编译程序设计算法

刘晓东　　傅　强
朱建刚　林瑞春　米祖强　编著

前　言

高级程序设计语言的编译程序既是计算机系统软件最重要的组成部分之一，也是计算机用户最直接关心的工具之一。从20世纪50年代中期世界第一个编译程序——FORTRAN编译程序的诞生至今，经过人们40多年的努力，编译理论与技术得到迅速发展，已形成了一套比较成熟的、系统化的理论与方法。我们并且开发出了一些好的编译程序的实现语言、环境和工具，这使得在此基础上设计并实现一个高级语言的编译程序，不再是高不可攀的事情。

本书主要介绍设计和构造高级语言编译程序中的基本原理及各种算法，重点描述了设计编译程序所涉及的各种算法。书中给出的算法涵盖了程序设计语言的文法及编译程序设计各个阶段的基本技术和主要算法。全书共分11章。第1章对编译程序进行了简要介绍，第2章、第3章介绍了有限自动机与正则表达式及上下文无关文法中的基本算法，第4章至第6章、第10章、第11章讲解了语法分析、语法制导的翻译、代码优化、代码生成等编译阶段中的基本技术和主要算法，第7章至第9章介绍了符号表管理、内存管理和出错处理的主要算法。书中采用类C语言描述了各种算法，并通过大量的例子来解释它们在编译程序设计中的具体应用。

本书的第1章至第3章和第11章由刘晓东编写，第4章、第5章由傅强编写，第6章、第7章由林瑞春编写，第8章、第9章由朱建刚编写，第10章由米祖强编写。刘晓东对全书进行了统稿。

由于我们学识浅薄，编写时间仓促，谬误之处在所难免，恳切希望读者不吝赐教。

目　录

此为试读,需要完整PDF请访问: www.ertongbook.com

第1章 绪论

1.1 编译程序的基本概念

编译程序(compiler)是指将高级语言程序翻译成为功能等价的其他语言程序的程序,也称为编译器。因此,编译程序的本质是一个翻译程序,其需要翻译的高级语言称为源语言,翻译后的低级语言称为目标语言。从某种角度来看,编译程序实际上是用来在特定的计算机上实现某种高级语言。交叉编译程序(crosscompiler)是指这样一类编译程序,即它虽然是在某一类型的计算机上运行,但是可以生成在另一类型的计算机上运行的目标代码。交叉编译的方法可用于实现别的编译器,这类编译器可以通过源语言、目标语言和编译器实现语言三个方面来刻画其特征。如果某一语言的编译器是由该语言本身编写、实现的,则称这种方式为自编译或自举(bootstrap)方式编译器。那么,如何实现自举方式的编译器呢?

假设有一个新语言L,我们期望在A机器和B机器上分别实现其编译程序。首先,我们用A机器上已有的某种语言编写一个小的编译程序${}^{S}C_{A}^{A}$,用于将L的一个子集S所编写的程序翻译为A机器上的目标代码。其次,我们用S编写L的编译程序${}^{S}C_{S}^{A}$,用于将L翻译为S的代码。由于S的代码不能在A机器上直接运行,因此${}^{S}C_{S}^{A}$必须作为${}^{S}C_{A}^{A}$输入,经过翻译才能成为L在A机器上的编译程序${}^{L}C_{A}^{A}$,即:

${}^{S}C_{S}^{A}\rightarrow{}^{S}C_{A}^{A}\rightarrow{}^{L}C_{A}^{A}$

在此基础上,我们可以采用如下的步骤,用L语言本身实现其在B机器上的编译器。

${}^{L}C_{L}^{B}\rightarrow{}^{L}C_{A}^{A}\rightarrow{}^{L}C_{A}^{B}$

${}^{L}C_{L}^{B}\rightarrow{}^{L}C_{A}^{B}\rightarrow{}^{L}C_{B}^{B}$

其中,${}^{L}C_{L}^{B}$首先经过${}^{L}C_{A}^{A}$的翻译生成可以在B机器上运行的${}^{L}C_{A}^{B}$,${}^{L}C_{A}^{B}$的输入为L,输出为B机器上的可执行代码。因此,将${}^{L}C_{L}^{B}$再经过${}^{L}C_{A}^{B}$翻译就可以生成运行于B机器上的L的编译器${}^{L}C_{B}^{B}$。

1.2 编译程序的结构

1.2.1 编译程序的翻译过程

编译程序的翻译过程非常复杂,一般把它分为若干阶段,而且每个阶段的输入都被看做源程序的某种表示形式,相应的输出被做作另一种形式,并且作为下一阶段的输入。图1-1是一种典型的编译过程的阶段划分。

此为试读,需要完整PDF请访问: www.ertongbook.com

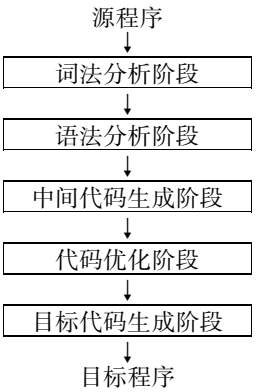

图 1－1　编译的过程

1.2.1.1　词法分析

词法分析是编译程序最初的处理阶段,其分析的对象是源程序。词法分析器对组成源程序的字符串从左向右扫描,从中识别出符合词法规则的具有独立意义的单词符号,或称记号(token),如关键字、运算符、分隔符、标识符等,并将识别出的单词符号以一种约定的形式来表示。例如,对于赋值语句 $X:=Y+Z$,经过词法分析后变化为 $id_1:=id_2+id_3$。其中的 id_1,id_2,id_3 分别为 X,Y,Z 对应的内部约定形式,如符号表中对应的下标或指针等。这样,源程序中的字符流经过词法分析就成为由内码表示的单词符号流,并作为语法分析的输入。

1.2.1.2　语法分析

语法分析是根据语法规则对单词符号串进行分析、识别,判定它们是否构成了各类语法单位,如短语、句子、程序等。词法分析采用层次结构分析,其输出通常为源程序的一棵语法树。如上述赋值语句的语法树可用图 1－2 来表示。

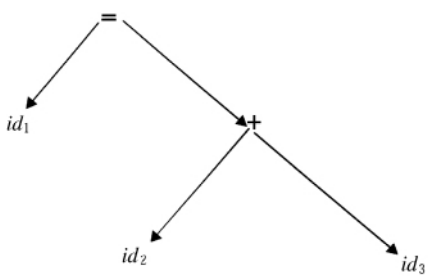

图 1－2　赋值语句 $X:=Y+Z$ 的语法树

1.2.1.3　中间代码生成

在语法分析的基础上,就可以开始语义分析,即对识别出的各类语法单位,根据语言的语义,用另一种语言形式来描述其语义。如果直接用目标语言来描述源程序的语义,可直接翻译为目标语言程序。然而,考虑到编译程序的实现、目标程序的效率、移植等因素,大多数编译采用中间语言来描述源程序的语义,即先生成中间语言代码。中间语言对应于某种抽

象的机器,语义明确,结构简单,并且易于将它们翻译成目标语言。中间语言有很多种形式,下面的例子给出了图1-2的三地址码(TAC)的语义描述形式:

$T_1 := id_2 + id_3$

$id_1 := T_2$

其中T_1,T_2为编译产生的临时存储单元。

1.2.1.4 代码优化

代码优化是指对中间代码进行各种各样的变换,如删除冗余运算、删除无用赋值、合并已知量等,这些变换将使代码运行得更快、占用的空间更少,即提高代码的运行效率。代码优化中最主要的是循环优化。

1.2.1.5 目标代码生成

目标代码生成是编译过程的最后阶段,它根据优化后的目标代码和其他有关信息,生成目标机器的二进制代码程序或汇编语言程序。此阶段与目标机器密切相关,应充分考虑目标机器的硬件资源,如寄存器的充分利用、指令的合理选择、内存的有效利用等,以生成高效的目标代码。如果生成的目标代码是汇编语言程序,则需经过汇编程序生成可重定位指令代码。对生成的可重定位代码,需由连接装配程序,将输入输出、标准函数等系统模块与目标模块连接在一起,并确定各数据和各程序点在内存中的地址,成为可运行的绝对指令代码。

从上述的编译程序的逻辑结构中不难看出,词法分析、语法分析、中间代码生成、中间代码优化等阶段独立于目标机器的指令代码,主要是针对源语言进行分析和变换。这些阶段构成了编译器的前端;而那些依赖于目标机器的阶段,如代码生成阶段,构成了编译器的后端。这种前端、后端的划分方法有利于编译程序的移植,即针对同一个语言在不同机器上的实现,其编译器的前端可以是一样的,只需根据不同的机器指令和硬件资源编写后端程序即可。

1.2.2 遍

前面介绍的编译过程的五个阶段,仅仅是从逻辑功能上划分的。具体实现时,受到不同的源语言、设计要求、适用对象和计算机资源的限制,往往将编译程序组织成为若干遍。所谓“遍”,就是对源程序或其中间结果从头到尾进行的一次扫描,并进行有关的分析、变换和处理,生成新的中间结果或目标程序。根据不同的需要和要求,既可以将几个不同的阶段合并为一遍,也可以把一个阶段的工作分为若干遍。从效率的角度来看,人们总希望用相对少的扫描次数(遍)来完成编译各阶段的工作。然而,有些阶段可以比较容易地合为一遍,而有些阶段却难以合并。例如,由于词法分析阶段和语法分析阶段的接口为单词符号,而单词符号之间是相互独立的,因此,可以很容易地将词法分析和语法分析合为一遍;为了便于处理,语法分析和中间代码生成也常常合为一遍。由于目标代码的生成不仅需要根据当前已经得到的中间代码,而且往往需要依赖于后续产生的中间代码来提供必要信息,因此,很难将中间代码生成阶段和目标代码生成阶段合为一遍。同时,考虑到优化的要求,往往还把优化阶

段分为若干遍来实现。

当一遍中包含若干阶段时,各阶段的工作是穿插进行的。例如,我们可以把词法分析、语法分析和中间代码生成这三个阶段安排成一遍,这时,词法分析处于核心位置,当它在识别语法结构需要下一个单词符号时,就调用词法分析器,一旦识别出一个语法单位时,就调用中间代码产生器,完成相应的语义分析并产生相应的中间代码。

1.2.3 编译程序的基本构成

根据编译过程的五个阶段,编译程序可以由词法分析器、语法分析器、中间代码生成器、代码优化器、目标代码生成器五个功能模块和表格管理、出错处理两个辅助部分构成,如图1-3所示。

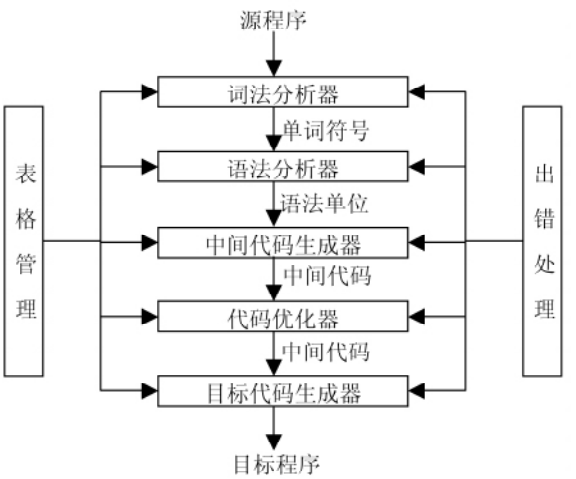

图1-3 编译程序结构框图

1.2.3.1 表格与表格管理

编译程序在工作中需要建立和维护一系列表格,以登记源程序的各类信息和编译各个阶段的进展状况,其中最重要的是用来登记源程序中出现的每个名字及其各种属性的符号表。例如,变量名、类型、数据的结构、存储位置及相应的值等。通常,编译程序在处理到名字的定义性出现时,要把名字的各种属性填入到符号表中,不同的属性需要在不同的阶段填入,如标识符的名字可以在词法分析中填入,类型要等到语法分析时才能确定,而地址可能要等到目标代码生成时才能确定;当处理到名字的使用性出现时,需要对名字的属性进行查证。

随着编译的各个阶段的推进,对各种表格中的数据将不断会有建立、定义、引用、查找、更新等操作。编译的不少时间是花在建表、查表、更新等操作上,所以选择适当的表结构及其相应算法,对于编译程序来说是至关重要的。

1.2.3.2 出错处理

一个编译程序不仅应对正确的程序进行翻译,而且应能够发现源程序中的错误,并立即

报告错误的性质、类型及位置,同时将错误所造成的影响限制在尽可能小的范围内,使得源程序的其余部分能继续被编译下去,以便进一步发现其他可能的错误。这部分工作一般由专门的出错处理程序来完成。理想的情况是不仅能够发现错误,而且还能够自动校正错误。但是,自动校正错误的代价是非常高的。

编译过程的每一个阶段都可能检测到错误,其中,绝大多数错误可以在编译的前三个阶段被检测出来。源程序的错误一般分为语法错误和语义错误两大类。语法错误是指在源程序中不符合语法(或词法)规则的错误,它们可以在词法或语法分析时检测出来。语义错误是指源程序中不符合语义规则的错误,这类错误一般在语义分析时检测出来,有的语义错误要在运行时才能检测出来。

1.3 预备知识

1.3.1 字符串

字母表是指有限字符符号的集合,通常用 Σ 表示。

字符串是由 Σ 中的符号构成的有穷序列,通常用 α,β,γ 等希腊字母表示。字符串 α 的长度是指构成 α 的字符的个数,用 $|\alpha|$ 表示。例如,若 $\alpha=xyz$,则 $|\alpha|=3$;如果 $|\alpha|=0$,则称 α 为空字符串,通常用 ε 表示。

一个字符串的前缀是指该字符串任意的前部子串。例如,$\alpha=xyz$,则 ε,x,xy,xyz 均为 α 的前缀。一个字符串除去它自身以外的前缀称为其真前缀。一个字符串的后缀是指该字符串任意的后部子串,例如,$\alpha=xyz$,则 ε,z,yz,xyz 均为 α 的后缀。一个字符串除去它自身以外的后缀称为其真后缀。

一个语言是指特定字母表中字符串的集合。例如,$\Sigma=\{0,1\}$,则定义在该字母表上的一个语言可以是 $L=\{\varepsilon,0,00,000,1,11,111,\cdots\}$。

1.3.2 集合运算

设 A 和 B 分别为两个集合,则:

A 和 B 的并集记为 $A\cup B=\{x\mid x\in A$ 或者 $x\in B\}$;

A 和 B 的交集记为 $A\cap B=\{x\mid x\in A$ 并且 $x\in B\}$;

A 和 B 的差集记为 $A-B=\{x\mid x\in A$ 并且 x 不属于 $B\}$;

A 和 B 的笛卡儿乘积记为 $A\times B=\{(a,b)\mid a\in A$ 并且 $b\in B\}$;

A 的幂集记为 $2^A=\{P\mid P$ 为 A 的子集$\}$,例如,$A=\{0,1\}$,则 $2^A=\{\varnothing,\{0\},\{1\},\{0,1\}\}$;

A 和 B 的连接积记为 $AB=\{ab\mid a\in A$ 并且 $b\in B\}$;

A 的闭包记为 $A^*=A^0\cup A^1\cup A^2\cup\cdots\cup A^\infty$,其中 $A^0=\{\varepsilon\}$,$A^1=A$,$A^2=AA$,$A^i=A\cdots A$ 为 A 的 i 次连接积。同时,记 $A^+=AA^*$,称 A^+ 是 A 的正则闭包。如果 A 是一个字符串的集合,闭包 A^* 中的所有字符串都是 A 中的字符串经有限次连接而成的,且包含空字符串,而 A^+ 则不

此为试读,需要完整PDF请访问: www.ertongbook.com

包含空字符串。

1.3.3 关系

设 A 和 B 为两个集合,则 A 和 B 之间的一个关系 R 被定义为一个序偶的集合:

$$R=\{(a,b)\mid a\in A,b\in B,\text{并且 } a \text{ 和 } b \text{ 具有关系 } R\}$$

例如,$A=\{0,1\}$,$B=\{1,2\}$,则 A 和 B 上的小于关系(<)可以定义为

$$<=\{(0,1),(0,2),(1,2)\}$$

不难看出,集合 A 和 B 之间的关系 R 是 $A\times B$ 的一个子集。如果序偶 $(a,b)\in R$,则 aRb 为真;否则,aRb 为假。如果 A 和 B 是同一个集合 A,则把 R 称为集合 A 上的关系。

对于集合 A 上的关系 R,如果对于所有的 $a\in A$,均有 aRa,则称 R 是自反关系;如果对于每一个 aRb,$a,b\in A$,均有 bRa,则称 R 是对称关系;如果对于每一个 aRb 和 bRc,$a,b,c\in A$,均有 aRc,则称 R 是传递关系。如果 R 具备自反、传递和对称三个特性,则称 R 是等价关系。

对于集合 A 上的关系 R,设 P 是一个特性集合,则 R 的特性闭包 P - closure 是由 R 中的元素及其根据 P 中所有特性所产生的元素构成的最小集合。如果 P 仅包含传递特性,则 P - closure被称为 R 的传递闭包,记为 R^+;如果 P 包含传递特性和自反特性,则 P - closure 被称为R 的自反传递闭包,记为 R^*。例如,$R=\{(0,1),(1,2),(3,4)\}$,则

$$R^+=\{(0,1),(1,2),(3,4),(0,2)\}$$

$$R^*=R^+\cup\{(a,a),\mid a\in A\}$$

$$=\{(0,1),(1,2),(3,4),(0,2),(0,0),(1,1),(2,2),(3,3),(4,4)\}$$

我们可以通过下面的算法来求得 R^+:

```
{
    R+old = Φ;
    R+new = R;
    while(R+old   != R+new)
    {
        R+old = R+new;
        for( every pair(a,b) and(b,c) in R+old)
       {
          Add pair(a,c) to R+new if not already present;
       }
    }
        R+ = R+new;
}
```

第 2 章　有穷状态自动机和正则表达式

2.1　有穷状态自动机

有穷状态自动机(FA,Finite Automata)包含一组有限的状态和一组定义于特定输入符号的有限的状态转换,其中的一个状态作为自动机的开始状态,称为初始状态,还有一些状态被标识为终止状态。一个有穷状态自动机包含以下五个部分:

自动机的有限状态;

输入符号;

状态转换规则;

初始状态;

终止状态集。

因此,有穷状态自动机 M 可以被定义为如下的 5 个元组:

$$M=(Q,\Sigma,\delta,q_0,F)$$

其中:

Q 是自动机的状态集合,

Σ 是输入符号的集合,

δ 是状态转换函数,

$q_0 \in Q$,是唯一的初始状态,

$F \subseteq Q$,为终止状态的集合。

如果当前状态为 p,对于输入符号 a 存在一个从 p 到 q 的状态转换,则记为 $\delta(p,a)=q$。可以看出,δ 函数的定义域为状态和输入符号序偶的集合,值域为状态的集合。因此,δ 是一个从 $Q\times\Sigma\rightarrow Q$ 的映射。

一个有穷状态自动机既可以用状态转换矩阵来表示,也可以用状态转换图来描述。状态转换矩阵的行表示状态,列表示输入符号,矩阵元素表示 $\delta(p,a)$ 的值。例如,有 FA,其

$$M=(\{q_0,q_1\},\{0,1\},\delta,q_0,\{q_1\})$$

式中 δ 为:

$$\delta(q_0,0)=q_1,\quad \delta(q_0,1)=q_0,$$

$$\delta(q_1,0)=q_1,\quad \delta(q_1,1)=q_0。$$

它所对应的状态转换矩阵见表 2-1,其中 * 表示终止状态。

表 2-1　状态转换矩阵

	0	1
q_0	q_1	q_0
$^*q_1$	q_1	q_0

状态转换图是一个有向图,图中的结点对应于自动机的状态,边(弧)表示状态的转换。如果自动机中有 $\delta(p,a)=q$,则在相应的状态转换图增加一条由结点 p 射向结点 q 且标记为 a 的弧。初始状态节点由⇨表示,终止状态节点用双圈来表示。这样,上述自动机 M 的状态转换图如图 2-1 所示。

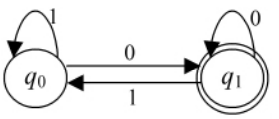

图 2-1　M 的状态转换图

有穷状态自动机可以用来识别某些特定的符号串集合。对于由自动机的输入符号集 Σ 中的符号构成的一个符号串 x(即 Σ^* 中的一个元素),如果存在一条从初始状态节点到某一终止状态节点的通路,且这条通路上所有弧的标记符号构成的字符串等于 x,则称符号串 x 能够被自动机 M 识别。自动机 M 能够识别的所有字符串的全体称为 $L(M)$。由于 x 是 Σ^* 中的一个元素,因此,我们可以定义一个从 $Q\times\Sigma^*$ 到 Q 的映射 δ_1,并且当存在 $\delta_1(q_0,x)=q_f,q_f\in F$ 时,x 能被自动机接受(识别)。如果我们记 $x=wa$,其中 a 为 x 的最后一个字符,w 为 x 的一个前缀,则

$$\delta_1(q_0,x)=\delta(\delta_1(q_0,w),a)$$

$$\delta_1(q_0,a)=\delta(q_0,a)$$

例如,对于自动机 M

$$M=(\{q_0,q_1\},\{0,1\},\delta,q_0,\{q_1\})$$

其中 δ 为:

$$\delta(q_0,0)=q_1,\quad \delta(q_0,1)=q_0,$$

$$\delta(q_1,0)=q_1,\quad \delta(q_1,1)=q_0。$$

假设 x 为 010,为了确定 x 能否为 M 识别,我们可以进行如下的计算:

由

$$\delta_1(q_0,0)=\delta(q_0,0)=q_1$$

得到

$$\delta_1(q_0,01)=\delta\{\delta_1(q_0,0),1\}=q_0$$

进而

$$\delta_1(q_0,010)=\delta\{\delta_1(q_0,01),0\}=q_1$$

由于 $q_1\in F$,所以 $x=010$ 能够被自动机 M 识别。

如果 $x=0101$,则 $\delta_1(q_0,0101)=\delta\{\delta_1(q_0,010),1\}=q_0$。由于 q_0 不是 M 的终止状态,所以不能被 M 识别。

因此,对于自动机 M 能够识别的语言 $L(M)$,可以表示为

$$L(M)=\{x\mid\delta_1(q_0,x)=q_f,q_f\in F\}$$

上述讨论的有穷状态自动机的状态转换函数是从 $Q\times\Sigma^*$ 到 Q 的映射,即对于任一状态上的一个输入符号,其状态转换函数是唯一确定的,因此,这类有穷状态自动机被称为确定的有穷状态自动机,简称 DFA(Deterministic Finite Automata)。

2.2　非确定性有穷状态自动机(NFA)

典型的程序设计语言中都有许多记号,且每一个记号都能被其自己的 DFA 识别出来。如果每一个记号都以不同的字符开头,则只需将其所有的初始状态统一到一个单独的初始状态上,就能很便利地将它们放在一起了。例如,考虑串“:=”、“ <= ”和“ = ”给出的记号。其中每一个都是一个固定串,它们的 DFA 如图 2 -2 所示:

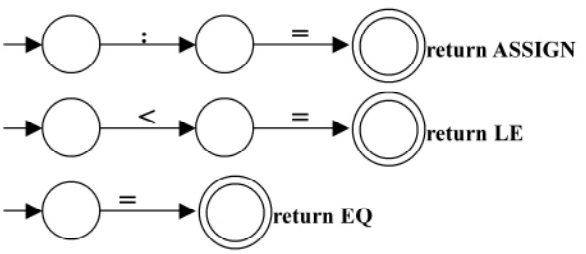

图 2 -2　:=、<=. = 的 DFA

因为每一个记号都是以不同的字符开始的,故只需通过标出它们的初始状态就可得出以下的 DFA,如图 2 -3 所示:

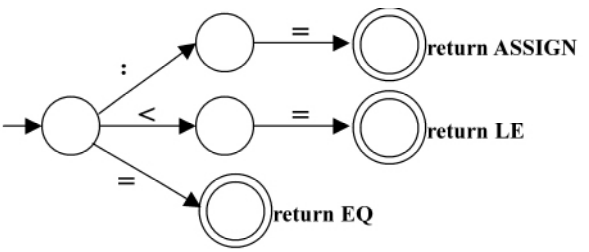

图 2 -3　合并后的 DFA

但是假设有若干个以相同字符开头的记号,例如“ < ”、“ <= ”和“ < > ”,就不能简单地如图 2 -4 所示那样来表示。这是因为它不是 DFA(给出一个状态和字符,则通常肯定会有一个指向单个的新状态的唯一转换)。

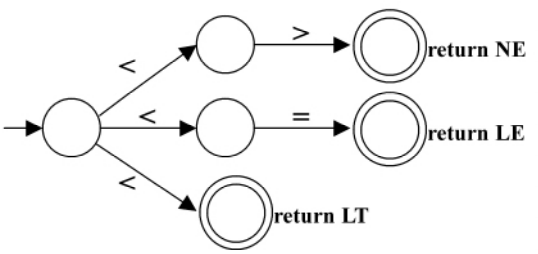

图 2 -4　< 、<= 、<> 的 FA

相反的,我们必须做出安排,以便在每一个状态中都有一个唯一的转换,如图 2 -5 所示。

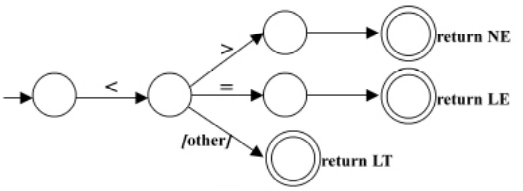

图 2 -5　变换后的 DFA

理论上应该能够将所有的记号都合并为具有这种风格的一个巨大的 DFA,但是它非常复杂,在使用一种非系统性的方法时尤为如此。解决这个问题的一个方法是将有穷自动机的定义扩展到包括了对某一特定字符一个状态存在有多个转换的情况,同时系统地为这些新生成的有穷自动机转换成 DFA 编写一个算法。这里首先讲解到这些生成的自动机,但有关转换算法的内容要在下一节才能提到。新的有穷自动机称为非确定性有穷自动机(Nondeterministic Finite Automata)或简称为 NFA。

NFA 的定义与 DFA 很相似,对于有穷自动机

$$M=(Q,\Sigma,\delta,q_0,F)$$

扩展转换函数 δ 为 $Q\times\Sigma$ 到 2^Q 映射,即令 δ 的值是状态的一个集合而不是一个单独的状态,这样某一状态的每一个字符都可以导致多个状态。例如:

$$M=(\{q_0,q_1,q_2,q_3\},\{0,1\},\delta,q_0,\{q_3\})$$

其中 δ 为:

$$\delta(q_0,0)=\{q_0\},\qquad \delta(q_0,1)=\Phi,$$
$$\delta(q_1,0)=\{q_1\},\qquad \delta(q_1,1)=\{q_1,q_2\},$$
$$\delta(q_2,0)=\Phi,\qquad \delta(q_2,1)=\{q_3\},$$
$$\delta(q_3,0)=\{q_3\},\qquad \delta(q_3,1)=\{q_3\}。$$

其对应的状态转换图如图 2 -6 所示:

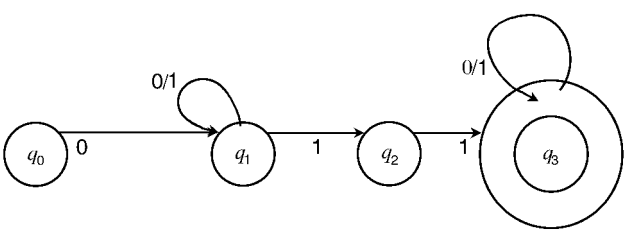

图 2 -6　NFA 的状态转换图

对于 Σ^* 中的任意一个字符串 x,若存在一条从初始状态节点到某一终止状态节点的通路,且这条通路上所有弧的标记构成的字符串等于 x,则称 x 可以为 NFA 所识别(读出或接收)。同 DFA 一样,我们可以定义一个从 $Q\times\Sigma^*$ 到 2^Q 的映射 δ_1,并且当存在 $\delta_1(\{q_0\},x)=P$,且 P 中至少包含一个终止状态,则 x 能被自动机接受。如果我们将其记为 $x=wa$,其中 a 为 x 的最后一个字符,w 为 x 的一个前缀,则

$$\delta_1(\{q_0\},x)=\delta_1(\delta_1(\{q_0\},w),a)$$

$$\delta_1(P,a)=\cup_{q\in P}\delta(q,a)$$

例如,对于如图 2 -6 所示的 NFA,如果 $x=0111$,则可以通过如下的变换来判定 x 能否被该 NFA 接收:

由 $\delta_1(\{q_0\},0)=\delta(q_0,0)=\{q_1\}$ 得

$$\begin{aligned}\delta_1(\{q_0\},01)&=\delta_1(\delta_1(\{q_0\},0),1)\\&=\delta_1(\{q_1\},1)\\&=\delta(\{q_1\},1)\\&=\{q_1,q_2\}\end{aligned}$$

$$\begin{aligned}\delta_1(\{q_0\},011)&=\delta_1(\delta_1(\{q_0\},01),1)\\&=\delta_1(\{q_1,q_2\},1)\\&=\delta(q_1,1)\cup\delta(q_2,1)\\&=\{q_1,q_2\}\cup\{q_3\}\\&=\{q_1,q_2,q_3\}\end{aligned}$$

$$\begin{aligned}\delta_1(\{q_0\},0111)&=\delta_1(\delta_1(\{q_0\},011),1)\\&=\delta_1(\{q_1,q_2,q_3\},1)\\&=\delta(q_1,1)\cup\delta(q_2,1)\cup\delta(q_3,1)\\&=\{q_1,q_2\}\cup\{q_3\}\cup\{q_3\}\\&=\{q_1,q_2,q_3\}\end{aligned}$$

由于 $\delta_1(\{q_0\},0111)=\{q_1,q_2,q_3\}$,且 $q_3\in P$,所以,该字符串能够被该 NFA 接收。

若M 是一个 NFA,则可被 M 识别的语言 $L(M)$可定义为

$$L(M)=\{x\mid\delta_1(\{q_0\}x)=P,P\text{ 中至少包含一个终止状态}\}$$

2.3　带有 ε 弧的非确定性有穷状态自动机

如果自动机允许在没有输入符号的情况下发生状态转换,则称这类有穷自动机为带有 ε 弧的 NFA,其对应的五元组 $M=(Q,\Sigma,\delta,q_0,F)$中 δ 被扩展为 $Q\times(\Sigma\cup\{\varepsilon\})$到 2^Q映射。这类 DFA 中无需考虑输入串(且无需消耗任何字符)就有可能发生的转换称为 ε -转换(ε -transition),它可看作是一个空串的“匹配”。图 2 -7 给出了一个带有 ε 弧的 NFA。

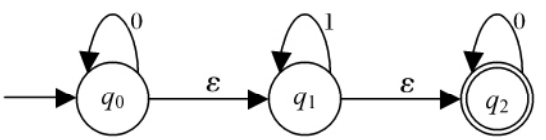

图 2 -7　带有 ε 弧的 NFA

ε -转换在图 2 -8 中的表示就好像是一个真正的字符。

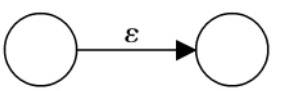

图 2 -8　ε -转换

此为试读,需要完整PDF请访问: www.ertongbook.com