，子集构造法(算法3.20)在计算 ϵ-*closure*(*move*(T, a))(即可以从 T 出发在输入 a 上到达的状态的集合)的时候，它只使用了集合 T 中的重要状态。也就是说，只有当状态 s 是重要的，状态集合 *move*(s, a)才可能是非空的。在子集构造法的应用过程中，两个 NFA 状态集合可以被认为是一致的(即把它们当作同一个集合来处理)条件是它们：

1）具有相同的的重要状态，且

2）要么都包含接受状态，要么都不包含接受状态。

如果这个 NFA 是使用算法3.23根据一个正则表达式生成的，那么我们还可以指出更多的关于重要状态的性质。重要状态只包括在基础规则部分为正则表达式中某个特定符号位置引入的初始状态。也就是说，每个重要状态对应于正则表达式中的某个运算分量。

此外，构造得到的 NFA 只有一个接受状态，但该接受状态(没有离开转换)不是重要状态。我们可以在一个正则表达式 r 的右端连接一个独特的右端结束标记符#，使得 r 的接受状态增加

一个在#上的转换，使之成为$(r)\#$的 NFA 的重要状态。换句话说，通过使用扩展的(augment)正则表达式$(r)\#$，我们可以在构造过程中不考虑接受状态的问题。当构造过程结束后，任何在#上有离开转换的状态必然是一个接受状态。

NFA 的重要状态直接对应于正则表达式中存放了字母表中符号的位置。使用抽象语法树来表示扩展的正则表达式是非常有用的。该语法分析树的叶子结点对应于运算分量，内部结点表示运算符。标号为连接运算符(·)、并运算符|、星号运算符 * 的内部结点分别称为 cat 结点、or 结点和 star 结点。我们可以使用 2.5.1 节中处理算术表达式的方法来构造一个正则表达式对应的抽象语法树。

例 3.31 图 3-56 是一个正则表达式的抽象语法树。其中的小圆圈表示 cat 结点。 □

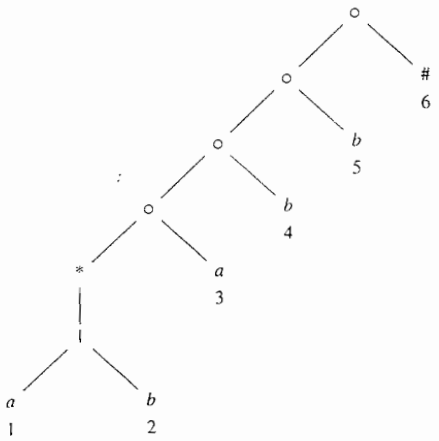


图 3-56 $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}\#$的抽象语法树

抽象语法树的叶子结点可以标号为 ϵ，也可以用字母表中的符号作为标号。对于每一个标号不为 ϵ 的叶子结点，我们赋予一个独有的整数。我们将这个整数称为叶子结点的位置(position)，同时也表示和它对应的符号的位置。请注意，一个符号可以有多个位置。比如，在图 3-56 中，a 有位置 1 和位置 3。抽象语法树中的这些位置对应于构造出的 NFA 中的重要状态。

例 3.32 图 3-57 显示了对应于图 3-56 中的正则表达式的 NFA，其中的重要状态已经被编号，而其他状态则用字母表示。我们很快就会看到，NFA 的编号状态和抽象语法树中的位置是如何对应的。 □

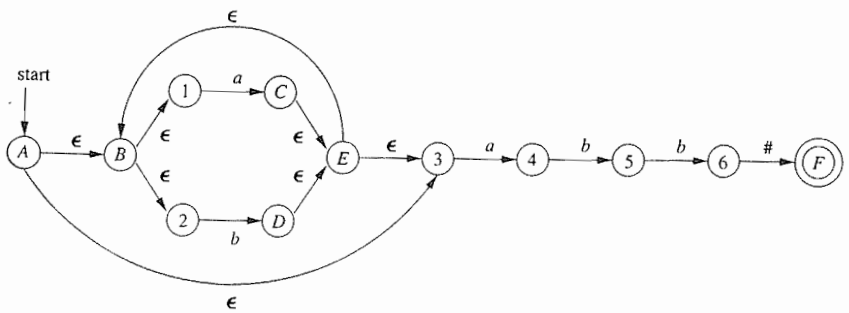


图 3-57 使用算法 3.23 构造得到的$(\mathbf{a}|\mathbf{b})^*\mathbf{abb}\#$的 NFA

3.9.2 根据抽象语法树计算得到的函数

要从一个正则表达式直接构造出 DFA，我们要首先构造它的抽象语法树，然后计算如下四个

函数：*nullable*、*firstpos*、*lastpos* 和 *followpos*。每个函数的定义都用到了一个特定增广正则表达式 $(r)\#$ 的抽象语法树。

1) *nullable*(n) 对于一个抽象语法树结点 n 为真当且仅当此结点代表的子表达式的语言中包含空串 ϵ。也就是说，这个子表达式可以“生成空串”或者本身就是空串，即使它也可能表示一些其他的串。

2) *firstpos*(n) 定义了以结点 n 为根的子树中的位置集合。这些位置对应于以 n 为根的子表达式的语言中某个串的第一个符号。

3) *lastpos*(n) 定义了以结点 n 为根的子树中的位置集合。这些位置对应于以 n 为根的子表达式的语言中某个串的最后一个符号。

4) *followpos*(p) 定义了一个和位置 p 相关的、抽象语法树中的某些位置的集合。一个位置 q 在 *followpos*(p) 中当且仅当存在 $L((r)\#)$ 中的某个串 $x = a_1 a_2 \cdots a_n$，使得我们在解释为什么 x 属于 $L((r)\#)$ 时，可以将 x 中的某个 a_i 和抽象语法树中的位置 p 匹配，且将位置 a_{i+1} 和位置 q 匹配。

例 3.33 考虑图 3-56 中对应于表达式 $(\mathbf{a}|\mathbf{b})^*\mathbf{a}$ 的 cat 结点 n。我们说 *nullable*(n) = false，因为这个结点生成所有以 a 结尾的由 a、b 组成的串；它不生成空串 ϵ。而另一方面，它下面的 star 结点是可以为空，它的正则表达式生成 ϵ 以及所有由 a、b 组成的串。

firstpos$(n) = \{1, 2, 3\}$。在由 n 对应的正则表达式生成的像 aa 这样的串中，该串的第一个位置对应于树中的位置 1；在像 ba 这样的串中，串的第一个位置来自于树中的位置 2。然而，当由 n 代表的正则表达式生成的串仅包含 a 时，这个 a 来自于位置 3。

lastpos$(n) = \{3\}$。也就是说，不管结点 n 的表达式生成什么串，该串的最后一个位置总是来自位置 3 上的 a。

followpos 的计算要困难一些，但是我们很快会给出计算这个函数的规则。下面是推导得到 *followpos* 值的一个例子：*followpos*$(1) = \{1, 2, 3\}$。考虑一个串 $\cdots ac \cdots$，其中 c 代表 a 或 b，且 a 来自位置 1。也就是说，这个 a 是由表达式 $(\mathbf{a}|\mathbf{b})^*$ 中的 $\mathbf{a}$ 生成的多个 a 之一。这个 a 后面可以跟随由同一表达式 $(\mathbf{a}|\mathbf{b})^*$ 生成的 a 或 b，此时 c 来自位置 1 或位置 2。也有可能这个 a 是表达式 $(\mathbf{a}|\mathbf{b})^*$ 生成的串的最后一个字符，那么 c 一定是来自位置 3 的 a。因此，1、2、3 就是可以跟在位置 1 后的位置。 □

3.9.3 计算 nullable、firstpos 及 lastpos

我们可以使用一个对树的高度直接进行递归的过程来计算 *nullable*、*firstpos* 和 *lastpos*。在图 3-58 中总结了计算 *nullable* 和 *firstpos* 的基本规则和归纳规则。计算 *lastpos* 的规则在本质上和计算 *firstpos* 的规则相同，但是在针对 *cat* 结点的规则中，子结点 c_1 和 c_2 的角色需要对调。

结点 n	*nullable*(n)	*firstpos*(n)
一个标号为 ϵ 的叶子结点	**true**	$\emptyset$
一个位置为 i 的叶子结点	**false**	$\{i\}$
一个 or-结点 $n = c_1 \mid c_2$	*nullable*(c_1) **or** *nullable*(c_2)	*firstpos*$(c_1) \cup$ *firstpos*(c_2)
一个 cat-结点 $n = c_1 c_2$	*nullable*(c_1) **and** *nullable*(c_2)	**if** (*nullable*(c_1)) *firstpos*$(c_1) \cup$ *firstpos*(c_2) **else** *firstpos*(c_1)
一个 star-结点 $n = c_1^*$	**true**	*firstpos*(c_1)

图 3-58 计算 *nullable* 和 *firstpos* 的规则

例 3.34 在图 3-56 的语法树的所有结点中，只有星号结点是可为空的。由图 3-58 可知，图中的所有叶子结点都是不可为空的，因为它们都对应于非 ϵ 运算分量。图 3-56 中的 or 结点是不可为空的，因为它的子结点都不可为空。图中的 star-结点是可空的，因为这是 star 结点的特征之一。最后，图 3-56 中的所有 cat 结点(至少包含一个不可为空的子结点)都是不可为空的。

对各个结点的 *firstpos* 和 *lastpos* 的计算结果显示在图 3-59 中，其中，*firstpos*(*n*)显示在结点 *n* 的左边，*lastpos*(*n*)显示在结点右边。每个叶子结点的 *firstpos* 和 *lastpos* 只包含它自身，这是由图 3-58 中关于非 ϵ 叶子结点的规则决定的。图 3-56 中的 or 结点的 *firstpos* 和 *lastpos* 分别是它的所有子结点的 *firstpos* 和 *lastpos* 的并集。针对 *star* 结点的规则是，它的 *firstpos* 及 *lastpos* 分别是它的唯一子结点的 *firstpos* 和 *lastpos*。

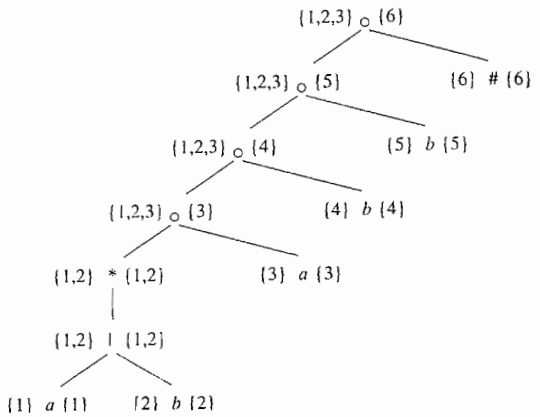


图 3-59 (**a**|**b**)* **abb#**的语法分析树的结点的 *firstpos* 和 *lastpos*

最后考虑最下面的 cat-结点，我们将把这个结点称为 *n*。要计算 *firstpos*(*n*)，我们首先考虑其左边的运算分量是否可为空。在这个例子里面，左运算分量可以为空，因此，*n* 的 *firstpos* 是它的各个子结点的 *firstpos* 的并集，也就是 $\{1,2\} \cup \{3\} = \{1,2,3\}$。图 3-58 中没有明确说明 *lastpos* 的运算规则，但是前面提到过，它的规则和 *firstpos* 的规则相同，只是需要互换子结点的角色。也就是说，要计算 *lastpos*(*n*)，我们需要知道它的右子结点(位置为 3 的叶子结点)是否可为空。它不可为空，因此 *lastpos*(*n*)就是它的右子结点的 *lastpos*，即 $\{3\}$。 □

3.9.4 计算 followpos

最后，我们来了解一下如何计算函数 *followpos*。只有两种情况会使得一个正则表达式的某个位置会跟在另一个位置之后：

1) 如果 *n* 是一个 cat 结点，且其左右子结点分别为 c_1、c_2，那么对于 *lastpos*(c_1)中的每个位置 *i*，*firstpos*(c_2) 中的所有位置都在 *followpos*(*i*)中。

2) 如果 *n* 是 star 结点，并且 *i* 是 *lastpos*(*n*)中的一个位置，那么 *firstpos*(*n*)中的所有位置都在 *followpos*(*i*)中。

例 3.35 现在让我们继续考虑那个贯穿全节的例子。回顾一下，*firstpos* 和 *lastpos* 已经在图 3-59中计算出来了。*followpos* 的计算规则 1 要求我们查看每个 cat 结点，并将它的右子结点的 *firstpos* 中的每个位置放到它的左子结点的 *lastpos* 中的各个位置的 *followpos* 中。对于图 3-59 中最下面的 cat 结点，该规则说位置 3 在 *followpos*(3)和 *followpos*(2)中。其上一个 cat 结点说 4 在 *followpos*(3)中，余下的两个 cat 结点告诉我们 5 在 *followpos*(4)中，6 在 *followpos*(5)中。

我们还必须对 star 结点应用规则 2。该规则告诉我们位置 1 和 2 既在 *followpos*(1)中又在 *followpos*(2)中，因为这个结点的 *firstpos* 和 *lastpos* 都是 $\{1,2\}$。图 3-60 给出了全部的 *followpos* 集合。 □

我们可以创建一个有向图来表示函数 *followpos*，其中每个位置有一个对应的结点，从位置 *i* 到位置 *j* 有一条有向边当且仅当 *j* 在 *followpos*(*i*)中。图 3-61 显示的有向图表示了图 3-60 所示的 *followpos* 函数。

毫不奇怪，表示 *followpos* 函数的有向图几乎就是相应的正则表达式的不包含 ϵ 转换的 NFA。

如果我们进行下面的处理，这个图就变成了这样的一个 NFA。

1）将根结点的*firstpos*中的所有位置设为开始状态。

2）在每条从i到j的有向边上添加位置i上的符号作为标号。

3）把和结尾#相关的位置当作唯一的接受状态。

位置n	*followpos*(n)
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	∅

图3-60 函数*followpos*

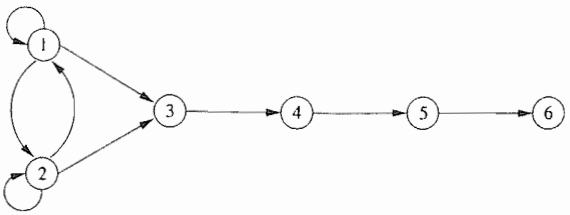


图3-61 表示函数*followpos*的有向图

3.9.5 根据正则表达式构建DFA

算法3.36 从一个正则表达式r构造DFA。

输入：一个正则表达式r。

输出：一个识别$L(r)$的DFA D。

方法：

1）根据扩展的正则表达式(r)#构造出一棵抽象语法树T。

2）使用3.9.3节和3.9.4节的方法，计算得到T的函数*nullable*、*firstpos*、*lastpos*和*followpos*。

3）使用图3-62中所示的过程，构造出D的状态集*Dstates*和D的转换函数*Dtran*。D的状态就是T中的位置集合。每个状态最初都是“未标记的”，当我们开始考虑某个状态的离开转换时，该状态变成“已标记的”。D的开始状态是*firstpos*(n_0)，其中结点n_0是T的根结点。这个DFA的接受状态集合是那些包含了和结束标记#对应的位置的状态。 □

```
初始化Dstates，使之只包含未标记的状态 firstpos(n0),
        其中n0 是(r)#的抽象语法树的根结点；
while ( Dstates中存在未标记的状态 S ) {
        标记S;
        for (每个输入符号 a ) {
                令 U为S中和 a对应的所有位置p的followpos(p)的并集；
                if ( U不在Dstates中 )
                        将 U作为未标记的状态加入到 Dstates 中；
                Dtran[S, a] = U;
        }
}
```

图3-62 从一个正则表达式直接构造一个DFA

例3.37 现在我们可以把我们的连续使用的例子的各个步骤综合起来，为正则表达式$r = (\mathbf{a}|\mathbf{b})^*\mathbf{abb}$构造一个DFA。$(r)$#的语法分析树如图3-56所示。我们观察到，在这棵语法分析树中，只有star结点使*nullable*为真。我们将函数*firstpos*和*lastpos*显示在图3-59中。函数*followpos*的值显示在图3-60中。

这棵树的根结点的*firstpos*的值是{1, 2, 3}，因此D的开始状态就是这个集合。我们称这个集合为A。我们必须计算*Dtran*[A, a]和*Dtran*[A, b]。在A的位置中，1和3对应于a，而2对应于b。因此*Dtran*[A, a] = *followpos*(1) ∪ *followpos*(3) = {1, 2, 3, 4}；*Dtran*[A, b] = *followpos*(2) = {1, 2, 3}。后一个集合就是A，因此不需要加入到*Dstates*中。但是前一个状态集B = {1, 2, 3, 4}是新状态，因此我们将它加入到*Dtrans*中并计算它的转换。完整的DFA如图3-63所示。 □

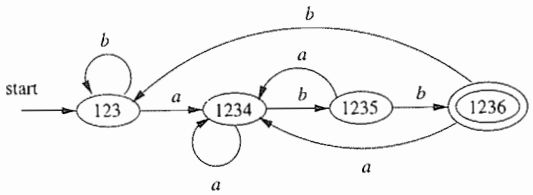


图 3-63 根据图 3-57 构造得到的 DFA

3.9.6 最小化一个 DFA 的状态数

对于同一个语言，可以存在多个识别此语言的 DFA。例如，图 3-36 和图 3-63 中的 DFA 都识别语言 $L((\mathbf{a}|\mathbf{b})^*\mathbf{abb})$。这两个 DFA 不但各个状态的名字不同，就连它们的状态个数也不一样。如果我们使用 DFA 来实现词法分析器，我们总是希望使用的 DFA 的状态数量尽可能地少，因为描述词法分析器的转换表需要为每个状态分配条目。

状态名字的问题是次要的。如果我们只需改变状态名字就可以将一个自动机转换成为另一个自动机，我们就说这两个自动机是同构的。图 3-36 和图 3-63 中的两个自动机不是同构的。然而，这两个自动机的状态之间有很紧密的关系。图 3-36 中的状态 A 和 C 实际上是等价的，因为它们都不是接受状态，且对任意输入，它们总是转到同一个状态——在输入 a 上转到 B，在输入 b 上转到 C。不仅如此，状态 A 和 C 的行为都和图 3-63 中的状态 123 相似。类似地，图 3.36 中状态 B 的行为和图 3-63 中状态 1234 的行为相似，状态 D 的行为和状态 1235 的行为相似，状态 E 的行为和状态 1236 的行为相似。

可以得出一个重要的结论：任何正则语言都有一个唯一的(不计同构)状态数目最少的 DFA。而且，从任意一个接受相同语言的 DFA 出发，通过分组合并等价的状态，我们总是可以构建得到这个状态数最少的 DFA。对于 $L((\mathbf{a}|\mathbf{b})^*\mathbf{abb})$，图 3-63 就是状态最少的 DFA，将图 3-36 中 DFA 的状态划分为{A，C}{B}{D}{E}然后合并等价状态就可以得到这个最小 DFA。

我们将给出一个将任意 DFA 转化为等价的状态最少的 DFA 的算法。该算法首先创建输入 DFA 的状态集合的分划。为了理解这个算法，我们要了解输入串是如何区分各个状态的。如果分别从状态 s 和 t 出发，沿着标号为 x 的路径到达的两个状态中只有一个是接受状态，我们说串 x 区分状态 s 和 t。如果存在某个能够区分状态 s 和状态 t 的串，那么它们就是可区分的(distinguishable)。

例 3.38 空串 ϵ 可以区分任何一个接受状态和非接受状态。在图 3-36 中，串 bb 区分状态 A 和 B，因为从 A 出发经过标号为 bb 的路径会到达非接受状态 C，而从 B 出发则到达接受状态 E。□

DFA 状态最小化算法的工作原理是将一个 DFA 的状态集合分划成多个组，每个组中的各个状态之间相互不可区分。然后，将每个组中的状态合并成状态最少 DFA 的一个状态。算法在执行过程中维护了状态集合的一个分划，分划中的每个组内的各个状态尚不能区分，但是来自不同组的任意两个状态是可区分的。当任意一个组都不能再被分解为更小的组时，这个分划就不能再进一步精化，此时我们就得到了状态最少的 DFA。

最初，该分划包含两个组：接受状态组和非接受状态组。算法的基本步骤是从当前分划中取一个状态组，比如 $A = \{s_1, s_2, \cdots, s_k\}$，并选定某个输入符号 a，检查 a 是否可以用于区分 A 中的某些状态。我们检查 $s_1, s_2, \cdots, s_k$ 在 a 上的转换，如果这些转换到达的状态落入当前分划的两个或多个组中，我们就将 A 分割成为多个组，使得 s_i 和 s_j 在同一组中当且仅当它们在 a 上的转换都到达同一个组的状态。我们重复这个分割过程，直到无法根据某个输入符号对任意个组进行分割为止。这个思想体现在下面的算法中。

状态最小化算法的原理

我们需要证明两个性质：仍然位于 Π_{final} 的同一组中状态不可能被任意串区分，以及最后存在于不同子集中的状态之间是可区分的。要证明第一个性质，需要对算法 3-39 中步骤 2 的迭代次数进行归纳。如果在步骤 2 的第 i 次迭代之后 s 和 t 在同一子组中，那么就不存在长度小于等于 i 的串可以将 s 和 t 区分开。请读者自行完成这个归纳证明。

第二个性质的证明也是通过对迭代次数的归纳来完成的。如果在步骤 2 的第 i 次迭代时状态 s 和 t 被放在不同的组中，那么必然存在一个串可以区分它们。归纳的基础很容易证明：当 s 和 t 放在初始分划的不同组中时，它们必然一个是接受状态，另一个是非接受状态。因此 ϵ 就可以区分它们。归纳步骤如下：必然存在一个输入符号 a 和状态 p、q，使得 s 和 t 在输入 a 上分别进入状态 p 和 q。并且 p 和 q 必定已经被放到不同的组中了。那么根据归纳假设，必然存在某个串 x 可以区分 p 和 q。因此可知 ax 能够区分 s 和 t。

算法 3.39 最小化一个 DFA 的状态数量。

输入：一个 DFA D，其状态集合为 S，输入字母表为 Σ，开始状态为 s_0，接受状态集为 F。

输出：一个 DFA D'，它和 D 接受相同的语言，且状态数最少。

方法：

1）首先构造包含两个组 F 和 $S-F$ 的初始划分 Π，这两个组分别是 D 的接受状态组和非接受状态组。

2）应用图 3-64 的过程来构造新的分划 Π_{new}。

```
最初，令 Π_new = Π;
for (Π中的每个组G) {
    将G分划为更小的组，使得两个状态s和t在同一小组中当且仅当对于所有
        的输入符号a，状态s和t在a上的转换都到达Π中的同一组;
    /* 在最坏情况下，每个状态各自组成一个组*/
    在Π_new中将G替换为对G进行分划得到的那些小组;
}
```

图 3-64 Π_{new} 的构造

3）如果 $\Pi_{new} = \Pi$，令 $\Pi_{final} = \Pi$ 并接着执行步骤 4；否则，用 Π_{new} 替换 Π 并重复步骤 2。

4）在分划 Π_{final} 的每个组中选取一个状态作为该组的代表。这些代表构成了状态最少 DFA D' 的状态。D' 的其他部分按如下步骤构建：

ⓐ D' 的开始状态是包含了 D 的开始状态的组的代表。

ⓑ D' 的接受状态是那些包含了 D 的接受状态的组的代表。请注意，每个组中要么只包含接受状态，要么只包含非接受状态，因为我们一开始就将这两类状态分开了，而图 3-64 中的过程总是通过分解已经构造得到的组来得到新的组。

ⓒ 令 s 是 Π_{final} 中某个组 G 的代表，并令 DFA D 中在输入 a 上离开 s 的转换到达状态 t。令 r 为 t 所在组 H 的代表。那么在 D' 中存在一个从 s 到 r 在输入 a 上的转换。注意，在 D 中，组 G 中的每一个状态必然在输入 a 上进入组 H 中的某个状态，否则，组 G 应该已经被图 3-64 的过程分割成更小的组了。

消除死状态

这个最小化算法有时会产生带有一个死状态的 DFA。所谓死状态就是在所有输入符号上都转

向自己的非接受状态。从技术上来讲，这个状态是必须的，因为在一个 DFA 中，从每个状态出发在每个输入符号上都必须有一个转换。然而，如 3.8.3 节所讨论的，我们需要知道在什么时候已经不存在被这个 DFA 接受的可能性了，这样我们才能知道已经识别到了正确的词素。因此，我们希望消除死状态，并使用一个缺少某些转换的自动机。这个自动机的状态比状态最少 DFA 的状态少一个，但是因为缺少了一些到达死状态的转换，所以严格地讲它并不是一个 DFA。

例 3.40 让我们重新考虑图 3-36 中给出的 DFA。初始分划包括两个组{A, B, C, D}，{E}，它们分别是非接受状态组和接受状态组。构造 Π_{new} 时，图 3-64 中的过程考虑这两个组和输入符号 a 和 b。因为组{E}只包含一个状态，不能再被分割，所以{E}被原封不动地保留在 Π_{new} 中。

另一个组{A, B, C, D}是可以被分割的，因此我们必须考虑各个输入符号的作用。在输入 a 上，这些状态中的每一个都转到 B，因此使用以 a 开头的串无法区分这些状态。但对于输入 b，状态 A、B 和 C 都转换到组{A, B, C, D}的某个成员上，而 D 转到另一个组中的成员 E 上。因此在 Π_{new} 中，组{A, B, C, D}被分割为{A, B, C}和{D}。这一轮得到的 Π_{new} 是{A, B, C}{D}{E}。

在下一轮中，我们可以把{A, B, C}分割为{A, C}{B}，因为 A 和 C 在输入 b 上都到达{A, B, C}中的元素，但 B 却转到另一个组中的元素 D 上。因此在第二轮之后，Π_{new} = {A, C}{B}{D}{E}。在第三轮中，我们不能够再分割当前分划中唯一一个包含多个状态的组{A, C}，因为 A 和 C 在所有输入上都进入同一个状态(因此也就在同一组中)。因此我们有 Π_{final} = {A, C}{B}{D}{E}。

现在我们将构建出状态最少 DFA。它有 4 个状态，对应于 Π_{final} 中的四个组。我们分别挑选 A、B、D 和 E 作为这四个组的代表。其中，状态 A 是开始状态，状态 E 是唯一的接受状态。它的转换函数如图 3-65 所示。例如，在输入 b 上离开状态 E 的转换到达状态 A，因为在原来的 DFA 中，E 在输入 b 上到达 C，而 A 是 C 所在组的代表。因为同样的原因，在输入 b 上离开 A 的状态回到 A 本身，而其他的转换都和图 3-36 中的相同。 □

状态	a	b
A	B	A
B	B	D
D	B	E
E	B	A

图 3-65 状态最少 DFA 的转换表

3.9.7 词法分析器的状态最小化

如果要将状态最小化算法应用于 3.8.3 节中生成的 DFA，我们必须在算法 3.39 中使用不同的初始分划。我们会将识别某个特定词法单元的所有状态放到对应于此词法单元的一个组中，同时把所有不识别任何词法单元的状态放到另一组。下面用一个例子来说明这个扩展。

例 3.41 对于图 3-54 的 DFA，初始分划为

$$\{0137,7\}\{247\}\{8,58\}\{68\}\{\emptyset\}$$

其中，状态 0137 和 7 分在同一组的原因是它们都没有识别任何词法单元；状态 8 和 58 分在一组的原因是它们都识别词法单元 **a** * **b** + 。请注意，我们添加了一个死状态 $\emptyset$，我们假设它在输入 **a** 和 **b** 时会转到它自身。这个死状态同时也是状态 8、58 和 68 在输入 a 上的目标状态。

我们必须将 0137 和 7 分开，因为它们在输入 a 上转到不同的组。我们也要把 8 和 58 分开，因为它们在输入 b 上转到不同的组。这样，所有的状态都自成一组。图 3-54 所示的 DFA 就是识别这三个词法单元的状态最少 DFA。请记住，被用作词法分析器的 DFA 通常会丢掉它的死状态，同时我们把所有消失的转换当作结束词法单元识别过程的信号。 □

3.9.8 DFA 模拟中的时间和空间权衡

最简单和最快捷的表示一个 DFA 的转换函数的方法是使用一个以状态和字符为下标的二维表。

给定一个状态和下一个输入字符，我们访问这个数组就可以找出下一个状态以及我们必须执行的特殊动作，比如将一个词法单元返回给语法分析器。由于词法分析器的 DFA 中通常包含数百个状态，并且涉及 ASCII 字母表中的 128 个输入字符，因此这个数组需要的空间少于一兆字节。

但是，在一些小型的设备中也可能使用编译器。对于这些设备来说，即使一兆内存也显得太大了。对于这种情况，可以应用很多方法来压缩转换表。比如，我们可以用一个转换链表来表示每个状态，这个转换链表由字符 - 状态对组成。我们在链表的最后存放一个默认状态：对于没有出现在这个链表中的字符，我们总是选择这个状态作为目标状态。

还有一个更加巧妙的数据结构，它既利用了数组表示法的访问速度，又利用了带默认值的链表的压缩特性。我们可以把这个结构看作四个数组，如图 3-66 所示[⊖]。其中的 *base* 数组用于确定状态 *s* 的条目的基准位置。这些条目位于数组 *next* 和 *check* 中。如果数组 *check* 告诉我们由 *base*[*s*] 给出的基准位置不正确，那么我们就使用数组 *default* 来确定另一个基准位置。

在计算 *nextstate*(*s*, *a*) 时，即计算状态 *s* 在输入 *a* 上的后继状态时，我们首先查看数组 *next* 和 *check* 中在位置 *l* = *base*[*s*] + *a* 上的条目，其中 *a* 被当作 0 ~ 127 之间的整数。如果 *check*[*l*] = *s*，那么这个条目是有效的，状态 *s* 在输入 *a* 上的后继状态就是 *next*[*l*]；如果 *check*[*l*] ≠ *s*，那么我们得到另一个状态 *t* = *default*[*s*]，并把 *t* 当作当前的状态重复这个过程。函数 *nextState* 的定义如下：

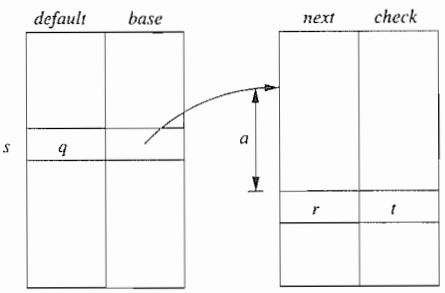


图 3-66 表示转换表的数据结构

```
int nextState(s, a) {
    if ( check[base[s] + a] == s ) return next[base[s] + a];
    else return nextState(default[s], a);
}
```

使用图 3-66 中所示数据结构的目的是利用状态之间的相似性来缩短 *next-check* 数组。例如，*s* 状态的默认状态 *t* 可能是一个“正在处理一个标识符”的状态，就像图 3-14 中的状态 10。而状态 *s* 可能是在读入字母 th 之后进入的状态。这里 th 既是关键字 then 的一个前缀，同时也可能是一个标识符的词素的前缀。当输入字符为 e 时，我们必须从状态 s 到达一个特别的状态。该状态记住我们已经看到了 the；当输入字符不等于 *e* 时，状态 *s* 的动作和状态 *t* 的动作相同。因此，我们将 *check*[*base*[*s*] + *e*] 的值设置为 *s*（以确认这个条目对于状态 *s* 有效），并将 *next*[*base*[*s*] + *e*] 的值置为前面提到的特殊状态。同时 *default*[*s*] 被设置为 *t*。

虽然我们可能无法选择适当的 *base* 值，使 *next* - *check* 的所有条目都被充分利用。经验表明，采用下述简单策略就可以有很好的效果：按照顺序将 *base* 值赋给各个状态，将各个 *base*[*s*] 的值设置为最小的、能够使得状态 *s* 的特殊条目的位置都尚未被占用的值。这个策略需要的空间只比最小可能值多一点点。

3.9.9 3.9 节的练习

练习 3.9.1：扩展图 3-58 中的表，使得它包含如下运算符：

1）?

2）+

练习 3.9.2：使用算法 3.36 将练习 3.7.3 中的正则表达式直接转换成 DFA。

⊖ 在实践中可能还有另一个以状态为下标的数组，如果某个状态相关的动作，那么这个数组的相应元素会指明这个动作。

! 练习3.9.3：我们只需要说明两个正则表达式的最少状态DFA同构，就可以证明这两个正则表达式等价。使用这种方法来证明下面的正则表达式$(\mathbf{a}|\mathbf{b})^*$，$(\mathbf{a}^*|\mathbf{b}^*)^*$以及$((\epsilon|\mathbf{a})\mathbf{b}^*)^*$相互等价。注意：你可能已经在完成练习3.7.3时构造出了这些表达式的DFA。

! 练习3.9.4：为下列的正则表达式构造最少状态DFA：

1) $(\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})$

2) $(\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$

3) $(\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})\ (\mathbf{a}|\mathbf{b})$

你有没有看出什么规律？

!! 练习3.9.5：为了证明例3.25中非正式给出的结论，说明正则表达式

$$(\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})\cdots(\mathbf{a}|\mathbf{b})$$

的任何DFA至少具有2^n个状态。在这个正则表达式中，$(\mathbf{a}|\mathbf{b})$在其尾部出现了$n-1$次。提示：观察练习3.9.4中的规律。各个状态分别表示了关于已输入串的哪些信息？

3.10 第3章总结

- *词法单元*。词法分析器扫描源程序并输出一个由词法单元组成的序列。这些词法单元通常会逐个传送给语法分析器。有些词法单元只包含一个词法单元名，而其他词法单元还有一个关联的词法值，它给出了在输入中找到的这个词法单元的某个实例的有关信息。
- *词素*。每次词法分析器向语法分析器返回一个词法单元时，该词法单元都有一个关联的词素，即该词法单元所代表的输入字符串。
- *缓冲技术*。为了判断下一个词素在何处结束，常常需要预先扫描输入字符。因此，词法分析器往往需要对输入字符进行缓冲。可以使用两个技术来加速输入扫描过程：循环使用一对缓冲区，以及在每个缓冲区末尾放置特殊的哨兵标记字符。该字符可以通知词法分析器已经到达了缓冲区末尾。
- *模式*。每个词法单元都有一个模式，它描述了什么样的字符序列可以组成对应于此词法单元的词素。那些和一个给定模式匹配的字(或者说字符串)的集合称为该模式的语言。
- *正则表达式*。这些表达式常用于描述模式。正则表达式是从单个字符开始，通过并、连接、Kleene闭包、“重复多次”等运算符构造得到的。
- *正则定义*。多个语言的复杂集合，比如用以描述一个程序设计语言所有词法单元的多个模式常常是通过正则定义来描述的。一个正则定义是一个语句序列，其中的每个语句定义了一个表示某正则表达式的变量。定义一个变量的正则表达式时可以使用已经定义过的变量。
- *扩展的正则表达式表示法*。为了使正则表达式更易于表达模式，一些附加的运算符可以作为缩写在正则表达式中使用。比如+(一个或多个)、?(零个或一个)以及字符类(由特定字符集中单个字符组成的字符串的集合)。
- *状态转换图*。一个词法分析器的行为经常可以用一个状态转换图来描述。它有多个状态。在搜寻可能与某个模式匹配的词素的过程中，各个状态代表了已读入字符的历史信息。它同时具有多条从一个状态到达另一个状态的转换(箭头)。每个转换都指明了下一个可能的输入字符，该字符将使词法分析器改变当前状态。
- *有穷自动机*。它是状态转换图的形式化表示。它指明了一个开始状态、一个或多个接受状态，以及状态集、输入字符集和状态间的转换集合。接受状态表明已经发现了和某个词法单元对应的词素。与状态转换图不同，有穷自动机既可以在输入字符上执行转换，也可以在空输入上执行转换。

- 确定有穷自动机。一个确定有穷自动机是一种特殊的有穷自动机。它的任何一个状态对于任意一个输入符号有且只有一个转换。同时它不允许在空输入上的转换。确定有穷自动机类似于状态转换图，对它的模拟相对容易，因此适于作为词法分析器的实现基础。
- 不确定有穷自动机。不是确定有穷自动机的自动机称为不确定的。NFA 通常要比确定有穷自动机更容易设计。词法分析器的另一种体系结构如下：对应于各个可能模式都有一个 NFA，并且我们使用表格来记录这些 NFA 在扫描输入字符时可能进入的所有状态。
- 模式表示方法之间的转换。我们可以把任意一个正则表达式转换为一个大小基本相同的 NFA，这个 NFA 识别的语言和该正则表达式识别的相同。更进一步，任何 NFA 都可以转换为一个代表相同模式的 DFA，虽然在最坏的情况下自动机的大小会以指数级增长，但是在常见的程序设计语言中尚未碰到这些情况。可以将任意一个确定或不确定有穷自动机转化为一个正则表达式，使得该表达式定义的语言和这个自动机识别的语言相同。
- Lex。有一系列的软件系统，包括 Lex 和 Flex，可以作为生成词法分析器的工具。用户通过扩展的正则表达式来描述各种词法单元的模式。Lex 将这些表达式转化为词法分析器。这个分析器实质上是一个可以识别所有模式的确定有穷自动机。
- 有穷自动机的最小化。对于每一个 DFA，都存在一个接受同样语言的最少状态 DFA。不仅如此，一个给定语言的最少状态 DFA(不计同构)是唯一的。

3.11 第 3 章参考文献

正则表达式首先由 Kleene 在 20 世纪 50 年代开始研究[9]。McCullough 和 Pitts[12]提出了一种描述神经活动的有穷自动机模型，而 Kleene 的兴趣就是描述那些可以用这些模型表示的事件。从那以后，正则表达式和有穷自动机在计算机科学中得到了广泛应用。

各种各样的正则表达式已经应用于很多流行的 UNIX 工具中，比如 awk、ed、egrep、grep、lex、sed、sh 和 vi 等。可移动操作系统接口(Portable Operating System Interface，POSIX)的标准文档 IEEE 1003 和 ISO/IEC 9945 中定义了 POSIX 扩展正则表达式，它们和最初的 UNIX 正则表达式非常相近，只有少量例外，比如字符类的助记表示方式。许多脚本语言，像 Perl、Python 和 Tcl，都采用了正则表达式，但常常使用不兼容的扩展表示方式。

我们熟悉的有穷自动机模型和算法 3.39 中的有穷自动机最小化方法由 Huffman[6]和 Moore[14]给出。而 Rabin 和 Scott[15]最先提出了不确定有穷自动机的概念，他们还给出了子集构造法，即算法 3.29。这个算法证明了确定自动机和不确定自动机在语言识别能力上是等价的。

McNaughton 和 Yamada[13]最先给出了一个利用正则表达式直接构造 DFA 的算法。3.9 节中描述的算法 3.36 最早被 Aho 用于构建 UNIX 正则表达式匹配工具 egrep，这个算法还被应用于 awk[3]中的正则表达式模式匹配例程。将不确定自动机用作中间表示的匹配方法首先由 Thompson[17]提出。该文还提出了直接模拟 NFA 的算法(算法 3.22)。这个算法被 Thompson 用于文本编辑器 QED 中。

Lesk 开发了 Lex 的第一个版本，随后 Lesk 和 Schmidt 用算法 3.36 编写了 Lex 的第二个版本[10]。此后出现了 Lex 的很多变体。GNU 版本的 Flex 及其文档可以在[4]下载。流行的 Lex 的 Java 版本包括 JFlex[7]和 JLex[8]。

在 3.4 节的练习 3.4.3 之前讨论的 KMP 算法来自[11]。可处理多个关键字的此算法的扩展版本可以在[2]中找到。Aho 在 UNIX 工具 fgrep 的第一个实现中使用了这个算法。

在[5]中完整地介绍了有关有穷自动机和正则表达式的理论，而[1]给出了字符串匹配技术的概述。

1. Aho, A. V., "Algorithms for finding patterns in strings," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), Vol. A, Ch. 5, MIT Press, Cambridge, 1990.

2. Aho, A. V. and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Comm. ACM* **18**:6 (1975), pp. 333–340.

3. Aho, A. V., B. W. Kernighan, and P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley, Boston, MA, 1988.

4. `Flex` home page `http://www.gnu.org/software/flex/`, Free Software Foundation.

5. Hopcroft, J. E., R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Boston MA, 2006.

6. Huffman, D. A., "The synthesis of sequential machines," *J. Franklin Inst.* **257** (1954), pp. 3–4, 161, 190, 275–303.

7. `JFlex` home page `http://jflex.de/` .

8. `http://www.cs.princeton.edu/~appel/modern/java/JLex` .

9. Kleene, S. C., "Representation of events in nerve nets," in [16], pp. 3–40.

10. Lesk, M. E., "Lex - a lexical analyzer generator," Computing Science Tech. Report 39, Bell Laboratories, Murray Hill, NJ, 1975. A similar document with the same title but with E. Schmidt as a coauthor, appears in Vol. 2 of the *Unix Programmer's Manual*, Bell laboratories, Murray Hill NJ, 1975; see `http://dinosaur.compilertools.net/lex/index.html` .

11. Knuth, D. E., J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM J. Computing* **6**:2 (1977), pp. 323–350.

12. McCullough, W. S. and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bull. Math. Biophysics* **5** (1943), pp. 115–133.

13. McNaughton, R. and H. Yamada, "Regular expressions and state graphs for automata," *IRE Trans. on Electronic Computers* **EC-9**:1 (1960), pp. 38–47.

14. Moore, E. F., "Gedanken experiments on sequential machines," in [16], pp. 129–153.

15. Rabin, M. O. and D. Scott, "Finite automata and their decision problems," *IBM J. Res. and Devel.* **3**:2 (1959), pp. 114–125.

16. Shannon, C. and J. McCarthy (eds.), *Automata Studies*, Princeton Univ. Press, 1956.

17. Thompson, K., "Regular expression search algorithm," *Comm. ACM* **11**:6 (1968), pp. 419–422.