第3章 词法分析

3.1 词法分析基本思想

3.1.1 词法分析任务

词法分析(lexical analysis)是编译器的第一个阶段,它的主要任务是从左至右逐个字符地对源程序进行扫描,产生一个个单词序列,用于语法分析。完成词法分析任务的程序称为词法分析程序,通常也称为词法分析器或扫描器(scanner)。它一般是一个独立的子程序或作为词法分析器的一个辅助子程序。

词法分析程序的主要任务是按照高级语言的词法规则从左到右逐个扫描字符流的源程序,从中识别出各类有独立意义的单词。单词是具有独立意义的最小语法单位。词法分析功能的具体说明参见图 3-1。词法分析程序的输出一般是将单词变换成带有单词性质且定长的属性字,并填充到符号表中。

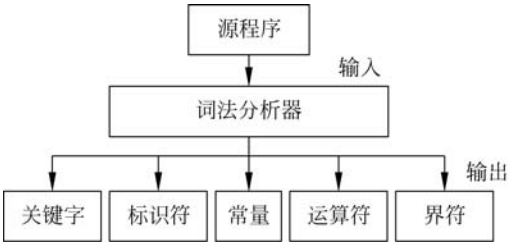

图 3-1 词法分析器功能示意图

【例 3-1】 有 C++代码段: while (i!= j) i++;

经词法分析器处理以后,它将被转换为如表 3-1 所示的单词符号串。

表 3-1 单词符号串

单 词	单词说明	单 词	单词说明
while	关键字	)	界符
(	界符	i	标识符
i	标识符	++	运算符
!=	运算符	;	界符
j	标识符		

除了识别单词外，为方便下一阶段的工作，词法分析程序还应该完成其他一些任务，主要包括以下几个方面。

（1）消除无用字符。

对源程序文本进行处理，过滤掉源程序文本中的注释、空格、换行符及其他一切与语法分析和代码生成均无关的信息。

（2）进行内部编码。

将长度不一、种类不同的单词变成长度统一、格式规整、分类清晰的内部机器码表示。

（3）建立各种表格。

在词法分析时，可以根据单词特点建立不同表格，例如：名字表（标识符表）、常数表、数组向量表、过程表、界限表（包含保留字、运算符等）。

（4）进行词法检查。

为了使编译程序能将发现的错误信息与源程序的出错位置联系起来，词法分析程序负责记录新读入的字符行的行号，以便行号与出错信息相关联；另外，在支持宏处理功能的源语言中，可以由词法分析程序完成其预处理等。

3.1.2　词法分析方式

现阶段词法分析主要有两种分析方式。

1. 将词法分析程序和语法分析程序按顺序执行

在多遍扫描的编译程序中，词法分析可以单独作为一遍扫描来完成，此时可将词法分析程序的输出放在一个中间文件上，语法分析程序可以从该文件取得它的输入，如图 3-2 所示。词法分析从语法分析独立出来的原因主要是考虑这样便于集中进行语法分析，便于建立有效的词法分析技术，给语法分析提供更多更详细的信息。

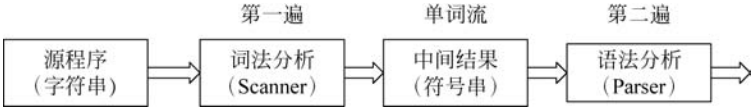

图 3-2　词法分析程序和语法分析程序按顺序执行

2. 将词法分析程序编写成一个独立子程序

在一遍扫描的编译程序中，往往将词法分析编写成语法分析的一个子程序，供语法分析时随时调用，每调用一次，则从源程序字符串中读出一个具有独立意义的单词，如图 3-3 所示。这种模式不需要在内存中构造和保留中间文件，所以可以节省内存空间。

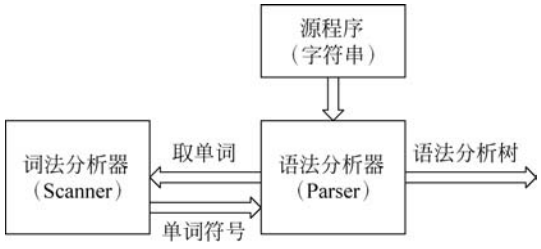

图 3-3　词法分析程序作为语法分析程序的子程序

3.2 单词的描述工具

3.2.1 正规集和正规式

正规式及正规式所表示的语言——正规集,其概念是美国数学家 Kleen 在 20 世纪 50 年代提出来的。这种方法现在已成为处理有限自动机问题的主要数学工具,无论在理论上还是在计算机科学领域的诸多工程实践中都有重要应用。

定义 3-1 正规式与正规集

设 Σ 为有限字母表,在 Σ 上的正规式与正规集可递归定义如下:

(1) ε 和$\varnothing$是 Σ 上的正规式,它们表示的正规集分别为$\{\varepsilon\}$和$\varnothing$。

(2) 对任何 $a\in\Sigma$,a 是 Σ 上的正规式,它表示的正规集为$\{a\}$。

(3) 若 r、s 都是正规式,它们表示的正规集分别为 R 和 S,则$(r|s)$(或表示为 $r+s$)、$(r\cdot s)$(或表示为 rs)、$(r)^*$ 也是正规式,它们表示的正规集分别是 $R\cup S$、RS、R^*。

(4) 有限次使用上述 3 条规则构成的表达式称为 Σ 上的正规式,仅由这些正规式表示的集合为正规集。

规定正规式运算的优先级由高到低的次序为 *(闭包)、·(连接)和|(并),它们的结合性都为左结合。在此规定下,书写正规式时可以省去不致造成混淆的括号。例如$((0\cdot(1^*))|0)$可写成 $01^*|0$。

【例 3-2】 设字母表 $\Sigma=\{0,1\}$,则 0、1、ε、$\varnothing$是 Σ 上的正规式,$0|1$、$0\cdot1$、$1\cdot0$、0^*、1^*是 Σ 上的正规式。

【例 3-3】 设字母表 $\Sigma=\{A,B,0,1\}$,试说明正规式$(A|B)(A|B|0|1)^*$和$(0|1)(0|1)^*$分别表示的符号串集合。

正规式$(A|B)(A|B|0|1)^*$表示的正规集是以字母 A 或 B 开头后跟任意多个字母 A、B 和数字 0、1 的符号串(标识符)的全体。

正规式$(0|1)(0|1)^*$表示的正规集是二进制数字串。

正规式 r 所表示的正规集 R 是字母表 Σ 上的语言,称为正规语言,用 $L(r)$表示,即 $R=L(r)$。$L(r)$中的元素为字符串(也称为句子)。若两个正规式 r 和 s 所表示的语言 $L(r)=L(s)$,则称 r、s 等价,记作 $r=s$。例如,$1(01)^*=(10)^*1$。正规式的代数性质参见表 3-2,其中 s、t、r 为正规式。

表 3-2 正规式的代数性质

公 理	描 述	公 理	描 述
$s\|t=t\|s$	并是可交换的	$\varepsilon s=s$ $s\varepsilon=s$	ε 是连接的恒等元素
$s\|(t\|r)=(s\|t)\|r$	并是可结合的	$s^*=(s\|\varepsilon)^*$	闭包和 ε 间的关系
$(st)r=s(tr)$	连接是可结合的	$a^{**}=a^*$	闭包是幂等的
$s(t\|r)=st\|sr$	连接对并可分配		
$(t\|r)s=ts\|rs$			

利用正规式的代数性质，可以对正规式进行等价变换及化简。

有些语言不能用正规式表达，说明了正规式的描述能力有限。例如，正规式不能描述配对或嵌套的结构。

3.2.2 正规式与有限自动机的等价性

从前面的讨论得知，有限自动机接收的语言等价于正规文法产生的正规语言。而正规式或正规集定义的语言也是正规语言。那么，正规式与有限自动机在描述语言上应该等价。

定理 3-1

(1) 字母表 Σ 的确定的有限自动机 M 所接受的语言 $L(M)$ 是 Σ 上的一个正规集。

(2) 对于 Σ 上的每一个正规式 r，存在一个字母表是 Σ 的非确定有限自动机 M，使得 $L(M)=L(r)$。

定理 3.1 表明，正规式所表示的语言即正规集与有限自动机所识别的语言是完全等价的，只是表示形式不同。也就是说，从描述语言的角度，没有必要对非确定的有限自动机、确定的有限自动机及它们所识别的语言(正规集)加以区分。同一种语言，既可以用有限自动机描述，也可以用正规式描述。有兴趣的读者可以对定理 3.1 进行证明。

根据上面的等价性定理，构造出等价的转换算法。

对于字母表 Σ 上任意一个正规式 r，一定可以构造一个非确定的有限自动机 M，使得 $L(M)=L(r)$。首先构造非确定的有限自动机 M 的一个广义的状态图，也是该非确定的有限自动机 M 的初始状态图。其中，只有一个开始状态 S 和一个终止状态 Z，连接 S 和 Z 的有向弧上的标记是正规式 r。然后，按照图 3-4 所示的替换规则对正规式 r 依次进行分解，分解的过程是一个不断加入结点和弧的过程，直到转换图上的所有弧标记上都是 Σ 上的元素或 ε 为止。

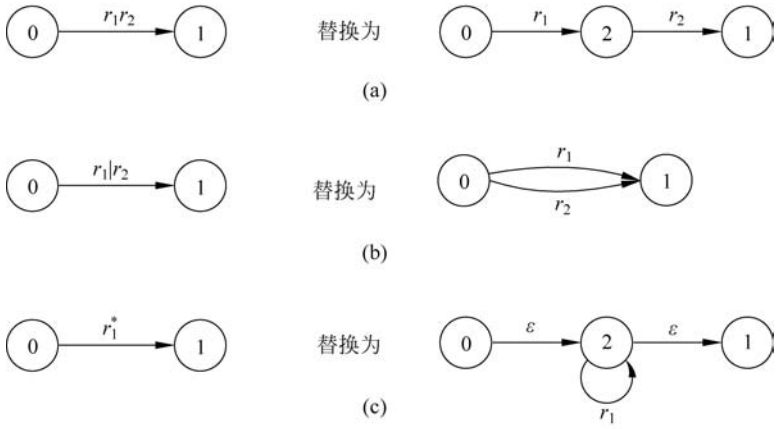

图 3-4 从正规式到有限自动机的等价替换规则

【例 3-4】 设 $\Sigma=\{x,y\}$，Σ 上的正规式 $r=xy^*(xy|yx)x^*$，构造一个非确定的有限自动机 M，使 $L(M)=L(r)$。

第一步，构造 M 的初始状态图，得到如图 3-5(a)所示的状态图。

第二步，将 $r=xy^*(xy|yx)x^*$ 拆成 4 个正规式 x、y^*、$xy|yx$、x^* 的连接，得到如图 3-5(b)

所示的状态图。

第三步,将 y^*、$xy|yx$、x^* 分别拆成正规式 y 的闭包、xy 与 yx 的并、x 的闭包,得到如图 3-5(c)所示的状态图。

第四步,将 xy、yx 分别拆成正规式 x 与 y 的连接和 y 与 x 的连接,得到如图 3-5(d)所示的状态图。

所有弧上的标记都属于 $\Sigma \cup \{\varepsilon\}$,构造完毕。

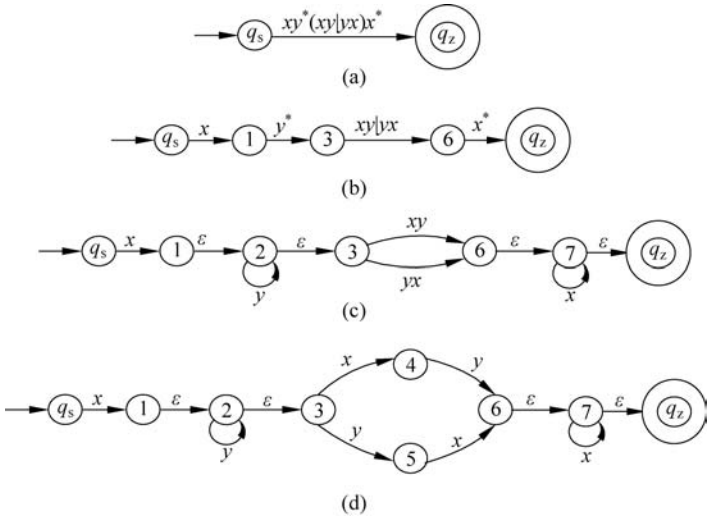

图 3-5 正规式 $r=xy^*(xy|yx)x^*$ 的分解

上面讨论了由正规式到非确定的有限自动机的转换,同样,对于一个字母表 Σ 上的非确定的有限自动机 M,也可以在 Σ 上构造相应的正规式 r,使 $L(r)=L(M)$。

根据前面的定理,构造之前,首先对非确定的有限自动机 M 对应的有限自动机进行拓广,加进两个状态,一个为唯一初态 S,一个为唯一终态 Z。具体用状态图描述的构造步骤如下。

(1) 在非确定的有限自动机 M 的状态转换图中加进两个结点,一个为 S 结点,另一个为 Z 结点。其中 S 是唯一的开始状态,Z 是唯一的终止状态。

(2) 从 S 用 ε 弧连接到 M 的初态结点,从 M 的所有终态结点用 ε 弧连接到 Z 结点,形成一个与 M 等价的 M'。M'有一个没有射入弧的初态结点 S 和一个没有射出弧的终态结点 Z。

(3) 对新的非确定的有限自动机按照图 3-6 所示的替换规则反复进行替换,这个过程实际上是正规式的合成过程,即对 M'不断消去结点和弧的过程。直到状态图中只剩下状态结点 S 和 Z 为止。当状态图中只有状态 S 和 Z 时,在 S 到 Z 的弧上标记的正规式即所求结果。

【例 3-5】 设非确定的有限自动机 M 的状态转换图如图 3-7 所示。在 $\{x,y\}$ 上构造一个正规式 r,使 $L(M)=L(r)$。

第一步,拓广状态图,得到如图 3-8(a)所示的状态图。

第二步,消去状态 2 和 3,得到如图 3-8(b)所示的状态图。

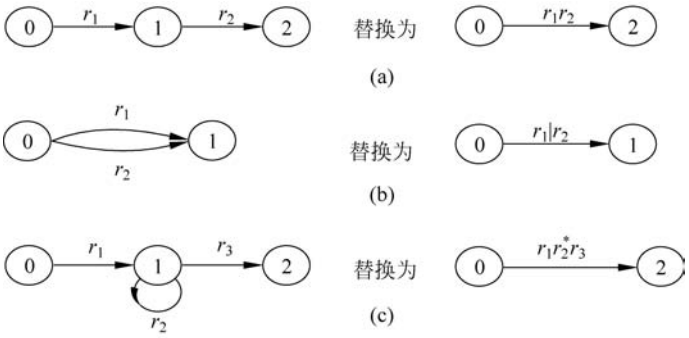

图 3-6 从有限自动机到正规式的等价替换规则

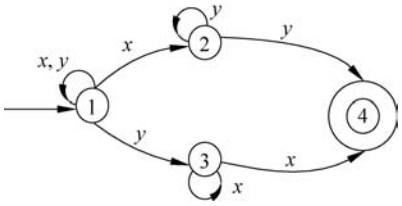

图 3-7 NFA M 的状态转换图

第三步，消去状态 4，得到如图 3-8(c)所示的状态图。

第四步，消去状态 1，得到如图 3-8(d)所示的状态图。

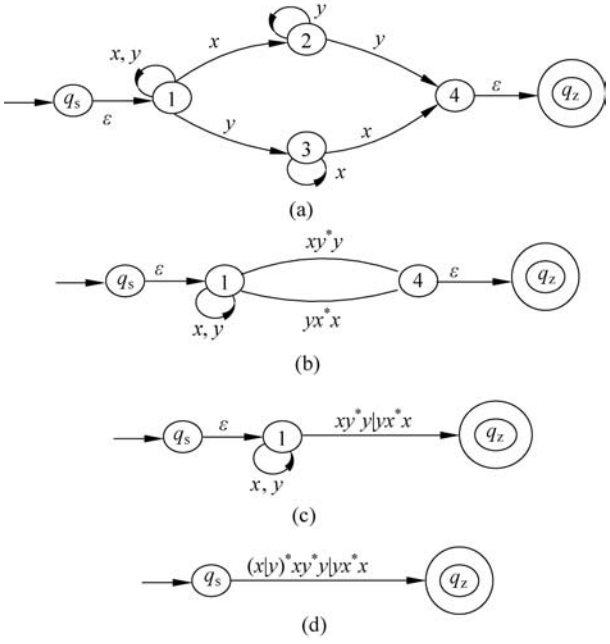

图 3-8 NFA M 到正规式的转换

此时状态转换图中只有状态 S 和 Z，所以非确定的有限自动机 M 对应的正规式为 $r=(x|y)^*xy^*y|yx^*x$。

【例 3-6】 设某种语言由标识符和无符号正整数两类单词构成,设 L 表示字母,D 表示十进制数字,其中 $L=[a\sim z, A\sim Z]$, $D=[0\sim 9]$。

设标识符和无符号正整数的词法规则如下:

<标识符>→$(L|\_)(L|D|\_)^*$

<无符号正整数>→DD^*

识别标识符和无符号正整数的状态转换图如图 3-9 所示。

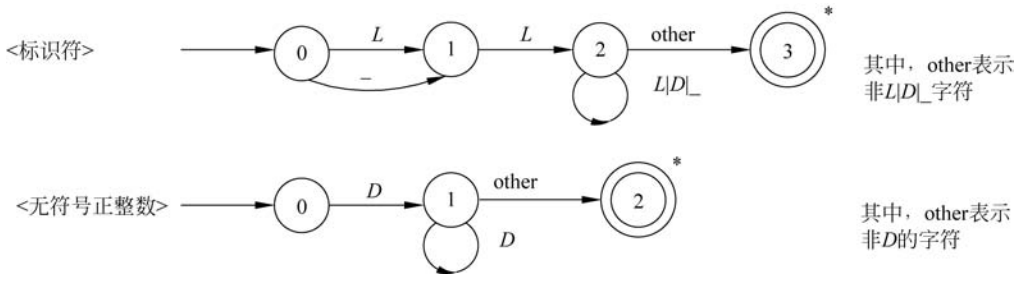

图 3-9 识别标识符和无符号正整数的状态转换图

注意:状态转换图中的终态结点若带有"*",表示在扫描和识别单词过程中,到达一个单词识别状态时,对于当前识别的单词多读进了一个字符,即超前搜索,则源程序扫描指针需要做自减运算以回退一个字符。

根据图 3-9 所示的状态转换图,同时可以得到:

<标识符>对应的正则式:$(L|\_)(L|D|\_)^*$

<无符号正整数>对应的正则式:DD^*

3.3 单词的识别

3.3.1 单词分类

逐个读入源程序字符并按照构词规则切分成一系列单词。单词是语言中具有独立意义的最小单位,包括保留字、标识符、运算符、标点符号和常量等。例如,在 C 语言中,一般的表达式不一定是单词,因为它们虽然具有独立的意义,但不一定是最小的语法单位。如表达式 a+b,它的单词是 a、+、b。C 语言中的字母和数组也不一定是单词,因为它们虽然是最小的语法单位,但常常不具有独立意义,如程序中定义一个变量 a10,a 和 10 若单独作为一个变量名和一个常数存在于程序中,则是合法的单词,但作为对变量标识的定义,只有 a10 作为一个整体才是合法的单词。

到底哪些语法符号是语言中具有独立意义的最小语法单位呢? 这与具体语言的词法规则有关。一般常用的程序设计语言的单词可分为如下几类。

(1) 保留字,也称关键字。

保留字一般是语言系统本身定义的,通常是由字母组成的字符串。例如,C 语言中的 int 、float、while、for、break 和 switch 等。

(2) 常数。

程序设计语言中包含各种类型的常数,如整型常数、实型常数、不同进制的常数、布尔常数、字符及字符串常数等。

(3) 标识符。

标识符用来表示各类名字的标识,如常量名、变量名、数组名、结构名、函数名、类名、对象名和文件名等。

(4) 运算符。

运算符表示程序中算术运算、逻辑运算、字符及位串等运算的字符(或串),如各类语言中较通用的+、-、*、/、<=、>=、<和>等。还有一些语言包含特有的运算符,如C语言中的++、?:、||、+=等。

(5) 界符。

如逗号、分号、括号、单引号和双引号等均为界符。

词法分析程序所输出的单词符号常常采用以下二元式表示:(单词种类,单词值)。其中,单词种类表示单词的类别,用来刻画和区分单词的特性或特征,通常用整数编码来表示。例如,可以将C语言中的关键字设计为一类,标识符为一类。运算符既可以按优先级分为不同的类,也可以作为一类。单词值则是编译其他阶段需要的信息,可以省略。例如,C语言的语句"int i = 10, j = 1;"中的单词i和j的种类都是整型,单词的值为10和1对于代码生成来说是必不可少的。有时,对某些单词来说,不仅需要它的值,还需要其他一些信息以便于编译。例如,对于标识符来说,还需要记载它的类别、层次和其他属性,如果这些属性统统收集在符号表中,那么可以将单词的二元式表示设计成这种形式:(标识符,指向该标识符所在符号表中位置的指针)。

对于一种语言而言,如何对其中的单词进行分类、分成几类、怎样编码、单词属性部分能包含多少信息等,并没有一个原则性的规定,要视具体情况而定,主要取决于处理上的方便。一般来说,一个单词为一类,处理较为方便,但对于标识符却是不可行的。可以将具有一定共性的单词视为一类,统一给出一个属性,但属性部分信息包含越多,实现起来越复杂。属性字中的单词值部分要直接或间接给出单词在计算机内存储的内码表示。如对于某个标识符或某个常数,常把指向存放它的有关信息的符号或常数表入口的指针作为它的单词值。

3.3.2 单词的内部表示

单词的内部表示Token的结构一般由两部分组成:单词类别和语义信息。单词类别用来区分单词的不同种类,通常可以用整数编码来表示。单词的语义信息也取决于今后处理上的方便。对于常量和标识符还可以单独构造常量表和标识符名字表,此时,单词的语义信息的值就是指向常量表或标识符名字表中相应位置的指针。

3.3.3 单词的形式化描述

描述程序设计语言中单词的工具主要有以下3种:正则表达式、自动机和正则文法。它们的功能彼此相当。对于一个一般的程序设计语言,各类单词的正则表达式可能如下。

(1) 标识符:$L(L|D)^*$,其中L=[a～z, A～Z], D=[0～9]。

(2) 整数：$D|D^*$，其中 $D=[0\sim9]$。

(3) 特殊符号：+|;|:|:=|<|<=|…

(4) 保留字：if|else|while|…

构造识别单词的有限自动机的方法与步骤如下。

(1) 根据构成规则对高级程序语言的单词按类构造出相应的状态转换图。

(2) 合并各类单词的状态转换图，构成一个能识别该语言的所有单词的状态转换图。合并方法如下。

① 将各类单词的状态转换图的初始状态合并为一个唯一的初始状态。

② 化简调整状态冲突和对冲突状态重新编号。

③ 如果有必要，增加出错状态。

【例 3-7】 如图 3-9 示出的例 3-6 两类单词的状态转换图，经合并和调整后，得到识别标识符和无符号正整数的一个状态转换图，如图 3-10 所示。

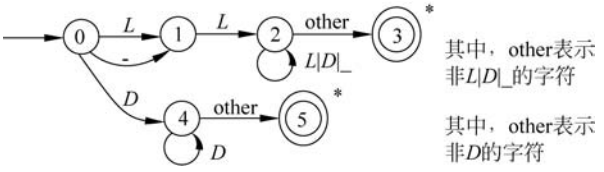

图 3-10 对图 3-9 合并后的状态转换图

3.4 词法分析程序的设计及实现

词法分析程序的设计步骤如下。

(1) 确定词法分析器的接口，即确定词法分析器是作为语法分析的一个子程序还是作为独立的一遍扫描过程。

(2) 确定单词分类和单词结构。

(3) 构造每一类单词的正规式以及对应的 NFA。

(4) 合并各类单词的状态图，增加一个出错处理终态，构成一个识别该语言所有单词的状态转换图 NFA，对该 NFA 确定化及化简，得到最终的能识别该语言所有单词的 DFA。

(5) 编程实现第(4)步得到的 DFA。

3.4.1 词法分析程序的预处理

词法分析器工作的第一步是源程序以字符流形式输入。词法分析器通常由两部分构成：预处理程序和扫描器(单词识别程序)，如图 3-11 所示。

由于在源程序中，特别是非自由格式书写的源程序中，往往有大量的空白符、回车换行符及注释等，这是为增加程序可读性及程序编辑的方便而设置的，对程序本身无实际意义。另外，像 C 语言有宏定义、文件包含、条件编译等语言特性，为了减轻词法分析器实质性处理的负担，在源程序从输入缓冲区进入词法分析器之前，要先对源程序进行预处理，预处理子程序一般完成的主要功能如下：滤掉源程序中的注释；剔除源程序中的无用字符；进行

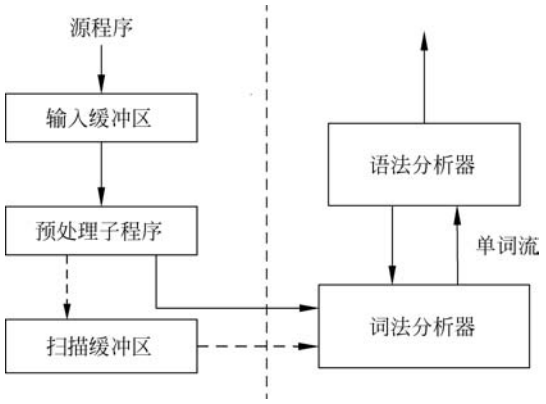

图 3-11 词法分析器结构示意图

宏替换；实现文件包含的嵌入和条件编译的嵌入等。

扫描器真正接受的输入是经过预处理后的源程序串，输出是识别出的单词流。

根据内存空间情况有两种选择。

(1) 如果内存比较大，可以直接输入到内存的一个源程序区，然后词法分析程序从源程序区依次读入字符进行扫描和处理。这种方式可以大大节省源程序输入的时间，提高词法分析器的效率。但在一个有限的内存空间内要满足各种规模的源程序的一次输入是困难的，这样的系统开销也比较大。

(2) 如果内存不足，将源程序以文件的形式存储在外部介质上，如磁盘、U 盘等。可以先在内存中开辟一个大小适宜的缓冲区，这个缓冲区称为输入缓冲区。词法分析程序工作时，先从外部介质将输入符号串分批读入缓冲区，然后进行预处理子程序，将无用的字符剔除。如果内存空间有限，可以将预处理后的字符流送入扫描缓冲区，然后送入扫描器进行单词符号的识别，如图 3-11 所示。

源程序以文件形式存于外存，首先要将其读入内存才可进行词法分析。早期计算机内存较小，只能在内存设置长度有限的输入缓冲区，分段读入源程序进行处理。在编制程序时，必须考虑由于源程序分段读入所产生的问题。例如，由多个字符构成的单词有可能被缓冲区边界所打断。目前计算机所使用的内存已超过若干年前硬盘容量，计算机内存足以容纳源程序的全部，故源程序可一次全部读入内存进行处理。

扫描缓冲区就是在内存中开辟一部分单元，供识别单词用。注意：扫描缓冲区和输入缓冲区是不同的，输入缓冲区是从外存上读入部分字符，而扫描缓冲区仅供识别单词用。显然，无论缓冲区设定为多大，都不能保证单词不会被它的边界打断，例如若有代码段 i++、j++，由于空间限制，可能在缓冲区中只存储了 i++、j，如图 3-12 所示。在这种情况下，即使读到缓冲区的最后一个字符，也仍不能找到该单词的右边界，这时，若从外存上再读一部分源程序进入缓冲区，则会将没有处理过的字符(++)冲掉。为此，可将缓冲区分成相等的两个区域，其结构如图 3-13 所示。

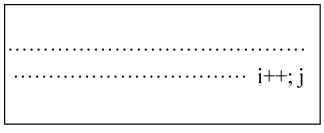

图 3-12 扫描缓冲区示意图

在扫描缓冲区中一般设两个指针，一个指向当前正在识别的单词的开始位置，另一个用

于向前搜索寻找单词的终点。不论扫描缓冲区定为多大,都不能保证单词符号不会被它的边界所打断,因此,扫描缓冲区最好使用如图 3-13 所示的一分为二的区域。

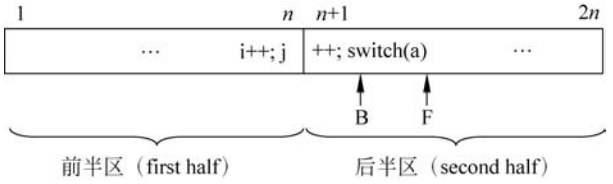

图 3-13 互补的扫描缓冲区结构示意图

开始时两个扫描缓冲区均为空。首先填满扫描缓冲区的前半区,并将两个指针指向扫描缓冲区一的第一个字符。将搜索指针向后移动,当识别出一个单词之后,搜索指针已指向下一个单词的第一个字符,然后再将起始指示器移到搜索指示器所指字符,接着搜索指针又开始扫描再下一个单词。当搜索指针越界时,说明扫描缓冲区前半区中的字符不足一个单词,这时填满扫描缓冲区后半区,再将搜索指示器扫描缓冲区后半区第一个字符。这样,两个扫描缓冲区交替工作。一般,扫描缓冲区的长度可以存放最长的一个单词即可正常工作,否则就不能保证单词符号不会被扫描缓冲区边界所打断。

扫描缓冲区两个半区互补功能的算法描述如下(设 F 为搜索指针):

```
if F at end of first half
{
    reload second half;
    F++;
}
else
    if F at end of second half
{
    reload first half;
    move F to beginning of first half
}
    else F++;
```

3.4.2 由词法规则画出状态转换图

通过综合实例画出识别 C 语言子集的单词符号的状态转换图。

设 C 语言的一个子集由下列单词符号构成,以正规式的形式表示如下。

(1) 关键字: int,if,for,while。

(2) 标识符: 字母(字母|数字)* 。

(3) 无符号整常数: 数字(数字)* 。

(4) 运算符或分界符: = , * , + ,++, += ,{,},;。

根据正规式和有限自动机的等价转换规则,可以得到如图 3-14 所示的各类单词状态转换图。将各类单词对应的状态转换图合并,构成一个能识别语言所有单词的状态转换图。然后化简并调整冲突和状态编号。

经合并和调整后,确定并最简化得到最终的单词的状态转换图,如图 3-15 所示。注意,

针对保留字，可以将保留字存储为一个特定文件，当识别出标识符后进行查表，判断是否为保留字。

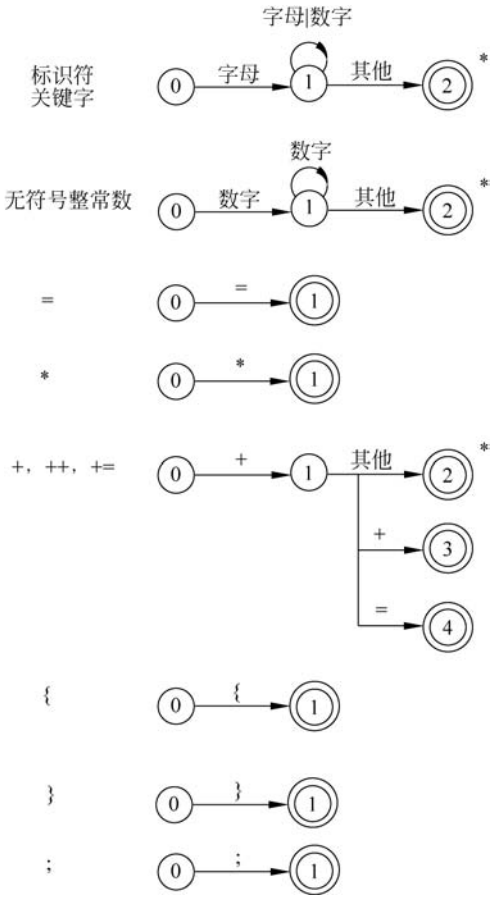

图 3-14 C 语言子集各类单词状态转换图

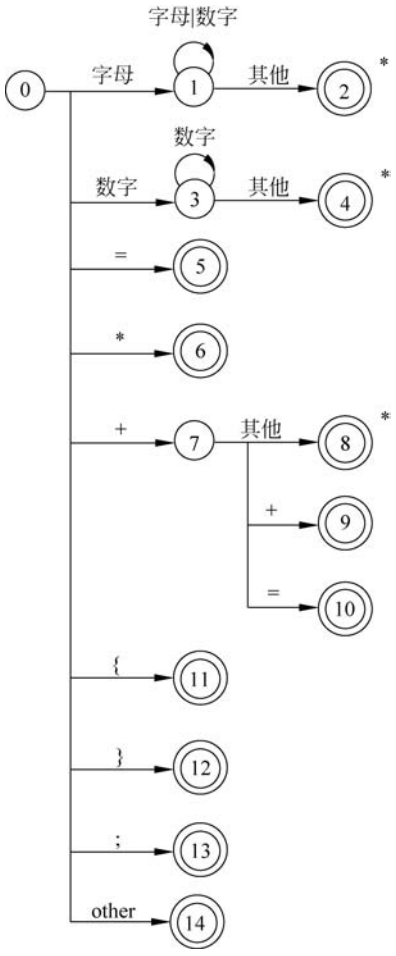

图 3-15 C 语言子集单词状态转换图

3.4.3 单词对应状态转换图的实现

根据语言的词法规则构造出识别其单词的状态转换图，仅仅是理论上的词法分析器，是一个数学模型。那么，如何将状态转换图变为一个可行的词法分析器呢？最常用的状态转换图的实现方法称为程序中心法，即把状态转换图看成一个流程图，从状态转换图的初态开始，对它的每个状态结点编一段相应的程序。它要做的工作如下。

(1) 从输入缓冲区中取一个字符。为此，我们使用函数 nextchar()，每次调用它，搜索指针向前移一位，返回一个字符送给 CHAR 变量。

(2) 确定在本状态下，哪一条弧是用刚刚送来的输入字符标识的。如果找到，控制就转到该弧所指向的状态；若找不到，那么寻找该单词的企图就失败了，转向(3)。

(3) 错误处理：先行指针必须重新回到开始指针处，并用另一状态图来搜索另一个单

词。如果所有的状态转换图都试过之后,还没有匹配的,就表明这是一个词法错误,此时,调用错误处理程序。

以图 3-16 为例,可以得到以下伪代码段:

```
state i:
        CHAR = GETNEXTCHAR();
        CASE CHAR OF
        'A'…'Z': … state j … ;
        '0'…'9': … state k … ;
        '; ' : … state l … ;
        END;
        ERROR();
```

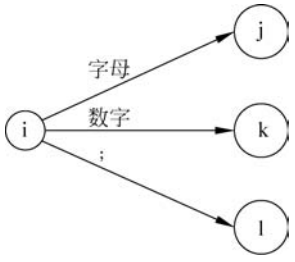

图 3-16 状态图示例

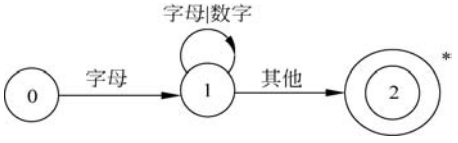

图 3-17 标识符状态图

【例 3-8】 以标识符状态图为例,将图 3-17 所示转换为伪代码段。设标识符类别为 1,输出单词形式为<1,单词名称>。

对于图 3-17 的状态转换图,状态 0 和状态 1 的代码如下所示:

```
state 0:
    C = GETNEXTCHAR();
    if  LETTER(C) then goto state 1
    else ERROR( )
state 1:
    C = GETNEXTCHAR();
    WHILE  (ISLETTER(C) OR ISDIGIT(C)) DO
    {
       当前字符放入一临时字符数组;
    C = GETNEXTCHAR();                  //从缓冲区取下一字符
    };
    UNGETCH;                            //回退一字符
    OUTPUT(1,标识符名字);
```

【例 3-9】 关键字的伪代码处理流程。

关键字通常由字母构成,符合标识符规则。考虑简化程序设计,将关键字作为一种特殊标识符来处理。设置一个关键字表,如表 3-3 所示,当状态转换图识别出一个标识符时,就去查对这张表,确定它是关键字还是标识符。

表 3-3 关键字表示例

关 键 字	类 别	关 键 字	类 别
int	0	for	2
if	1	while	3

```
char reserve(char token[])              /* 查找表函数 */
{//基本字及编码表
       const char * table[] = {"int","if","for","while"};
       for(int i = 0;i < strlen(code);i++)
           if(strcmp(token,table[i][0]) == 0) return table[i][1];
       return 'id';                     //标识符的单词种别为'id'
}
```

综上所述,给出图 3-15 所示的状态转换图实现的 C 语言伪代码如下:

```
int state = 0;
enum letter('a'…'z');
enum digit('0'…'9');
char char1;

char1 = nextchar();
switch(state)
{
    case 0:switch(char1)
           {
               case 'a'…'z':   state = 1;break;
               case '0'…'9':   state = 3;break;
               case ' = ':     state = 5;break;
               case ' * ':     state = 6;break;
               case ' + ':     state = 7;break;
               case '(':       state = 10;break;
               case ')':       state = 11;break;
               case ';':       state = 12;break;
               default :       state = 13;
           }
           break;
    case 1: while(char1 == letter||number) char1 = nextchar();
            state = 2;
            break;
    case 2:untread();                          /* 函数 untread()回退一个已读进的字符,属性
                                                  01 表示关键字,属性 02 表示标识符 */
            return(02,value)or return(01,value);
            break;
    case 3:while(char1 == number) char1 = nextchar();
            state = 4;
            break;
    case 4:untread();return(03,value);break;  /* 属性 03 表示无符号整常数 */

  case 5:return(04, );break;                   /*  属性 04 表示 =  */
```

```
  case 6:return(05, );break;                    /* 属性 05 表示 * */
  case 7:if(char1 == ' + ')state = 9;
         else if(char1!= " + ")state = 8;
         break;
  case 8:untread();return(08, );break;          /* 属性 08 表示 + */
  case 9:return(09, );break;                    /* 属性 09 表示++ */
  case 10:return(10, );break;                   /* 属性 10 表示 += */
  case 11:return(11, );break;                   /* 属性 11 表示( */
  case 12:return(12, );break;                   /* 属性 12 表示) */
  case 13: return(13, );break;                  /* 属性 13 表示; */
  case 14:error();                              /* error 是语法错误处理函数 */
}
```

状态转换图实现的另一种方法是数据中心法,即将状态转换图看成一种数据结构(如状态矩阵表),用控制程序控制输入字符在其上运行,从而完成词法分析。而一个实际的状态矩阵表往往是一个稀疏矩阵,这会增加存储空间的开销。可以采用压缩的二级目录表的数据结构。所谓二级目录表,即分为主表和分表。主表结构为状态和分表地址两个数据项,若状态为终态(即单词接收态),则分表地址是处理相应单词的子函数入口。分表为当前输入字符及转换状态两个数据项。以图 3-15 为例,给出主表与分表的关系和结构,其表示如图 3-18 所示。

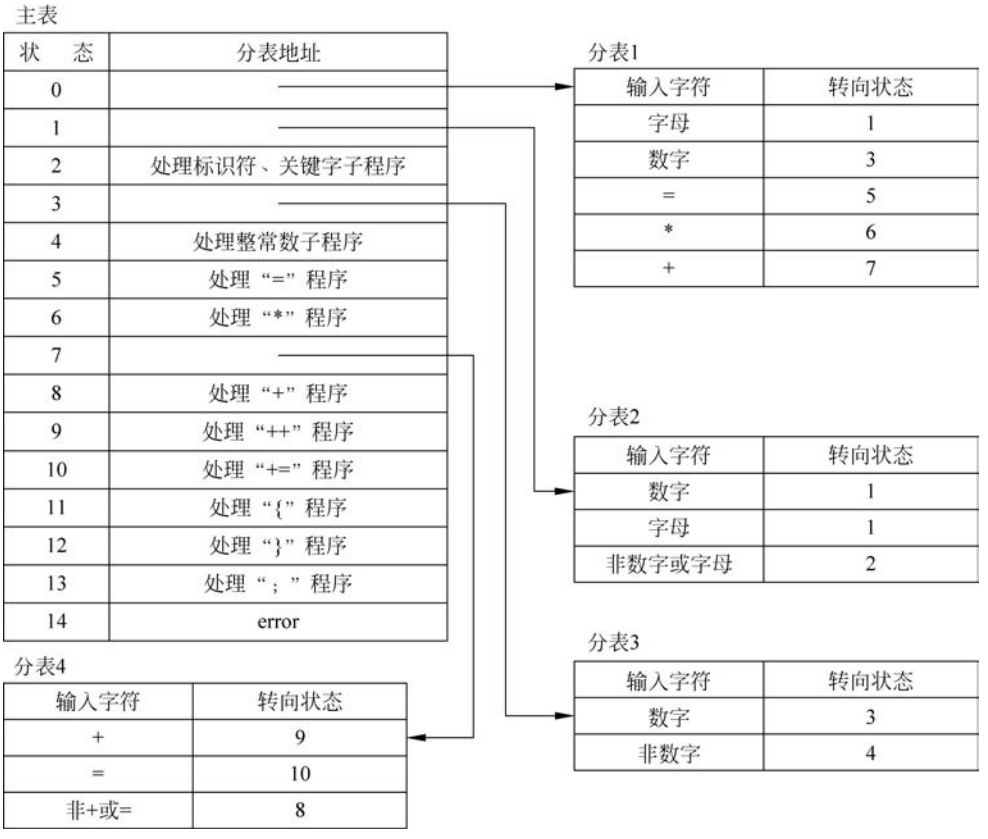

图 3-18 状态转换图的二级目录表

以二级目录表为主的数据中心实现法的控制程序请读者自己给出。

3.4.4 词法分析中的错误处理

词法错误是编译程序在词法分析阶段出现的错误，词法错误指单词的拼写错误，例如，保留字的拼写错误。词法分析程序主要功能是读词，分解单词符号拼写单词，该阶段的错误大多数是单词拼写错误，或者是书写错误。源程序所有的拼写错误一般有如下几种情况。

(1) 错写了一个字母或字符，如将 CONST 写成 COMST。

(2) 少写了一个字符，如将 WHILE 写成 WHIE。

(3) 多写了一个字符，如将 ELSE 写成 ELSET。

(4) 相邻的字符颠倒了位置，如 THEN 写成 THNE。

对词法分析产生的错误进行错误处理要涉及下面几个问题。

(1) 判断是哪个标识符拼写错误。

(2) 错误的性质，产生错误的原因，是哪类拼写错误。

(3) 错误的位置，产生错误的位置。

(4) 错误的局部化。

(5) 重复错误信息的遏止。

确定出错，采用错误恢复策略。最简单的错误恢复策略：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字符为止。这也被称为“恐慌模式”恢复。

3.5 词法分析程序的自动实现

本节介绍一个著名的词法分析器自动生成工具 Lex。它是以有限自动机理论为基础而设计的，本质上是一个可以识别所有模式的确定的有限自动机。

3.5.1 Lex 介绍

Lex 是一个词法分析程序的自动生成器(lexical analyzer generator)。1972 年贝尔实验室在 UNIX 上首次实现了 Lex。1984 年，GNU 工程推出 Flex(fast lexical analyzer generator)，它是对 Lex 的扩充。Lex/Flex 支持使用正则表达式来描述各个单词的模式，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，而工具本身称为 Lex 编译器。图 3-19 示出了 Lex 编译器的主要组成部分。从图中可以看出，Lex 编译器接收用正规式表示的单词规则，然后利用算法 X 从此正规式出发构造能识别正规式描述的单词集(正规集)的非确定的有限自动机 M'，再用算法 Y(即子集法)将 M'确定化，得到与之等价的确定的有限自动机 M''，最后还可用算法 Z(即划分算法)对 M''进行化简，得到确定的有限自动机 M，则这个有限自动机 M 即是理论上的扫描器。

使用 Lex 自动生成词法分析器的步骤一般分为 3 步，如图 3-20 所示。

(1) 用 Lex 语言编写一个输入文件，描述将要生成的词法分析器。在图 3-20 中这个输

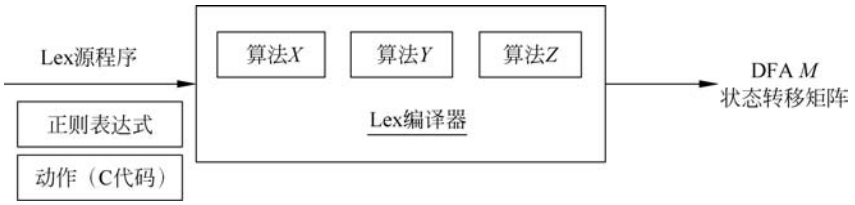

图 3-19　词法分析器自动构造的思想

入文件称为 lex.l。

(2) Lex 编译器将 lex.l 转换成 C 语言程序,生成的文件称为 lex.yy.c。

(3) lex.yy.c 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单词流的可运行的词法分析器。

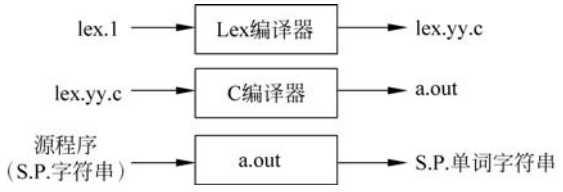

图 3-20　用 Lex 自动生成一个词法分析器流程

3.5.2　Lex 语法基础

Lex 语言是对表示语言单词集的正规式的描述,以解决正规式规则输入问题。一个 Lex 程序具有如下形式:

```
说明部分                    /* 包含模式宏定义和 C 语言的说明信息 */
%%
规则部分                    /* 转换规则 */
%%
辅助函数                    /* 规则动作部分所需的辅助过程的 C 语言代码 */
```

(1) 第一部分“说明部分”包括 C 语言代码、模式宏定义等。模式宏定义实际是对识别规则中出现的正规式的辅助定义。如语言中的字母可定义为 Letter[A|B|…|Z|a|b|…|z]; 数字可定义为 digital[0|1|2|…|9]; 除宏定义外,定义部分的其余代码需要用符号%{和}%括起来。另外,Lex 源程序所使用的 C 语言库文件和外部变量也应分别用 #include 及 extern 予以说明,并置于%{和}%之内。Lex 扫描源文件时将%{和}%之间的部分原封不动地复制到输出文件 lex.yy.c 中。

例如, C 语言说明和 Lex 宏定义示例如下:

```
%{
#include <stdio.h>
#include <ctype.h>
/* definitions of manifest constants LT,LE,EQ,GT,GE,IF,THEN,ELSE,ID */
}%
/* regular expression */
```

```
delim    [\t\n]
ws       {delim}+
letter   [A-Za-z]
digit    [0-9]
id       {letter}({letter}|{digit})*
%%
```

在 Lex 源程序中,起标识作用的符号%%、%{和}%都必须处于所在行的最左字符位置。另外,在其中也可以随意添加 C 语言形式的注释。

(2) 第二部分“规则部分”是 Lex 程序的主体部分。其一般形式如下:

```
模式 1 {动作 1}
模式 2 {动作 2}
⋮
模式 n{动作 n}
```

其中模式是对单词的描述,用正规式表示。动作是与匹配的模式对应的,用 C 语言代码表示对模式处理的动作,表示当识别出某个模式所表示的单词后,词法分析器需要做的处理工作,即应执行动作的程序。通常使用的 Lex 模式定义如表 3-4 所示。

表 3-4 Lex 模式定义

模 式	说 明	示 例
x	匹配单个字符 x	
[abc]	匹配 a 或 b 或 c	[a-h0-5]
[^abcde]	匹配 a～e 之外的任意字符(可为[^a-e])	[^abA-Z\n]表示匹配除小写字母 a,b、大写字母 A～Z 和换行符外任意字符
\	转义符定义同 ANSI C	
.	匹配除去换行符之外的任意字符	
r*	r 是正规式,r* 匹配 0 个或多个 r	
r+	r 是正规式,r+匹配 1 个或多个 r	
r?	r 是正规式,r? 匹配 0 个或 1 个 r	
r{2,5}	r 同上,匹配 2～5 次 r	
r{2,}	r 同上,匹配 2 次或更多次 r	
r{2}	r 同上,匹配 2 次 r	
{name}	name 是在定义部分出现的模式宏名	
"text"	匹配字符串“text”	
r|s	匹配正规式 r 或 s	
rs	匹配正规式 r 和 s 的连接	
…		

(3) 第三部分“辅助函数”定义对模式进行处理的 C 语言函数、主函数等,它是支持规则的动作部分所需要的处理过程,是对规则部分中动作的补充。这些过程若不是 C 语言的库函数,需给出具体定义,然后分别编译且与生成的词法分析器装配在一起。

需要说明的是,说明部分和辅助函数部分是任选的,规则部分是必需的。

3.5.3 词法分析器自动构造

词法分析程序主要由两部分组成：一张转换矩阵表和一个控制程序。转换矩阵表由Lex对某语言单词集描述的Lex源程序变换得来。基于Lex源程序，Lex编译器的实现步骤可以描述如下。

(1) 对Lex源程序识别规则中的每个 P_i 构造一个相应的非确定的有限自动机 M_i。

引入唯一初态 X，从初态 X 通过ε弧将所有非确定的有限自动机 $M_i(i=1,2,\cdots,n)$ 连接成新的非确定的有限自动机 M'。

(2) 对非确定的有限自动机 M' 确定化，产生确定的有限自动机 M。

(3) 化简确定的有限自动机 M。

(4) 给出总控程序。总控程序的工作原理和手工构造相类似，差异在于如何实现状态迁移。手工构造的扫描器是利用程序控制流程的改变来实现状态迁移，而使用DFA的控制程序是利用状态转换矩阵来实现状态迁移。使用DFA的控制程序远比手工构造的扫描器简单，并且控制程序与源语言的单词集无关。

所谓自动构造词法分析器，实际上就是构造DFA。所以，自动构造词法分析器的难点在于构造DFA，对于实际程序设计语言来说，用人工构造DFA是不可能的，必须由程序来实现。

【例3-10】 编制Lex源程序，分别统计文本文件a.txt中出现的标识符和整数个数，并显示之。标识符定义为字母开头，后跟若干字母、数字或下画线。整数可以带＋或－号，也可不带，且不以0开头。非单词和非整数则忽略不计，将之滤掉不显示。

设Lex源文件名为count.l，文件内容如下：

```
%{
#include "stdio.h"
#include "stdlib.h"
int num_num = 0,num_id = 0;
%}
INTEGER  [-+]?[1-9][0-9]*
ID       [a-zA-Z][a-zA-Z_0-9]*
SPACE    [ \n\t]
%%
{INTEGER} { num_num++;
printf("(num = %d)",atoi(yytext));       //打印数字值
}
{ID} {
num_id++;
printf("(id = %s)",yytext);
}
{SPACE} |
. {
//什么也不做,滤掉白字符和其他字符
}
%%
void main()
```

```
{
yylex();
printf("num = %d,id = %d",num_num,num_id);
}
int yywrap()                                    //此函数必须由用户提供
{return 1;}
```

【例 3-11】 下面是识别某语言单词的 Lex 源文件。

该语言的 Lex 源程序如下：

```
%{
#include<stdio.h>
#include<ctype.h>
#include<string.h>
#define IF              1
#define THEN            2
#define ELSE            3
#define ID              4
#define LT              5
#define LE              6
#define EQ              7
#define NE              8
#define GT              9
#define GE              10
}%
/*正规式模式宏定义*/
digit                   [0-9]
alpha                   [a-zA-Z]
alnum                   [a-zA-Z0-9]
delim                   [ \t\n]
ws                      {delim}+
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}                    {yyIval=install_id(); return(ID);}
{number}                {yyIval=install_num(); return(NUMBER);}
"<"                     {yyIval=LT; return(RELOP);}
"<="                    {yyIval=LE; return(RELOP);}
"="                     {yyIval=EQ; return(RELOP);}
"<>"                    {yyIval=NE; return(RELOP);}
">"                     {yyIval=GT; return(RELOP);}
">="                    {yyIval=GE; return(RELOP);}
int  main()
{
yyparse();
return 0;
}
```

```
int yyerror(char * msg)
{
printf("Error
encountered: % s \n", msg);
}
```

程序开始是内部表示常数的定义,即用字符%{和}%括起来的部分。接下来是说明部分的模式宏定义。正规式规则定义部分出现的第一个正规名 delim,它对字符属性的描述是无意义的,仅表示 3 个字符,即空白、回车标记(\t)、换行标记(\n)。注意:正规式允许递归定义。

程序中第一次出现的字符%%标记识别规则部分,表示当某个模式被识别,则执行其后的处理动作。例如,当关键字 if 被识别,则产生执行语句 return(IF),该单词属性为 IF,此时,单词属性字的值部分为空。在识别规则中,对关系运算符"<"的识别,其动作部分由两条语句来完成,一是给出单词"<"的属性 RELOP,由语句 return(RELOP)完成;二是给出单词的内部值,由语句 yylval=LT 完成。而对标识符的识别,其动作部分也是由两条语句完成的,但在给出标识符内部值的语句中,则是转去调用辅助过程部分的一个过程 install_id,即由该过程给出相应的标识符的内部值。

Lex 程序的第三部分是辅助过程,定义了两个用 C 语言编写的函数,省略了具体代码,仅给出了函数的功能说明,其功能是完成对标识符及数值型常数的内部值的处理。

3.5.4 Lex 应用

作为对 Lex 应用的理解,给出如下实例。

【例 3-12】 编写 ANSI C 的 Lex 源程序如下:

```
    D          [0-9]
    L          [a-zA-Z]
    H          [a-fA-F0-9]
    E          [Ee][+-?]{D}*
    FS         (f|F|l|L)
    IS         (u|U|l|L)*
  %{
#include<stdio.h>
#include "y.tab.h"
void count();
}%
%%
"/*"                          {comment();}
"auto"                        {count();return(AUTO);}
"break"                       {count();return(BREAK);}
"case"                        {count();return(CASE);}
"char"                        {count();return(CHAR);}
"const"                       {count();return(CONST);}
"continue"                    {count();return(CONTINUE);}
"default"                     {count();return(DEFAULT);}
"do"                          {count();return(DO);}
```

```
"double"                    {count();return(DOUBLE);}
"else"                      {count();return(ELSE);}
"enum"                      {count();return(ENUM);}
"extern"                    {count();return(EXTERN);}
"float"                     {count();return(FLOAT);}
"for"                       {count();return(FOR);}
"goto"                      {count();return(GOTO);}
"if"                        {count();return(IF);}
"int"                       {count();return(INT);}
"long"                      {count();return(LONG);}
"register"                  {count();return(REGISTER);}
"return"                    {count();return(RETURN);}
"short"                     {count();return(SHORT);}
"signed"                    {count();return(SIGNED);}
"sizeof"                    {count();return(SIZEOF);}
"static"                    {count();return(STATIC);}
"struct"                    {count();return(STRUCT);}
"switch"                    {count();return(SWITCH);}
"typedef"                   {count();return(TYPEDEF);}
"union"                     {count();return(UNION);}
"unsigned"                  {count();return(UNSIGNED);}
"void"                      {count();return(VOID);}
"volatile"                  {count();return(VOLATILE);}
"while"                     {count();return(WHILE);}
{L}({L}|{D})*               {count();return(check type());}
0[xX]{H}+{IS}?              {count();return(CONSTANT);}
0{D}+{IS}?                  {count();return(CONSTANT);}
{D}+{IS}?                   {count();return(CONSTANT);}
L?'(\\.|[^\\'])+'           {count();return(CONSTANT);}
{D}+{E}{FS}?                {count();return(CONSTANT);}
{D}+"."{D}+({E})?{FS}?      {count();return(CONSTANT);}
{D}+"."{D}*({E})?{FS}?      {count();return(CONSTANT);}
L?\"(\\.|[^\\"])*\"         {count();return(STRING_LITERAL);}
"..."                       {count();return(ELLIPSIS);}
">>="                       {count();return(RIGHT_ASSIGN);}
"<<="                       {count();return(LEFT_ASSIGN);}
"+="                        {count();return(ADD_ASSIGN);}
"-="                        {count();return(SUB_ASSIGN);}
"*="                        {count();return(MUL_ASSIGN);}
"/="                        {count();return(DIV_ASSIGN);}
"%="                        {count();return(MOD_ASSIGN);}
"&="                        {count();return(AND_ASSIGN);}
"^="                        {count();return(XOR_ASSIGN);}
"|="                        {count();return(OR_ASSIGN);}
">>"                        {count();return(RIGHT_OP);}
"<<"                        {count();return(LEFT_OP);}
"++"                        {count();return(INC_OP);}
"--"                        {count();return(DEC_OP);}
"->"                        {count();return(PTR_OP);}
"&&"                        {count();return(AND_OP);}
"||"                        {count();return(OR_OP);}
```

```
"<="                    {count();return(LE_OP);}
">="                    {count();return(GE_OP);}
"=="                    {count();return(EQ_OP);}
"!="                    {count();return(NE_OP);}
";"                     {count();return(';');}
("{"|"<%")              {count();return('{');}
("}"|"%>")              {count();return('}');}
","                     {count();return(',');}
":"                     {count();return(':');}
"="                     {count();return('=');}
"("                     {count();return('(');}
")"                     {count();return(')');}
("["|"<:")              {count();return('[');}
("]"|":>")              {count();return(']');}
"."                     {count();return('.');}
"&"                     {count();return('&');}
"!"                     {count();return('!');}
"~"                     {count();return('~');}
"-"                     {count();return('-');}
"+"                     {count();return('+');}
"*"                     {count();return('*');}
"/"                     {count();return('/');}
"%"                     {count();return('%');}
"<"                     {count();return('<');}
">"                     {count();return('>');}
"^"                     {count();return('^');}
"|"                     {count();return('|');}
"?"                     {count();return('?');}
[\t\v\n\f]              {count();}
.                       {/* ignore bad characters */}
%%
yywrap()
{
   return(1);
}
comment()
{
   char c,c1;
   loop;
   while((c=input())!='*'&&c!=0)
        putchar(c);
   if((c1=input())!='/'&&c!=0)
   {
        unput(c1);
        goto loop;
   }
   if(c!=0)
        putchar(c1);
}
int column=0;
void count()
```

```
{
   int i;
   for(i = 0;yytext[i]!= '\0';i++)
        if(yytext[i] == '\n')
            column = 0;
        else if(yytext[i] == '\t')
            column += 8 - (column % 8);
        else
            column++;
}
int check_type()
{
/ * pseudo code --- this is what it should check
 *
 * if(ytext == type_name)
 *        return(TYPE_NAME);
 *
 * return(IDENTIFIER);
 * /
/ *
 * it actually will only return IDENTIFIER
 * /
    return(IDENTIFIER);
}
```

习题

1. 写出表示下列语言的正规式。

(1) {0,1}中不含形如 01 的所有串。

(2) {0,1}中至少只含两个 1 的所有串。

(3) {0,1}中最多只含两个 1 的所有串。

(4) {0,1}中以 1 开头和以 1 结尾的所有串。

(5) {0,1}中以 01 结尾的所有串。

(6) 能被 5 整除的十进制数。

(7) {0,1}中的含有子串 010 的所有串。

(8) 英文字母组成的所有符号串,要求符号串中的字母按字典序排列。

2. 构造下列正规式的等价 DFA。

(1) $(a|b)^*$

(2) $(a^*|b^*)^*$

(3) $(0|1)^*(0^*|1)^*(0|1)$

(4) $((\varepsilon|a)b^*)^*$

(5) $(a|b)^*a(a|b)$

3. 写出接受的字符串是分别满足和同时满足如下条件的确定的有限自动机及相应的正规式,$\Sigma=\{0,1\}$。

(1) 1 的个数为奇数。

(2) 两个 1 之间至少有一个 0 隔开。

4. 已知文法 $G=(\{S,A,B\},\{a,b,c\},P,S)$,其中:

$P:S\rightarrow aS\mid aB$

$B\rightarrow bB\mid bA$

$A\rightarrow cA\mid c$

构造一个与文法 G 等价的正规式 R。

5. 填空题。

(1) 设有 C 语言的程序段如下:

```
while(i!= j)
{ a = 34;
  j = k;
  i++;
}
```

则经过词法分析后可以识别的单词个数为(　　)。

(2) 设定义在字母表$\{a,b,c,x,y,z\}$上的正规式 $r=(a\mid b\mid c)(x\mid y\mid z)$,则 $L(r)$中元素有(　　)个。

(3) 正则表达式 $R1$ 和 $R2$ 等价是指(　　)。

(4) 已知文法 $G[S]:S\rightarrow A1$, $A\rightarrow A1\mid S0\mid 0$,与 G 等价的正规式是(　　)。

6. 判断题。

(1) Lex 是典型的词法分析程序。(　　)

(2) 词法分析的依据是源语言的文法规则。(　　)

(3) 源程序中的单词是具有独立意义的短语。(　　)

(4) 单词的属性字一般应该包括单词类别和单词内码。(　　)

(5) 编译的预处理程序的处理对象是源程序。(　　)

7. 设 A,B,C 为任意的正规式,试证明正规式的如下性质:

(1) $A\mid B=B\mid A$

(2) $A\mid(B\mid C)=(A\mid B)\mid C$

(3) $A(BC)=(AB)\mid C$

(4) $(A\mid B)C=AC\mid BC$

(5) $(A^*)^*=A^*$

(6) $A\mid A=A$

(7) $\varepsilon\mid A=A\varepsilon=A$

(8) $(AB)^*A=A(BA)^*$

8. 非确定的有限自动机 M 的状态图如图 3-21 所示,写出与其语言等价的正规式 r。

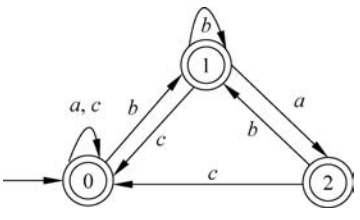

图 3-21　NFA M 状态转换图

9. 简答题。

(1) 词法分析程序的基本功能有哪些?词法分析的性质是什么?

(2) 词法分析中识别的单词具有什么特征？识别的依据是什么？

(3) 阐述词法分析器自动生成的思想。

(4) 何为超前搜索技术？

10. 用伪代码编写一个 C 源程序预处理程序，要求如下。

(1) 去除源程序中的注释(源程序中的注释用/*……*/标记，不允许嵌套使用)。

(2) 去除源程序中的续行符(\)。

(3) 将 Tab 和换行符替换为空格。

(4) 将大写字母变换成小写。

(5) 在源程序尾部添加字符 #，这是编译程序内部的一个特殊的单词，以示源程序结束。

(6) 每调用一次，将经预处理的源程序全部送入内存中的扫描缓冲区，供扫描器识别单词。

11. 假设有两类单词：①关键字 if；②标识符，它表示除 if 之外的所有字母组成的串。

(1) 给出识别这两类单词的 NFA。

(2) 给出识别这两类单词的 DFA。

12. 假设有一类单词为无符号数，例如：123，12.3，1.23E+2，1.23e-4。

(1) 写出该单词对应的文法。

(2) 给出识别这两类单词的 NFA。

(3) 给出识别这两类单词的 DFA。

13. C++有各种整型常量：十进制数的简单序列可以看作整型常量，0x 为前缀的十六进制数序列为整型常量，以 0 为前缀的八进制数序列为整型常量，以 L 或 l 为后缀的整型常量表示的类型为 long int。编写一个描述用于识别 C++中整型常量的正规式，并构造相应的 NFA 和 DFA。

14. 总结 C 和 C++语言的全部单词种类。

15. 指出下列 Lex 正规式所匹配的字符串：

(1) "{"[^{]*"}"

(2) [^0-9]|[\r\n]

(3) \'([^'\n]|\'\')+\'

(4) \"([^"\n]|\["\n])*\"

16. 编写一个 Lex 程序，该程序将程序中的关键字 int 的每个实例转换成 float。

17. 编写描述 C++的单词符号的 Lex 程序。

18. 编制 Lex 源程序，使用 Lex 实现统计 C 源文件中非注释的行数。