第 3 章　扫描——理论和实践

本章讲解构造扫描器的过程中遇到的理论和实践问题。为了支持编译器的构造，扫描器的工作是把字符的输入流翻译成词法单元流，其中每个词法单元对应程序设计语言中的一个终结符号（参见 2.4 节）。更普遍地来说，扫描器所执行的操作是由与输入字符相关联的模式来触发的。与扫描相关的技术可以用在大多数需要识别输入结构的软件程序中。例如，网络数据包的处理、网页的显示，以及数字视频和音频媒体的解释都需要进行某种形式的扫描。

3.1 节会概述扫描器的工作原理。3.2 节会重温在第 2.2 节中引入的声明正则表达式的标记法，它特别适合于对词法单元进行形式化的定义和扫描器的自动化生成。3.4 节会讲解正则表达式和有限自动机之间的对应关系。作为一个例子，3.5 节介绍广泛使用的扫描器生成工具 Lex。Lex 使用正则表达式来产生完整的扫描器构件，它可以直接编译、单独进行部署，也可以作为更大项目的一部分来使用。3.6 节会简要介绍其他扫描器生成工具。

3.7 节会讨论构造扫描器并把它同编译器的其他部分集成所需考虑的一些实际问题，其中包括可能会对扫描带来复杂性的词法单元和上下文，避免性能瓶颈，以及如何从词法错误中恢复。

3.8 节会讲解像 Lex 这样的工具把正则表达式转换成可执行的扫描器所使用的理论基础。虽然严格来讲，对于构造编译器来说这部分知识并不是必需的，但是扫描的理论知识不仅非常简洁，相对比较直接，而且有助于读者理解扫描器的能力和局限性。

3.1　扫描器概述

扫描器的主要功能是把字符流转换成词法单元流。有时候，扫描器也会被称做是**词法分析器**（lexical analyzer, 或简称 lexer）。“扫描器”、“词法分析器”和 lexer 这些名称都是可以互换使用的。第 2 章介绍的 ac 扫描器比较简单，几乎任何有一定能力的程序员都可以胜任它的编码工作。本章会全面、系统地讲解扫描的方法，从而可以为完整的程序设计语言创建扫描器。

本章会引入形式化的标记法来说明词法单元的精确结构。乍看起来，这似乎并没有必要，因为绝大多数程序设计语言中能找到的词法单元结构都比较简单。然而，词法单元的结构实际上比看起来要更加复杂和难以捉摸。举例来说，在 C、C++和 Java 中的**字符串常量**是由双引号括起来的。字符串的内容可以包含除了双引号之外的任意字符序列，因为双引号会意味着字符串的结束。如果一个双引号要在字符串中以字面形式出现的话，就必须在前面使用一个反斜杠（进行转义）。那么这个简单的定义真的是完全正确的吗？在字符串中可以出现换行字符吗？在 C 语言中是不允许的，除非它也使用反斜杠进行转义。这种记法可以避免出现“逃逸字符串”，也就是由于缺少结束引号，从而会匹配到其他词法单元中

的字符。虽然 C、C++和 Java 语言都允许在字符串中出现转义之后的换行符，但是在 Pascal 中是不允许的。Ada 则更进一步，禁止出现所有不可打印字符（实际上是因为它们通常都是无法看到的）。类似地，是否允许出现空字符串（长度为 0）呢？这在 C、C++、Java 和 Ada 中是可以的，但是 Pascal 中是禁止的。在 Pascal 语言中，一个字符串是装满了字符的一个数组，而长度为 0 的数组是不允许的。

要确保清晰地说明这些词法规则，并且做到正确地实施这些规则，就有必要给词法单元一个精确的定义。形式化的定义还可以允许语言设计者预先避免一些设计上的缺陷。例如，几乎所有语言都提供了一些语法来说明特定种类的**有理数常量**。这些常量通常可以使用十进制数值形式来表示，如 0.1 和 10.01。那么，像 .1 和 10. 这样的记法是不是也应当允许出现呢？在 C、C++和 Java 中这样的记法是允许的，但是在 Pascal 和 Ada 中是不允许的，而其中的原因也很有意思。扫描器通常会试图匹配尽可能多的字符，因此 ABC 会被扫描为 1 个标识符，而不是 3 个标识符。现在来看一下字符序列 1..10，在 Pascal 和 Ada 中，它应当被解释为一个范围说明符（1 到 10）。然而，如果我们在词法单元定义中不小心的话，那么就可能会把 1..10 扫描成两个常量：1. 和 .10，这样就会导致一个立即（而且意外）的语法错误。事实是两个常量不能相邻出现的规则是在上下文无关文法中描述的，因此是由分析器，而不是扫描器来强制实施的。

在给定了词法单元和程序结构的形式化规范之后，就可以检查语言中的设计缺陷。举例来说，我们可以分析所有可以彼此相邻出现的词法单元对，以确定如果两个词法单元连接起来的话，是不是可能会导致扫描上的错误。如果是这样的话，就需要在两者之间添加分隔符。对于相邻的标识符和保留字来说，一个空格就足以把两个词法单元分开了。然而，有时候可能需要对词法或程序语法进行重新设计。这里需要指出的是：语言设计是一项极其复杂的任务，而形式化的规范能够在设计完成之前就发现其中的缺陷。

不管要识别的词法单元是什么样的，所有的扫描器都会执行几乎相同的功能。因此，如果从头编写一个扫描器就意味着要重新实现对于所有扫描器都一样的构件；这也就会导致大量重复式投入。扫描器生成程序的目标就是为了把构造扫描器的工作限制为只需要提供扫描器要识别的词法单元的说明即可。使用一种形式化的记法，我们就可以告诉扫描器生成程序我们想要识别什么样的词法单元。接着扫描器会负责产生一个满足我们所给规范的扫描器。有些生成程序不会产生一个完整的扫描器，而是生成可以用于某个标准驱动程序的表格，而把所生成的表格同标准驱动器结合起来，就可以得到想要的定制扫描器。

对扫描器生成程序进行编程是一个**声明式编程**（declarative programming）的例子。也就是说它和普通的**过程式编程**（procedural programming）是不同的：我们不会告诉扫描器生成程序应该**如何**扫描，而只是告诉它要扫描**什么**。这是一种更为高级的方法，而且在许多情况下是一种很自然的方式。计算机科学领域最近的许多研究都在转向声明式编程的风格；典型的例子包括数据库查询语言和 Prolog（一种“逻辑式”程序设计语言）。声明式编程在特定的领域中是最为成功的，如扫描，其中必须要自动进行的实现决策的范围是很有限的。尽管如此，计算机科学家们的一个由来已久（却依然未能实现）的目标就是从一种源语言属性和目标机器的说明自动生成一个完整的产品级质量的编译器。

虽然本书的主要目标是产生正确的编译器，有时候性能也是一个很现实的考虑，特别是对广为使用的“产品编译器”来说。令人惊奇的是，虽然扫描器执行的任务很简单，但

是如果实现不好的话，它也会成为一个重要的性能瓶颈。这是因为扫描器必须逐个字符处理程序的所有代码。

假设要实现一个速度非常快的编译器，它可以在几秒钟就编译一个程序。可以把目标定为每分钟 30 000 行代码（也就是每秒 500 行）。例如，Turbo C++这样的编译器就可以达到这样的速度。如果每行平均包含 20 个字符的话，那么编译器就必须在每秒钟扫描 10 000 个字符。在每秒执行 10 000 000 条指令的处理器上，即使除了扫描之外什么都不做，对于每个输入字符来说，最多也只能花费 1 000 条指令。但是因为扫描并不是编译器要做的唯一事情，所以可能每个字符 250 条指令更为现实一些。如果考虑到在一个典型的处理器上即使简单的赋值也需要花费好几条指令的话，那么这是一个相当紧张的预算。虽然现在很容易找到比这更快的处理器，但是每分钟 30 000 行依然是一个不容易达到的速度，因此如果扫描器的编码有问题的话，那么显然它会显著影响一个编译器的性能。

3.2 正则表达式

正则表达式非常适合被用来描述各种简单（但是可能是无限数量）的字符串集合。它们之所以在实践中很重要，是因为可以被用来描述在程序设计语言中使用的词法单元的结构。具体来讲，读者可以使用正则表达式来开发扫描器生成工具。

除了编译器之外，正则表达式也广泛用于其他计算机应用程序中。UNIX®中的实用程序 grep 用它们来定义文件中的查找模式。UNIX shell 在命令中使用文件列表时也支持有限的正则表达式形式。绝大多数编辑器都会提供一个“上下文查找”命令，在其中通常允许使用正则表达式来描述想要的匹配。

由正则表达式定义的字符串集合被称做是**正则集合**（regular set）。为了扫描的目的，一个词法单元类是由正则表达式定义的一个正则集合。词法单元类的特定实例有时候被称做是**单词**（lexeme）；然而，我们会简单把词法单元类中的字符串称做是该词法单元的一个**实例**。例如，如果字符串 abc 可以匹配用来定义合法标识符词法单元集合的正则表达式的话，那么就把它称做是一个标识符。

正则表达式的定义首先会从一个有限的字符集开始，它被称做是字母表（vocabulary），用 Σ 来表示。这个字母表通常就是计算机所使用的字符集合。如今使用非常广泛的是 ASCII 字符集，它包含 128 个字符。然而，Java 语言使用的是 Unicode 字符集。这个集合中除了所有的 ASCII 字符之外，还包括大量其他字符。

空字符串（null）是允许出现的（用 λ 表示）。这个符号表示的是一个空的缓冲区，其中没有匹配到任何字符。它同样可以用来表示一个词法单元中的可选组成部分。因此，一个整数字面常量可以以正号或负号开始，而如果它是无符号整数的话，也可以用 λ 作为开始。

字符串是由字符集中的字符通过连接来构造的，也就是说把单个字符连起来形成一个字符串。当字符被连接字符串的时候，字符串的长度会增长。例如，字符串 do 的构造方法是首先把 d 连接到 λ 之上，然后再把 o 连接到字符串 d 上。空字符串同任意字符串 s 进行连接的结果依然是 s。也就是说，$s\lambda \equiv \lambda s \equiv s$。把 λ 连接到一个字符串就好像是把 0 加到一个整数上一样——都不会产生任何作用。

连接操作也可以按照下面的方式扩展到字符串集合上。假设 P 和 Q 是字符串集合。符号∈用来表示集合成员关系。如果 s_1∈P 且 s_2∈Q，那么字符串 s_1s_2∈(P Q)。要表示较小的有限集合，可以很方便地把它们的所有元素都列出来，这些元素可以是单个字符或者字符串。圆括号可以用来分隔表达式，而多选操作符 | 则被用来分隔多选分支。以包含 10 个数字的集合 D 为例，可以把它定义为 D = (0|1|2|3|4|5|6|7|8|9)。在本教材中，我们通常不会把所有的多选分支都完整列出来，而是使用类似 (0|. . .|9) 这样的简写形式。注意省略号（. . .）并不属于正则表达式标记法中的一部分。

元符号（meta-character）指的是任何分隔符号或正则表达式操作符。如果把元符号用作普通字符的话，就必须把它用引号括起来，以免造成混淆（也可以把任意字符或字符串用引号括起来，但是为了提高可读性，应当避免不必要的引号）。下面 6 个符号是元字符：( )' * + |。正则表达式 ('(' | ')' | ; | ,) 定义的是在程序设计语言中可能会用到的 4 个单字符词法单元（分别是左圆括号、右圆括号、分号和逗号）。把圆括号用引号括起来是为了说明它们的含义是单独的词法单元，而不是用来作为更大的正则表达式之内的分隔符。

多选结构也可以扩展到字符串集合之上。假设 P 和 Q 是两个字符串集合。那么字符串 s∈(P|Q)当且仅当 s∈P 或者 s∈Q。举例来说，如果 LC 是小写字母集合，而 UC 是大写字母集合，那么（LC|UC）就可以表示所有字母的集合（不论大小写）。

大集合或无限集合可以使用在有限的字符和字符串集合之上的操作来很方便地进行表示。连接和多选操作都可以使用。另外还可以出现第三个操作：**Kleene 闭包**（closure）。Kleene 闭包操作符是后缀操作符 *。举例来说，假设 P 是一个字符串集合。那么 P*表示的是由 P 中的 0 个或多个元素（可以重复）进行连接所构成的所有字符串（使用 λ 来表示选择了 0 个元素的情形）。例如，LC* 是只由小写字母构成的任意长度的所有单词的集合（包括了长度为 0 的单词 λ）。

精确地讲，字符串 s∈P*，当且仅当 s 可以被分解为 0 个或多个字串：$s=s_1s_2...s_n$，使得每个 s_i∈P ($n \geqslant 0, 1 \leqslant i \leqslant n$)。这里我们允许 n=0，从而 λ 总是属于 P*。

这样我们就介绍了在正则表达式中使用的所有操作符，下面可以给出正则表达式的定义：

- ∅是一个表示空集（不包含任何字符串的集合）的正则表达式。虽然很少会用到∅，但是为了完整性起见，才把它包含进来。
- λ 是表示只包含空字符串的集合的正则表达式。这个集合与空集是不一样的，因为它包含了一个元素（空字符串）。
- 符号 s 是一个表示 {s} 的集合：该集合中只包含单个的符号 s∈Σ。
- 如果 A 和 B 都是正则表达式，那么 A|B、AB 和 A*都是正则表达式。它们分别表示的是相应正则集合的多选操作、连接操作和 Kleene 闭包。

每个正则表达式都表示一个正则集合。任意的有限字符串集合都可以使用形为（$s_1|s_2|...|s_k$）的正则表达式来表示。因此，ANSI C 的保留字可以被定义为（auto|break|case|...）。

有时候，也可以使用下面一些额外的操作。严格来讲，它们并不是必需的，因为它们的效果可以使用上面 3 种标准正则操作符（多选、连接和 Kleene 闭包）来获得（虽然可能会有点笨拙）：

- P^+，有时候被称做是**正闭包**（positive closure），表示的是包含一个或多个 P 中的

字符串连接起来的所有字符串：$P^* = (P^+ \mid \lambda)$且 $P^+ = P\ P^*$。例如，表达式$(0|1)^+$指的是包含一个或多个比特的所有字符串。

- 如果 A 是一个字符集合，Not(A) 表示的是 (Σ - A)，也就是在 Σ 中不包含在 A 中的所有字符。因为 Not(A) 永远不可能比 Σ 大，而 Σ 是有限的，所以 Not(A)也是有限的，因此它是正则集合。Not(A) 不包含 λ，因为 λ 不是一个字符（它是一个长度为 0 的字符串）。举例来说，Not(Eol) 是除了 Eol（也就是行结束符号，在 Java 或 C 语言中是 \n）之外的所有字符组成的集合。

  我们也可以将 Not() 从字母表 Σ 扩展到字符串之上。如果 S 是一个字符串集合，那么我们可以把 Not(S) 定义为(Σ*-S)，也就是说除了 S 之外的所有字符串的集合。虽然 Not(S) 通常是无限的，但是如果 S 是正则的话，那么它也是正则的（练习 18）。
- 如果 k 是一个常量，那么集合 A^k 表示的是把 A 中的 k 个（可以是不同的）字符串连接起来构成的所有字符串。也就是说 A^k=(AAA…)（重复 k 次）。因此 $(0|1)^{32}$ 表示的是所有正好 32 位的比特字符串的集合。

3.3 示　　例

接下来会给出一些例子，讲解如何使用正则表达式来定义一些常见的程序设计语言词法单元。在这些定义中，D 是 10 个数字的集合，而 L 是所有大小写字母的集合。

- Java 或 C++语言中的单行注释以 // 开始，并以 Eol 作为结束，它可以定义为

  ```
  Comment = //(Not(Eol))* Eol
  ```

  这个正则表达式描述的注释以两个斜杠作为开始，当遇到第一个行结束符号时结束。在注释内部只允许出现除了行结束符之外的任意字符序列。（这样可以确保遇到的第一个行结束符就是注释的结束）。
- 定点常量（如 12.345）可以被定义为

  ```
  Lit = D+.D+
  ```

  在小数点的两边都必须有一个或多个数字，因此像 .12 和 35. 这样的数字是不包括在内的。
- 可选符号的整数常量可以被定义为

  ```
  IntLiteral = ('+'| - |λ)D+
  ```

  整数常量是在一个或多个数字之前跟着一个正号、负号或者没有符号（λ）。这里的正号是用引号括起来的，其目的是为了使之不会同正闭包操作符产生混淆。
- 一个更为复杂的例子是用 ## 词法单元分隔开的注释，并且在注释中允许出现单个的#符号：

  ```
  Comment2 = ## ((#|λ)Not(#))* ##
  ```

  在注释体中出现的任意 # 符号都必须紧跟着一个非 # 符号，从而在注释中间就不

会遇到注释的结束符号 ##。

所有的有限集合都是正则的。然而，有些（但不是全部）无限集合也是正则的。例如，考虑匹配的括号形式 [[[. . .]]]。该集合可以被形式化定义为 $\{[^m]^m \mid m \geq 1\}$，并且可以证明该集合不是正则的（练习 14）。问题是任何试图定义它的正则表达式都无法得到全部匹配的嵌套，或者是会包含其他并不需要的字符串。

另外，写一个上下文无关文法来精确定义匹配的括号是非常容易的。而且，所有正则集合都可以通过上下文无关文法来定义。因此，上述括号的例子说明了上下文无关文法是一种比正则表达式更为强大的描述机制。然而，正则表达式对于描述词法单元这样的语法也是足够的。而且，对于每个正则表达式来说，可以创建一种高效的装置用来恰好识别能够匹配该正则表达式模式的所有字符串，这种装置被称为**有限自动机**（finite automaton）。

3.4　有限自动机和扫描器

有限自动机（finite automaton，复数形式为 finite automata，缩写为 FA）可以用来识别由正则表达式表示的词法单元。FA 是一个简单的、理想化的计算机，可以识别属于正则集合的字符串。一个 FA 包括下列内容：

- 一个有限的**状态**集合。
- 一个有限的**字母表**，记为 Σ。
- 一个**转移**（或**动作**）集合，每个转移都是从一个状态到另外一个状态，并且以 Σ 中的字符作为标签。
- 一个特殊的状态被当做是**开始状态**。
- 一个状态子集，被称做是**接受状态**，或**最终状态**。

有限自动机的这些构成部分可以使用图 3.1 中的形式进行图形化的表示。

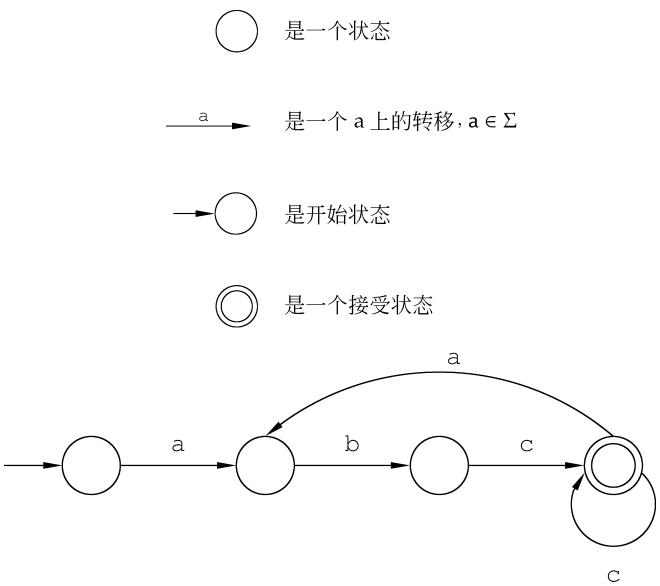


图 3.1　有限自动机图形的构成元素，以及用它们构造的可以识别 $(a\ b\ c^+)^+$的一个自动机

使用图 3.1 中所给的这些构成部分，可以把有限自动机以图形化的方式表示为**转换图**（transition diagram）。给定一个转换图，我们首先从开始状态作为起点。如果下一个字符与从当前状态出发的某个转移（的标签）匹配的话，那么就转到它指向的状态。如果没有可用的转移，我们就停下来。如果结束在一个接受状态，那么所读到的字符序列就是一个合法的词法单元；否则，看到的就不是合法词法单元。在图 3.1 中所给的转换图中，合法的词法单元是由正则表达式（a b c^+）$^+$描述的字符串集合。

作为一种缩写形式，每个转移之上可以用多于一个字符作为标签（例如，可以使用 Not（c））。如果当前的输入字符可以匹配转移之上的任意字符的话，就可以选取该转移。

3.4.1 确定性的有限自动机

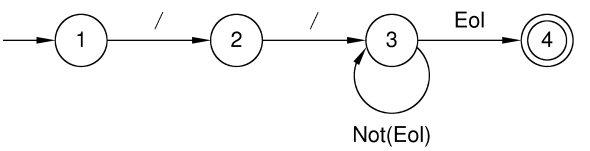
如果一个 FA 对于每个给定状态和输入字符，都拥有唯一的转移的话，它就是一个**确定性的有限自动机**（deterministic finite automaton，DFA）。DFA 易于编程，通常会被用来驱动扫描器。DFA 在计算机中可以很方便地使用**转移表**（transition table）来进行表示。转移表 T 是一个二维数组，分别是 DFA 状态和字母表中的符号。每个表项是一个 DFA 状态，或者是错误标记（通常可以使用空白表项来表示）。如果我们在状态 s 中，而读到的字符是 c，那么我们要访问的下一个状态就是 T[s,c]；而如果在 T[s,c]中包含的是一个错误标记的话，就说明当前的词法单元并不能使用 c 来接着进行扩展。例如，下面的正则表达式定义的是 Java 或 C++语言中的单行注释：

```
//(Not(Eol))*Eol
```

它可以使用在图 3.2（a）中所示的 DFA 来识别。在图 3.2（b）中给出了相应的转移表。

一个完整的转移表中会为每个字符包含一行。为了节省空间，有时候会采用**表格压缩**（table compression）。在这种情况下，只有非出错的表项才会显式表示在表格中。这种结构可以通过使用哈希或链表结构来实现[CLRS01]。

任何正则表达式都可以被转换为一个接受该正则表达式所表示的字符串集合（作为其合法词法单元）的 DFA。这样的转换可以由程序员手动完成，也可以使用扫描器生成工具来进行。



(a) 转换图

状态	字符				
	/	Eol	a	b	...
1	2				
2	3				
3	3	4	3	3	3
4					

(b) 对应的转移表

图 3.2　用来识别单行注释的 DFA

DFA 的编码

DFA 可以使用以下两种编码方式：

（1）表格驱动。

（2）显式控制。

在**表格驱动**（table-driven）的方式中，定义 DFA 动作的转移表会显式表示在一个运行时表格中，由一个驱动程序进行“解释”。在**显式控制**（explicit control）的方式中，定义 DFA 动作的转移表会显式地出现在程序的控制逻辑中。通常来说，每个程序语句会对应不同的 DFA 状态。例如，假设 CurrentChar 是当前的输入字符，而文件结尾使用一个特殊的字符值 Eof 来表示。使用前面所给的 Java 注释的 DFA，上面的两种方法分别会产生在图 3.3 和图 3.4 中所给的程序。

```
/*   假设 CurrentChar 中包含的是要扫描的第一个字符   */
State ← StartState
while true do
    NextState ← T[State, CurrentChar]
    if NextState = error
    then break
    State ← NextState
    CurrentChar ← READ( )
if State ∈ AcceptingStates
then  /* 返回或处理合法的词法单元 */
else  /* 报告一个词法错误 */
```

图 3.3　用来解释转移表的扫描器驱动

表格驱动的方式通常是由扫描器生成工具来产生的；它是与词法单元无关的。它使用一个简单的驱动器来扫描任意的词法单元，并假设转移表已经被正确地保存在 T 中。显式控制的形式可以自动生成，也可以手动编写。要扫描的词法单元会被“硬编码”到代码中。这种形式的扫描器通常比较容易阅读，而且经常是更加高效的，但是它是特定于单个的词法单元定义的。

下面给出的是另外两个正则表达式与它们相应的 FA 的例子：

（1）类似 FORTRAN 语言中的实数常量（要求在小数点一边或者两边包含数字，也可以只是一个数字串）可以被定义为

```
RealLit = (D+ (λ | . ) ) | (D*.D+)
```

图 3.5（a）给出了它对应的 DFA。

（2）另外一种标识符的形式中包含有字母、数字和下划线。它以字母开头，并且不允许出现相邻的下划线，也不允许以下划线结束。它可以被定义为

```
ID = L (L|D)* ( _ (L | D)+)*
```

这个定义包括了例如 sum 或 unit_cost 这样的标识符，但是不包含-one，two_和 grand_total。图 3.5（b）给出了它对应的 DFA。

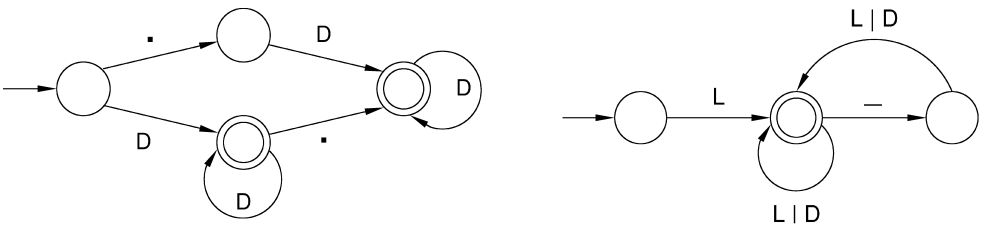
变换器

图 3.3 和图 3.4 中的扫描器在输入流中的某个点开始处理字符。它们最终会接受相应的词法单元，或者报告一个词法错误。它通常可以用于扫描器，不仅可以用来识别词法单元，还可以为发现的词法单元关联相应的**语义值**（semantic value）。例如，扫描器会发现输入 431 是一个整数常量，但是它最好还能在这个词法单元之上关联好 431 这个值。

```
/*   假设CurrentChar中包含的是要扫描的第一个字符          */
if CurrentChar = '/'
then
    CurrentChar ← READ( )
    if CurrentChar = '/'
    then
        repeat
            CurrentChar ← READ( )
        until CurrentChar ∈ {Eol, Eof}
    else   /* 报告一个词法错误 */
else   /* 报告一个词法错误 */
if CurrentChar = Eol
then   /* 结束识别注释 */
else   /* 报告一个词法错误 */
```

图 3.4　显式控制的方式

除了简单接受词法单元之外，还会对输入进行分析或转换的 FA 被称做是变换器（transducer）。图 3.5 中所示的 FA 会识别某种特定的常量和标识符。用来识别常量的变换器要负责处理用来表示该常量的相应的比特模式。处理标识符的变换器可以只保留该标识符的名称即可。对于一些语言来说，可以进一步要求扫描器使用一个**符号表**（symbol table）来区分标识符的类型。



(a) 浮点常量　　(b) 包含内嵌下划线的标识符

图 3.5　两个 DFA 的例子

通过在扫描器中添加基于状态转换的适当动作就可以把它转换成一个变换器。考虑在图 3.3 中所给的表格驱动的扫描器。图 3.2(b)中所给的转移表给出了当前状态和输入符号的下一个状态。可以为转移表建立一个与它共存的**动作表**（action table）。基于当前状态和输入符号，动作表中会保存 FA 应当执行的相应转移的动作。这样的信息可以使用一个整数进行编码，从而可以使用一个 switch 语句来选择适当的动作序列。如果采用面向对象的方法，就可以把动作编码为一个对象实例，在其中包含执行该动作的方法。

3.5　扫描器生成工具 Lex

下面讲解扫描器生成工具的设计实例，首先介绍一个非常流行的扫描器生成工具 Lex，然后会简要介绍几种其他扫描器生成工具。

Lex 是由 AT&T Bell 实验室的 M. E. Lesk 和 E. Schmidt 开发的。它的主要用于在 UNIX 操作系统下运行的使用 C 或 C++编写的程序。Lex 可以产生用 C 编写的完整的扫描器模块，并且可以在对它编译之后，把它同其他编译器模块链接在一起。Lex 的完整描述和它的使

用手册可以参考 [LS83] 和 [Joh83]。Flex [Pax] 是对 Lex 的一种重新实现，它的使用非常广泛，而且是免费发布的，它可以产生速度更快、更加可靠的扫描器。JFlex 是用在 Java 之上的一种类似工具 [KD]。通常来说，可以把正确的 Lex 扫描器说明不加修改直接用于 Flex。

图 3.6 给出了 Lex 的工作过程，它包含以下步骤：

（1）Lex 的输入是扫描器的说明，它用来定义要扫描的词法单元以及如何对它们进行处理。

（2）Lex 会生成由 C 编写的完整扫描器。

（3）该扫描器会被编译，并与其他编译器组成部分进行链接，创建完整的编译器。

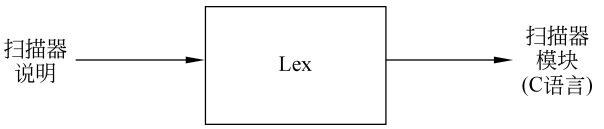


图 3.6　Lex 扫描器生成工具的工作过程

使用 Lex 可以节省手动编写扫描器要花费的大量开销。关于扫描器的许多底层细节（如高效地读入字符、对它们进行缓存、根据词法单元定义来匹配字符，等等）都不需要显式进行编码。这样就可以关注于词法单元的字符结构，以及如何对它们进行处理。

本节的主要目标是讲解如何把正则表达式和相关信息提供给扫描器生成工具。学习 Lex 的一种比较好的方式是先从这里讲的简单例子作为基础，然后逐渐对它们进行扩展来解决手上的实际问题。对于没有经验的读者来说，Lex 的规则看起来可能会觉着过于复杂。最好牢记这里的关键总是把对词法单元的说明写成正则表达式。余下的工作就只是简单地提高效率和处理各种细节。

3.5.1　定义 Lex 中的词法单元

Lex 的扫描方法很简单。它允许用户把正则表达式同用 C 或 C++编写的命令关联起来。当读入的是与正则表达式匹配的输入字符时，就会执行与之关联的命令。Lex 用户不用说明如何匹配词法单元，而只需要提供正则表达式即可。与之关联的命令会说明在匹配到某个特定词法单元的时候，应当做什么样的动作。

Lex 会创建一个文件 lex.yy.c，它包含一个返回值是整数的函数 yylex( )。这个函数通常由分析器来调用，每次会获取一个词法单元。yylex( )返回的值是 Lex 扫描到的词法单元的词法单元代码。像空格这样的词法单元会由与它们关联的命令简单删掉，而不会返回任何东西。扫描会一直继续，直到执行到一个含有 return 语句的命令。

图 3.7 给出了一个简单的 Lex 定义，它描述的是第 2 章介绍的 ac 语言中的 3 个关键字——f, i 和 p。当找到一个字符串可以匹配这 3 个保留字中的任意一个时，就会返回相应的词法单元代码。返回的词法单元代码一定要同分析器期望的代码是完全相同的，这一点非常关键。如果它们不相同的话，那么分析器就无法看到与扫描器产生的相同的词法单元序列。这样会造成分析器会根据错误的词法单元流生成假的语法错误。

扫描器和分析器通常会共享词法单元代码的定义，从而可以确保二者看到的值是一致