第 1 章 绪 论

随着科学技术的发展，无数的计算系统不断为现实世界带来令人惊奇的成果，折射出来人类生存与发展能力的不断提升。不难发现，许许多多计算机和程序形成难以计数的计算系统，无不是被人们一一构建出来的，其结果深刻地影响着真实世界的技术走向。解决如何有效地构建计算系统这样的技术问题，编译理论与技术在其中起到了不可替代的作用。编译系统将计算的自然表达转换为图灵机（Turing machine，TM），为人们构造众多计算系统提供了可能。理论上，每个图灵机都是一个可运行的计算系统，并在冯·诺依曼结构下得以存在，。人类并不擅长于运用这种表达方式，因此依赖于编译系统来建立有用的抽象层次以消除人与机器之间的隔阂。

编译系统的功能是把高级语言程序等价转换为低级语言程序，完成源语言、中间语言和目标语言至少三个抽象层次上的等价关联，为此形式语言理论被作为编译原理和编译技术的重要基础。建立在形式语言理论基础上的编译理论体系完整，能够很好地回答“是什么”、“如何解决”以及“为什么”这类与学习技能有关的问题。形式语言与编译所述内容注重编译理论方面的启发性，并试图体现出化难为易的效果。

作为本书的第 1 章，本章从概述的角度，介绍以下几方面的内容：首先简要介绍计算系统的理论模型，即图灵机和冯·诺依曼结构；其次介绍编译程序的功能及工具链，示意从高级语言程序到可执行程序的转换方式；再次介绍编译过程，围绕一个简单实例阐述词法分析、语法分析、语义分析、代码优化、代码生成这些典型的编译阶段，展现对编译过程的总体认识并介绍相关术语，后续章节内容都可以在这个例子中找到对应之处；最后概述形式语言及其中心概念，使读者初步了解词法、语法、语义分析涵盖的有关术语。

1.1 认识编译

编译是一个抽象概念，可具体化为编译过程、编译理论、编译程序等。本节首先使读者对建立在图灵机概念基础上的编译这一抽象概念建立认识，随后各节将就编译的具体方面展开阐述。

计算机科学是关于计算理论和计算机设计的科学分支，自 20 世纪 40 年代第一台计算机诞生以来，计算机科学得到了快速发展。尽管计算机科学发展至今还不满百年，却已产生形形色色的计算系统，为人们带来方方面面的变革和便捷，除了自身理论、技术和应用变得丰富而广阔以外，还促成了各个科学分支中热门的交叉增长点，展现出能够解决越来越多的各学科具有挑战性的问题的前景。

伴随着计算机的诞生，程序设计语言随之出现。利用高级语言，人们可以方便地将问题求解过程表达出来，形成可以存储于计算机内的程序，实现了快速自动计算这一宏伟理想。

存储程序计算机采用从图灵机计算模型发展出来的冯•诺依曼结构，也就是第一台计算机的组成结构。到目前为止，实际应用的计算机所采用的结构仍然没有超出冯•诺依曼结构，所以图灵机理论被公认为计算系统的理论基础。

基于存储程序计算机的结构，计算系统表现为不外乎计算装置及可存储的程序这两个部分的统一体。虽然我们可以采用软硬件协同设计的方法实现更优化的统一体，但大部分情况下计算装置具有相对独立性，从而使我们所做出的大部分问题求解努力聚焦到程序设计方面。就计算装置而言，它可以小到是一台微处理器，大到是超级计算机，也可以是抽象的机器，它们都源自图灵给出的计算模型。

从以上讨论可以理出一条思路，就是编译的任务本质上可理解为将任何计算系统转换为图灵机，更具体地，转换为能够运行在冯•诺依曼结构上的计算机程序。

程序被计算装置直接执行以获得计算结果，这是每个计算系统存在的原因。就计算机而言，根据冯•诺依曼结构，能够被直接执行的程序是二进制代码，这种程序采用了机器语言的表达方式（通常说这个程序是用机器语言写的）。机器语言并不方便于人用来写程序，尽管早期的不少计算机程序都由人采用机器语言以手工方式写成。20 世纪 80 年代，大学里仍有一门称为“手编程序”的课程，教授一种面向 DJS-1X 计算机（一种早期的国产晶体管计算机）的机器代码程序设计，一种特别枯燥的、需要过分谨慎的、带有一丝亲切的氛围弥漫在课堂内外，甚至于留在长期记忆里。由于二进制程序的任意局部都是有歧义的，虽然在知道整个程序的情况下能够得到消解，但是编程过程就像面对处处是歧途而走迷宫时的情形，心情和效率都受到极大挑战。

自然地，汇编语言首先有必要被设计出来以改进程序设计效率。汇编语言中引入助记符替代 0-1 串，从而允许相应含义的单词或缩写词被用于替代那些作为指令、地址，甚至数据的 0-1 串。而采用汇编语言写的程序，借助于汇编器可直接转换为机器语言程序，得以在计算机上执行。这样，采用汇编语言开发程序相比于采用机器语言在很大程度上减轻了编程负担，但是编程效率仍然很低。

高级语言的出现让构建程序成为可普遍接受的任务。据统计，现有编程语言达数千种之多，其中包括通用语言如 C 语言以及专用语言，在这些语言中，每年还有上百种“新面孔”出现。语言的分层抽象思想让众多语言获得分类带来的好处。一类语言被定格在低级语言层面，包括机器语言和汇编语言，另一类属于高级语言，人类可以较为方便地使用。低级语言与机器保持强相关性，由于硬件机器种类繁多，决定了低级语言的普遍存在有其必然性。低级语言的共同特点是由计算机的指令系统决定其表达形式。而高级语言是参照数学表达方式而设计的近似于日常会话的语言，还常常被引申借用作为描述算法的语言。高级语言自身成为一种独立于机器的面向过程或面向对象等多种范型的语言，使用高级语言大大减轻了程序员的编程负担。然而，用高级语言所写的程序不能为计算机直接执行，因此需要对其进行编译，从而产生出等价的低级语言程序，才能使程序得以运行。

把高级语言写的程序编译为机器代码程序，通常使用一个称为编译程序或编译器的系统软件来完成，这样的编译可表现为一个编译过程的实施，也可以表现为一个编译框架上的功能有效性的体现。因此，总的来说，借助于编译，使得构建计算系统的复杂度得以显著降低，而产品率得以显著提升。从中可以看出，对于编译理论知识的兴趣和对编译系统工程化的热衷，都源于自由探索精神和现实世界的驱动力。

1.2　图灵机与冯・诺依曼结构

作为计算系统的理论模型，图灵机和冯•诺依曼结构回答了一些与编译相关的根本问题，比如编译的目标是什么，编译为什么是不可替代的，等等。鉴于此，这一节简要介绍图灵机和冯•诺依曼结构，有助于明确编译知识的一些基本论域和前提。

图灵机是由英国数学家阿兰•图灵（A. M. Turing）于 1936 年提出的一种抽象计算模型，即将人们使用纸笔进行数学运算的过程进行抽象，由一个虚拟的机器替代人们进行数学运算。

图灵机模型如图 1-1 所示，它由一个有限状态控制器、一个读写头和一条无限长纸带组成。纸带上面划分为一个一个的格子，每个格子里可以写一个符号或者为空白（特别地，空白的格子可以看作为格子里写着一个特殊的空白符号）。读写头可以读出格子里的符号或写入一个符号到格子里，读写头还能向左或向右依次移动一个格子。控制器有有限个状态和状态转移规则在其中，控制器存储当前自身的状态。

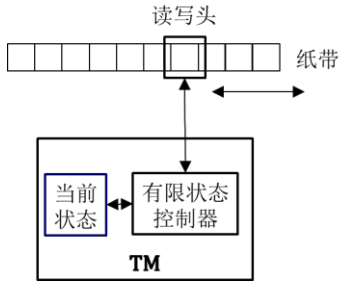


图 1-1 图灵机模型示意图

在图灵机运行之前，假定有一个输入符号串已经在纸带上一段连续的格子里写好，其中每个格子里写有一个符号，其余格子均为空白。初始时读写头位于输入串的左端第一个符号所在的格子上，同时状态控制器的当前状态设为初始状态，然后根据读写头读出的符号以及当前状态，由控制器决定转移到哪个状态，并决定写入什么符号，以及决定读写头移动到左邻格子还是右邻格子上。图灵机的运行过程就是重复地进行如上所述的状态转移过程，如果状态转移伴随停机则视当前状态是否为接受状态来决定是否接受输入串。

控制器的状态转移规则形如：

(<当前状态>, <读出符号>)→(<下一状态>, <写入符号>)<下一格>

其中，图灵机的控制器根据当前状态和读出符号决定它的转移状态、写入符号和读写头移入位置。如此一直运行下去，直到根据当前读入符号和当前状态知道需要停机为止。

例 1.1 计数器。

初始值以二进制数即 0-1 串形式写在纸带上，纸带上其余格子为空格，因此读写头读入的符号有 0、1 和空格。有如下转移规则：

(1, 0)→(0, 1) Left 若机器处于状态 1 且单元内容为 0，则将当前单元内容修改为 1，并向左移动一个格子，同时状态转为 0；

(1, 1)→(1, 0) Left 若当前状态为 1 读入符号为 1 则将当前单元内容置为 0，停留状态 1 不变并将读写头向左移动一格；

(1, B)→(0, 1) Right 若当前单元内容为空白则将其修改为 1，状态转为 0 并向右移动一格；

(1, B)→(0, 1) Halt 停机指令，若当前单元为空白则改为 1，状态转 0 并停机。

按此规则构建的计数器图灵机模型如下：

(0, 1)→(0, 1) Right

(0, 0)→(0, 0) Right

(0, B)→(1, B) Left

(1, 0)→(0, 1) Right

(1, 1)→(1, 0) Left

(1, B)→(0, 1) Right

这个图灵机没有用到停机指令，所以永远不能结束，这不是有效的计算过程。

另一个例子是无符号数加 1 图灵机，其模型为：将上面模型中第 4、第 6 条规则中的 Right 改为 Halt，其他不变。如此更改的图灵机对输入 01 加 1 后就停机，并得出结果 10。

在计算机科学中，通用图灵机（universal Turing machine，UTM）就是能够模拟任意输入的运行过程的图灵机。通用图灵机从根本上是通过从它自己的纸带上读入要模拟的机器的描述以及给定的输入来达成的。图灵于 1936—1937 年提出此机器的概念，被认为是存储程序计算机的起源。冯·诺依曼称这种机器为电子计算装置，它如今被命名为冯·诺依曼结构，也被称为通用计算机器、通用机器（universal machine，UM）等。

具体地说，如果把任意一个图灵机的指令集用图灵自己提出的一种规范方式编码并预存在纸带上，那么通用图灵机就能够根据纸带上已有的信息，在纸带的空白处模拟那台图灵机的运作，输出那台图灵机应该输出的计算结果。

冯·诺依曼结构是已知的唯一作为编译程序的目标机的结构模型。该模型的特点是采用二进制形式表示数据和指令，并且采用存储程序方式。对一个运行中的目标机而言，这种二进制形式的程序，被存储在存储器并且可自动被目标机执行，我们特别称之为可执行程序。可执行程序与目标机联合起来组成一个计算系统。同样的目标机，与不同的可执行程序组成一个一个计算系统，这是计算系统众多的原因。

冯·诺依曼结构除了有一个能随机存取的主存储器外，还有控制单元和算术逻辑单元，合起来称为中央处理单元（central processing unit，CPU）。中央处理单元按照取指令、译码、执行和写回四个典型阶段逐条执行指令。这些指令是存储器中可执行程序的组成部分。另外，还有输入输出设备完成与外界交换数据的工作。

目前除了量子计算机，基于冯·诺依曼结构，已发展出多种多样的硬件机器，比如这样一些计算机，它们有 X86 或 MIPS 等不同的指令系统，或者有不同的总线结构，或者它们的中央处理单元各有所长，等等。因此，编译器的目标机是多种多样的，因为编译程序要与这些硬件机器的细节打交道。总之，冯·诺依曼结构意味着事先编制程序，事先将程序（包含指令和数据）存入主存储器，运行时自动地、连续地从存储器中依次取出指令且执行之。对于大型或复杂程序，手工编制可执行程序几乎是不可能的，所以相应的程序转换工具应运而生，随之编译器的必要性也体现了出来。

程序是用语言表达的问题求解过程，所以常常会说这个程序是用这种语言写成的。如果我们将用某种语言能够写成的所有程序看作为一个集合的话，那么这个集合就是这种语言。从这个意义上来说，用一种语言写成的任意一个程序所能构成的每一个计算系统都需要等价地转换为通用图灵机。这个转换过程可实现为一些编译程序，它们针对冯·诺依曼机器来完成编译。

1.3 编译程序

若将编译概念具体化，首先意指编译程序。编译程序也叫编译器。一般来说，每一个编译程序都是一个复杂的程序，它读入用某种高级语言（源语言）写的程序并将其“翻译”成与之等价的低级语言（目标语言）程序。编译器的输入通常被称为源程序，意味着编译程序是处理程序的程序，而它的输出为目标程序，表明处理的结果还是程序。另外，作为翻译过程的一个重要组成部分，编译器能够向用户报告源程序中存在的错误。编译器的性能指标有时也会被关注，因为它们反映了编译器编译速度的高低和生成目标代码的优劣。

作为编译器及其相关方面的示意，T 形图直观反映编译器的整体外观，如图 1-2 所示。

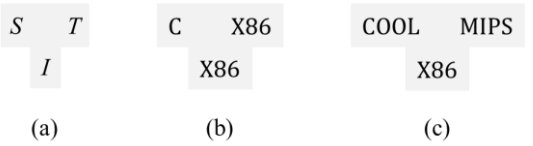


图 1-2 T 形图示意

图 1-2 中单个 T 形图的 3 个端分别写有表示不同含义的符号，都指代语言，分别与编译器输入、输出及自身有关。如图 1-2(a)所示为一个用语言 *I* 写的编译器，它的输入为一个用 *S* 语言写的源程序，而输出为一个用 *T* 语言写的目标程序，依次被称为编译器的实现语言、源语言和目标语言。举例而言，通常使用 PC 机开发程序，常见使用如图 1-2(b)所示的编译器，它的源程序为 C 语言程序，编译成 X86 语言程序，而编译器本身也是 X86 程序，因为它在该 PC 机上运行。图 1-2(c)的例子中源语言为 COOL、目标语言为 MIPS、实现语言为 X86，表示一个 X86 机器上运行的编译器，能够把 COOL 语言程序编译为 MIPS 程序作为输出。

将 T 形图组合起来并赋予序号容易表示编译器的移植过程。如图 1-3 所示，图中编译器(1)和编译器(2)为已知，需要构造编译器(4)。通过一个如下的编译器移植过程来完

成：首先将编译器(1)作为编译器(2)的输入，运行编译器(2)产生输出编译器(3)；然后以编译器(1)作为输入，运行编译器(3)就得到最终的编译器(4)。对于一个给定的编译器而言，它能运行在宿主机上，说明编译器的实现语言就是宿主机的机器语言，因为只有这样，机器才能够直接执行该编译程序的指令。编译器的目标程序运行在目标机上，所以目标机一般就是编译器的宿主机。有些场合下目标机不同于宿主机，比如嵌入式系统由于资源受限，目标代码在宿主机编译生成后下载到目标机执行。这种生成不同于宿主机机器语言写的目标程序的编译器有个特别的名字：交叉编译器。

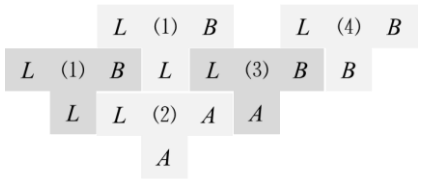


图 1-3　编译器移植示例

为了语言在抽象层次上的灵活性、对众多目标机器的适应性以及在源程序组织结构上的模块化等诸多方面的好处，所述编译程序所做的从源程序到目标程序的转换过程事实上是通过使用编译工具链完成，而不必由单个编译程序完成。尽管功能上可能有些重叠。粗略地说，常见编译工具链包含有编译器、汇编器、链接器、装载器等工具，如图 1-4 所示。

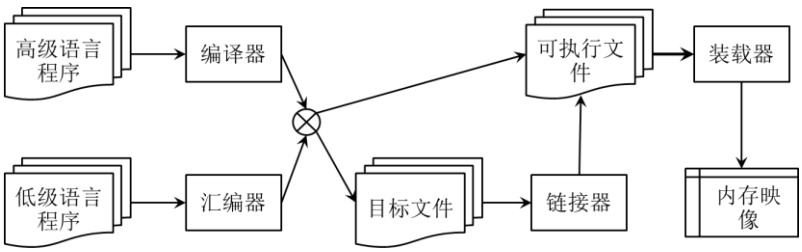


图 1-4　编译工具链

从图 1-4 可见，编译工具链包括一系列能够将源语言程序逐步转换为可执行程序的工具。其中编译器所做的工作有两个选择：一是将高级语言程序翻译为低级语言程序；二是将高级语言程序直接翻译为目标程序或者可执行程序。汇编器的作用是将低级语言程序（代码）翻译为目标程序（代码）或者可执行程序（代码）。链接器将目标程序，可能还有库、其他重定位目标代码文件等，合并在一起得到可执行程序或者其他目标文件。装载器一般属于操作系统的实用程序，负责将可执行程序装入内存以得到运行，形成内存映像。

使用编译工具链能带来几个好处：一是让程序在表达上更接近自然方式；二是可借助工具提供的自动化程序分析、优化和变换功能支持；三是程序开发者通常会借此大大提高他们的产品率，降低出现错误的概率，并且提高所生产的软件的质量。另外，相比于单一工具使用方式，几个工具的组合使用更有吸引力，主要是对模块性、可复用性有贡献。特别地，编译工具链为将程序拆分成一些组成部分，分别以不同方式利用相应的工具进行处理带来了方便。

程序拆分带来协作开发方面的便利，即各部分分别开发然后合并进入编译过程。如某个部分发生改变（添加功能或修改错误）只需对那个部分用工具链再处理，比如重新编译；如果一些部分用某种语言（如 C 语言或其他高级语言）写成，而另一些部分用另一种语言（如汇编语言）写成，就能够使用不同工具分别生成目标文件，然后再将这些目标文件链接成一个完整程序。如果需要复用另一程序的某一部分，也更容易将其提取出来作为这个程序的组成部分。例如通过链接器将常用函数库织入程序中。

考虑到编译程序与源语言和目标语言的相关性，采用“前端”+中间表示+“后端”这种结构是较为常见的框架方案，其中编译器前端部分对源程序进行分析，并生成中间表示形式作为结果，而编译器后端对输入的中间表示形式进行代码优化并生成优化的目标代码作为输出。

有一些典型的编译系统架构，前端支持多种源语言，将源程序翻译为中间代码。也可有一个中间端（属于后端）以完成与语言和目标机都无关的中间代码优化任务，一个代码生成器（属于后端）完成优化的中间代码到目标代码的生成任务。前端与中后端完全分离，便于扩展其他语言前端，取得如下支持：使用为源语言裁剪的抽象语法树；所有前端都从代码优化和代码生成获益；前端生成调试信息包含在中间表示中。中间表示是体积小的、简单的、容易理解的和良定义的。中间端容易扩展标准标量优化、过程间优化等功能。以上这些灵活的框架结构是编译器长期发展的结果，代表性的编译器如 LLVM（low level virtual machine，底层虚拟机）、GCC（GNU compiler collection，GNU 编译器套件）等就采用了这些框架结构。

1.4 编译过程

编译器将源程序翻译为目标程序是一个复杂的过程，但从中分解出相对独立的阶段是可能的，这样有助于建立起对这个复杂过程的理性认识。

一种典型划分方案是将编译过程分成几个阶段，彼此之间有良定义接口，因为从概

念上来说，这些阶段是按照顺序执行的（尽管实际中有某些交错），除了第一个阶段外各阶段之间都可统一为这样的输入输出关系，即将前一个阶段的输出作为后一阶段的输入，通常每一个阶段都对应一个程序模块来处理，作为编译程序的功能组成部分。

对于编译阶段的划分没有严格限制，可以有不同的划分方案，它们中各个阶段间的次序会有微小变化，一些阶段可以合并也可以拆解成更多阶段，或者有些额外的阶段插入其中。或粗或细的阶段划分都是对编译过程的认知途径。

初学阶段，为了简单起见，通过下面的示例给出一种常见的编译过程划分方案。这种划分方案将编译过程划分为词法分析、语法分析、语义分析、代码优化、目标代码生成五个阶段，具有普遍适用性。

编译过程的第一个阶段是词法分析。对于给定的源程序：

```
1    int x;
2    int fact(int n; int a;) {
3        if(n==1)return a else return fact (n−1, n*a,)
4    };
5    x = 123+fact(5, 1,);
6    print x
```

使用转义字符\n 和\t 分别表示换行符、制表符，那么第 2 至 4 行全部可见的源程序符号串为

```
int fact(int n; int a;) {\n\tif (n==1)return a else return fact(n−1, n*a,)\n};
```

经过词法分析，输出词法记号串：

```
(INT, _)(ID, fact)(LPA, _)(INT, _)(ID, n)(SCO, _)(INT, _)(ID, a)(SCO, _)(RPA, _)
(LBR, _)(IF, _)(LPA, _)(ID, n)(ROP, ==)(NUM, 1)(RPA, _)(RETURN, _)(ID, a)
(ELSE, _)(RETURN, _)(ID, fact)(LPA, _)(ID, n)(AOP, _)(NUM, 1)(CMA, _)(ID, n)(AOP, *)
(ID, a)(CMA, _)(RPA, _)(RBR, _)(SCO, _)
```

每个词法记号都表示为一个二元组，有种属和值两部分，其中值部分在必要时可省略。输出的记号串中，INT、IF、ELSE、RETURN 分别是关键字 int、if、else 和 return 的种属；ID 是标识符的种属，对应的值为标识符的符号名；ROP 和 AOP 分别是关系运算符和算术运算符的种属，对应的值为各自运算符；NUM 为整数的种属，对应的值为整数值；其余是各个括号的记号，它们的种属有 LPA、RPA、LBR 和 RBF，各自的值都是与其种属一一对应的，所以在表示上被省略掉。

词法分析的理论基础是形式语言理论。基于形式语言理论，每个种属就是一个正则语言。基于正则语言的判定性质，多语言联合起来逐一识别出一个一个词法单元，这就是词法分析过程。

第二个阶段是语法分析。该过程将第一阶段得到的一个记号串当作输入，针对输入构建出语法树，如果输入串有语法错误则报告之。

对于示例程序，采用本书提供的主文法（见附录 A）进行语法分析。语法分析所构建的带注释语法树如列表 1-1 所示，其中语法树是语法分析的结果，树上的注释是语义分析的结果。

列表 1-1　Fact 程序的带注释语法树示意

1 *P* @table 创建完成; @code:[t6 = 123; ③; t10 = t6+t9; x = t10; PRINT x]

2 ___*Ď* *place*: (x fact)

3 _____*Ď* *place*: (x)

4 _____*Ď* *place*:NIL; 创建空符号表@table

5 _____*ε*

6 _____*D* *palce*: (x); 登记 x 到@table

7 _____*T* *type*:INT

8 _____int

9 _____*d*<x>

10 _____;

11 _____*D* *place*: (fact); fact@code:[①; RETURN a; GOTO l3; LABEL l2; ②; RETRUN t5; LABLE l3]; 登记 fact 到@table

12 _____*T* *type*:INT

13 _____int

14 _____*d*<fact>

15 _____(

16 _____*Ǎ* *place*: (n a)

17 _____*Ǎ* *place*: (n)

18 _____*Ǎ* *place*:NIL; 创建空符号表 fact@table

19 _____*ε*

20 _____*A* *place*: (n); 登记 n 到 fact@table

21 _____*T* *type*:INT

22 _____int

23 _____*d*<n>

24 _____;

25 _____*A*　　*place*: (a); 登记 a 到 fact@table

26 _____*T*　　*type*:INT

27 _____int

28 _____*d*<a>

29 _____;

30 _____)

31 _____{

32 _____*Ď*　　*palce*:NIL

33 _____*ε*

34 _____*Š*　　*code*:同下

35 _____*S*　　*code*:[①; RETURN a; GOTO l3; LABEL l2; ②; RETRUN t5; LABLE l3]

36 _____if

37 _____(

38 _____*B*　　*tc*: (l1); *fc*: (l2); *code*:[t1 = 1; IF n==t1 THEN l1 ELSE l2]①

39 _____*E*　　*place*:n; *code*:[]

40 _____*d*<n>

41 _____*r*<==>

42 _____*E*　　*place*:t1; *code*:[t1 = 1]; 登记 t1 到 fact@table

43 _____*i*<1>

44 _____)

45 _____*S*　　*code*:[RETURN a]

46 _____return

47 _____*E*　　*place*:a; *code*:[]

48 _____*d*<a>

49 _____else

50 _____*S*　　*code*:[②; RETRUN t5]

51 _____return

52 _____*E*　　*place*:t5; *code*:[t2 = 1; t3 = n–t2; t4 = n*a; PAR t4; PAR t3; t5 = CALL fact, 2]②; 登记 t5 到 fact@table

53 _____*d*<fact>

54 _____(

55 _____*Ě* *place*: (t3 t4); *code*: ([t2 = 1; t3 = n−t2] [t4 = n*a])

56 _____*Ě* *place*: (t3); *code*: ([t2 = 1; t3 = n−t2])

57 _____*Ě* *place*:NIL; *code*:NIL

58 _____*ε*

59 _____*E* *place*:t3; *code*:[t2 = 1; t3 = n−t2]; 登记 t3 到 fact@table

60 _____*E* *place*:n; *code*:[]

61 _____*d*<n>

62 _____*o*<−>

63 _____*E* *place*:t2; *code*:[t2 = 1]; 登记 t2 到 fact@table

64 _____*i*<1>

65 _____,

66 _____*E* *place*:t4; *code*:[t4 = n*a]; 登记 t4 到 fact@table

67 _____*E* *place*:n; *code*:[]

68 _____*d*<n>

69 _____*o*<*>

70 _____*E* *place*:a; *code*:[]

71 _____*d*<a>

72 _____,

73 _____)

74 _____}

75 _____;

76 ___*Š* *code*:[t6 = 123; ③; t10 = t6+t9; x = t10; PRINT x]

77 _____*Š* *code*:同下

78 _____*S* *code*:[t6 = 123; ③; t10 = t6+t9; x = t10]

79 _____*d*<x>

80 _____ =

81 _____*E* *place*:t10; *code*:[t6 = 123; ③; t10 = t6+t9]; 登记 t10 到@table

82 _____*E* *place*:t6; *code*:[t6 = 123]; 登记 t6 到@table

83 _____*i*<123>

84 _____+

85 _____*E place*:t9; *code*:[t7=5; t8=1; PAR t8; PAR t7; t9=CALL fact, 2]③; 登记 t9 到@table

86 _____*d*<fact>

87 _____(

88 _____*Ě*　　*place*: (t7 t8); *code*: ([t7 = 5] [t8 = 1])

89 _____*Ě*　　*place*: (t7); *code*: ([t7 = 5])

90 _____*Ě*　　*place*:NIL; *code*:[]

91 _____*ε*

92 _____*E*　　*place*:t7; *code*:[t7 = 5]; 登记 t7 到@table

93 _____*i*<5>

94 _____,

95 _____*E*　*place*:t8; *code*:[t8 = 1]; 登记 t8 到@table

96 _____*i*<1>

97 _____,

98 _____)

99 _____;

100 _____*S*　*code*:[PRINT x]

101 _____print

102 _____*E*　*place*:x; *code*:[]

103 _____*d*<x>

列表 1-1 中的叶子节点为词法记号。词法记号在语法树上的表示有灵活性，以终结符表示，以源程序中的子串表示，或者二者结合起来，在本质上没有区别，因为语法树中的词法记号都是特指。比如第 83 行的 *i*<123>也可写成 *i* 或者 123，当写成 *i* 时从上下文知道是 123，写成 123 时从上下文知道是 *i*。再比如：第 103 行 *d*<x>，写成 *d* 或 x 均可；第 69 行 *o*<*>写成 *o* 或*均可。

列表中内节点为变元，指代语法单元，有表达式 *E*、表达式表 *Ě*、语句 *S*、语句表 *Š*、声明 *D*、声明表 *Ď*、类型指示符 *T*、形参 *A*、形参表 *Ǎ*，还有程序 *P* 等各类语法单元。语法单元及其相互间的关系是通过上下文无关文法定义的，列表中也不例外。该语言的定义参见附录 A 主文法定义。由此可见，语法分析的理论基础是形式语言的一部分内容，将在本书后续章节给予介绍。

在语法分析的基础上，编译过程进行到第三个阶段。这个阶段进行语义分析并生成中间语言代码。语义分析通常结合语法分析一起进行，称为语法制导翻译，这使得语义

分析任务能够依赖于语法规则进行划分。语义分析的主要任务是从程序声明中提取语义信息列至符号表中，每个函数一个符号表。语义分析的另一个任务是将程序语句翻译为中间语言代码，并以函数为单位，将函数体语句表 $\check{S}$ 翻译为函数的中间语言代码，同时也作为这个函数的符号表的一项登记信息。

对 Fact 程序经过语义分析构建出两个符号表，分别是最外层函数（无名函数）的符号表@table 和 fact()函数的符号表 fact@table。这里采用全局名 name@aspect 的形式表示名字 name 的 aspect 方面，例子中具体化为符号表方面，例子中也有代码方面，分别用@code 和 fact@code 表示无名函数代码和 f()函数代码，它们均为中间语言代码。

Fact 程序的中间表示都列在列表 1-2 中，所包含的临时变量登记项都是在中间代码生成时创建的。

列表 1-2　Fact 程序的函数符号表

```
@table: (
    outer:NIL width:32 argc:0 arglist:NIL rtype:INT level:0 code:[t6 = 123; t7 = 5; t8 = 1; PAR t8; PAR t7;
    t9 = CALL fact, 2; t10 = t6+t9; x = t10; PRINT x]
    entry: (name:x type:INT offset:4)
    entry: (name:fact type:FUNC offset:12 mytab:fact@table)
    entry: (name:t6 type:TEMP offset:16) entry: (name:t7 type:TEMP offset:20)
    ... entry: (name:t10 type:TEMP offset:32))

fact@table: (
    outer:@table width:30 argc:2 arglist: (n a) rtype:INT level:1
    code:[t1 = 1; IF n<= t1 THEN l1 ELSE l2; LABLE l1; RETURN a; GOTO l3; LABEL l2; t2 = 1; t3 = n−t2; t4 =
    n*a; PAR t4; PAR t3; t5 = CALL fact, 2; RETURN t5; LABEL l3]
    entry: (name:n type:INT offset:4) entry: (name:a type:INT offset:8)
    entry: (name:t1 type:TEMP offset:12) ... entry: (name:t5 type:TEMP offset:30))
```

中间代码生成过程及结果都标注在语法树上，见列表 1-1。当然最终代码作为符号表 code 域值在编译过程中被记录下来。语义分析的结果形成两个符号表见列表 1-2。这两个符号表都有表头和登记项，采用<域名>:<域值>的表示形式，属于本书所采用的主符号系统（见附录 B）。符号表登记项可有多个，每个登记项还可有自己的子域，仍采用<域名>:<域值>的表示形式。每个函数声明的局部名都登记在自己符号表中，并且各自都有偏移量表示在函数局部区里的存储位置。关于符号表的具体介绍参见本书第 9 章，而主符号系统贯穿于本书全篇。

接下来是代码优化阶段和目标代码生成阶段。由于代码优化可以针对中间语言代码和目标代码分别独立进行，所以关于这两项任务的先后次序可以有灵活性。为此，这样一种多遍迭代的决策是有趣的，一些指令被转换、被优化，经过多次迭代逐步从中间语言程序向可执行程序再向优化的可执行程序转化。

以尾递归优化为例，说明如下。针对中间语言代码进行尾递归优化，对于形参为 n 和 a 的函数 fact()的代码 fact@code 进行优化，将其中的递归调用指令序列 [PAR t4; PAR t3; t5 = CALL fact, 2; RETURN t5]转换为采用控制流转移的形式[n = t3; a = t4; GOTO fact@code]而不再是调用形式，但是@code 中的指令序列[PAR t8; PAR t7; t9 = CALL fact, 2]不变。转变后的 Fact 程序的 fact()函数就不是递归的了，故将此过程称为尾递归消除，是一种优化。优化后的代码节省了大量栈空间，提高了代码执行效率。

语义分析生成的函数代码并不能正确执行，原因是函数调用没有获得支持。活动记录机制是面向过程的语言运行时所依赖的函数调用栈结构，所以在设计好栈帧格式的前提下，函数代码将被变换形成可执行代码。变换后得到的可执行代码采用函数的 label 方面表示。

例如，把@code:[t6 = 123; t7 = 5; t8 = 1; PAR t8; PAR t7; t9 = CALL fact, 2; t10 = t6+t9; x = t10; PRINT x]变换为@label:

```
[sp = sp−4; M[sp] = $ra; sp = sp−@width;          #序言
t6 = 123; t7 = 5; t8 = 1;
sp = sp−4; M[sp] = t8; sp = sp−4; M[sp] = t7;     #构建参数区
sp = sp−4; M[sp] = fp;          #构建访问链
sp = sp−4; M[sp] = fp;          #构建控制链
fp = sp;                        #切换当前栈帧为被调的栈帧
JAL fact@label;                 #控制流转向被调代码入口
t9 = $v0;                       #暂存返回值
fp = M[fp];                     #切换当前栈帧为主调的栈帧
sp = sp+24;                     #释放被调的参数区并将 sp 切换为当前栈帧
t10 = t6+t9; x = t10; PRINT x;
$v0 = R0; GOTO l4;              #返回值寄存器置为 0 并转尾声
LABEL l4; t0 = M[fp−4];         #取出返址
sp = fp;                        #释放局部区，释放返址单元
JR t0]                          #控制流转回主调
```

尾递归优化之后代码转换示意如下，删除线所示代码被删去，下划线所示代码被添加。

[LABEL l0; t1 = 1; IF n = t1 THEN l1 ELSE l2; LABEL l1; RETURN a; GOTO l3; LABEL l2; t2 = 1; t3 = n−t2; t4 = n*a; ~~PAR t4; PAR t3; t5 = CALL fact, 2; RETURN t5~~ n = t3; a = t4; GOTO l0; LABEL l3]

t5 不再有用，从符号表里删去。更新代码及登记项后的符号表如下：

fact@table: (outer:@table width:24 argc:2 arglist: (n a) rtype:INT level:1

code:[LABEL l0; t1 = 1; IF t1<n THEN l1 ELSE l2; LABLE l1; RETURN a; GOTO l3;

LABEL l2; t2 = 1; t3 = n−t2; t4 = n*a; n = t3; a = t4; GOTO l0; LABEL l3]

entry: (name:n type:INT offset:4) entry: (name:a type:INT offset:8)

entry: (name:t1 type:INT offset:12)... entry: (name:t4 type:INT offset:24))

对 fact()函数实施尾递归优化，结果如该符号表所示。继续生成带有运行时的可执行代码如下：

fact@label:

```
[sp = sp−4; M[sp] = $ra; sp = sp−fact@width;          #序言
n = M[fp+8]; a = M[fp+12];                            #参数传递
LABEL l0; t1 = 1; slt t0, t1, n; beq t0, R0, l2;      #指令模板；替换 t1 为
                                                      M[fp−16];
LABEL l1; $v0 = a; GOTO l4;                           #置返回值并转尾声
LABEL l2; t2 = 1; t3 = n−t2; t4 = n*a; n = t3; a = t4; GOTO l0;   #已优化尾递归为循环
LABEL l3;
LABEL l4; t0 = M[fp−4]; sp = fp; JR t0]               #尾声
```

在可执行代码基础上进一步做优化，比如寄存器分配、局部优化、窥孔优化等，最后生成完全的 MIPS 代码。有关 Fact 程序的优化和全部生成 MIPS 代码的内容超出了本书范围。

1.5 形式语言概论

形式语言是用来精确地描述语言（包括人工语言和自然语言）及其结构的手段，通常以重写规则 $\alpha \longrightarrow \beta$ 的形式进行表达，其中 α 与 β 均为符号串。一个初始的符号串通过不同的顺序，不断应用不同的重写规则，可以得到不同的新符号串。

形式语言是应用于计算系统的理论工具，概念上与自然语言可以类比得到相应的名词术语，而其理论体系则形成于 20 世纪 30 年代至 50 年代。形式语言理论直接应用于解决程序设计语言的词法、语法、语义分析等编译前端的问题，也是解决编译验证等问题的可用手段。形式语言理论主要包含有穷自动机理论和形式文法理论。

1.5.1　有穷自动机

在理论计算机科学中，有穷自动机理论是对抽象机器和它们能解决的问题的研究。有穷自动机是许多重要类型的计算系统的有用模型。一些常见的应用不限于被用于构建如下系统模型：

(1) 数字电路设计与性能检查软件。

(2) 典型编译器的词法分析器，比如描述无符号十进制整数：[1-9][0-9]* | 0。

(3) 扫描大量文本来发现单词、短语或其他模式匹配对象，比如：一种社交用户密码(长度 8~16，至少含有一个字母的字母数字串)：(a-zA-Z0-9]*[a-zA-Z][a-zA-Z0- 9]*){8,16}；一种街道地址：([A-Z][a-z]*\s)+(Street | St\. | Avenue | Ave\. | Road | Rd\.)；等等。

(4) 所有类型的只有有穷多个不同状态的系统的验证软件，比如通信协议或安全交换信息的协议；

(5) 一些物理系统的建模与仿真，比如自动售货机、购物系统等。

上述及许多类似系统有其共性，表现在其始终处于有穷多个“状态”之中。借用状态的目的是记住系统历史的有关部分。由于只有有穷多个状态，在一般情况下不可能记住整个生命期历史，所以必须仔细地设计系统，以记住重要的而忘记不重要的生命期节点。限制自动机为有穷个状态的好处是用固定资源就能实现系统。例如，可用一定数量的门电路实现计算机的部件，或者用简单的策略实现程序，这种程序只需要查看有限的信息就能作出决定。

例 1.2　构建两相开关的有穷自动机模型。

最简单的有穷自动机大概是两相开关了，如图 1-5 所示。

图 1-5　一个两相开关实物

一个两相开关有一个按键，按键每按下一次，开关是开还是关将切换一次。建模结果如图 1-6 所示。

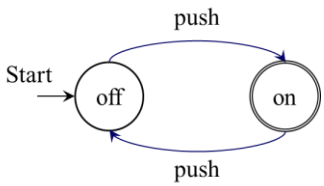


图 1-6　两相开关的模型

该模型尽管很简单，但仍然把两相开关看作一个物理系统。这个物理系统在任何时刻不是开着就是关着，所以“开”和“关”可作为它的两个状态，从而能够聚焦它在工作过程中的当前状态。作为物理系统的模型，不能不涉及系统边界。这个物理系统边界上有一个交互行为，就是开关按钮被按下一次。边界上发生交互事件引起物理系统的当前状态发生变化，一次交互行为要么让开关从开着转换为关着，要么反过来。描述这样的物理系统是为了实现感兴趣的工作过程，即从它的初始化状态开始，连续按动开关按键并最终使得一种接受状态成为当前状态。比如，设定初始状态和接受状态都为关着，就可反映出每天晚上书房照明灯的合理使用。

有穷自动机适合作为这样的物理系统的模型，如对两相开关建模，结果如图 1-6 所示。将状态开着和关着分别命名为“on”和“off”，将边界交互行为命名为“push”，初始状态和接受状态分别设定为 off 和 on，表明对书房灯到半夜还亮着之类的状态感兴趣。用自动机读入一个符号 push 来描述边界上的一次交互事件，那么连续的交互事件描述为一个由符号 push 组成的符号串。

读入符号 push 引起当前状态发生变化，用状态转换规则来表达，如图 1-6 所示，状态转移弧表示自动机由箭弧射出的那个状态转移到箭弧射入的那个状态，也称为状态转换规则。

“(off, push)→on”是一条状态转换规则，如果当前状态为 off，读入符号 push，那么消耗掉读入符号并且自动机的当前状态转换为 on。

“(on, push)→off”是另一条规则，如果当前状态为 on，读入符号 push，那么消耗掉读入符号并且自动机的当前状态转换为 off。

把自动机的一次活动中读入的符号依次连接起来形成读入符号的序列，称为输入串，回看这次活动中的自动机，一边消耗一个输入符号一边更新一次当前状态，一直到输入串消耗完为止，然后根据当前状态为 on 得出输入串为自动机接受的结论，否则拒绝。一般地，这个自动机接受奇数长度的输入串，而拒绝偶数长度的，表明建模

结果正确。

从所举示例不难看到，有穷自动机是有限状态机器，它在运行时通过状态之间的转移来消耗掉输入串，从而得出是否接受该输入串的结论。一个自动机所接受的符号串全集被称为这个自动机的“语言”。本质上有穷自动机是图灵机模型的特例，其概念和术语都忠实地出自后者。从简单到复杂，横向比较几类典型自动机，其结果如下。

有穷自动机：有少量存储的装置，用于对非常简单的事情建模。

下推自动机：有无限量存储并以受限方式访问的装置。

时间界限图灵机：存储器无限大但运行时间有界，这是以合理速度运行的计算机。

图灵机：有无限量存储的装置，这是实际的计算机。

如上所述，图灵机是实际的计算机模型，参照图灵机，就得到自动机模型的特点：有有穷个状态和或多或少的存储。它们都是定义语言的抽象机器，称为语言识别器。

1.5.2　形式文法

在 20 世纪四五十年代，语言学家乔姆斯基（A. N. Chomsky）开始研究形式文法，此后形式文法逐渐发展成为定义语言的另一种手段，也称为语言识别器。形式文法描述形式语言的基本想法是，从一个特殊的初始符号出发，不断地应用一些产生式规则，从而生成一个由符号串组成的集合。形式文法应用于典型编译器的语法分析器和语义分析器，是基于文法的通用问题求解方法的核心。

产生式规则指定了某些符号组合如何被另外一些符号组合替换。例如，有下列两个个产生式：

$S \longrightarrow aSb$

$S \longrightarrow ba$

那么，有如下始于 S 的推导过程，可推导出仅由 a 和 b 组成的符号串，这正是需要从文法得到的。在推导的每一步，符号串中 S 的一次出现被用规则右部替换产生下一步的符号串，如此反复进行直至不能推导为止。使用符号“⇒”表示一步推导，那么，$S \Rightarrow ba$，$S \Rightarrow aSb \Rightarrow abab$，$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aababb$，… 。结论是：所有从 S 推导出的不含 S 的符号串都是可知的。

在计算机科学中，形式语言是字母表上一些有限长符号串组成的集合。形式文法是定义这种语言的一种手段。乔姆斯基认为，“语言是句子的集合，句子是有穷长度的且由某个有穷字母表中的符号构成”，还认为“文法可以被看作一个装置，它恰好枚举一种语言的句子”。

对于确定的有穷自动机（deterministic finite automata，DFA）和非确定有穷自动机（nondeterministic finite automata，NFA）、下推自动机（push-down automata，PDA）、图灵机等，它们都分别有等价的形式文法和形式语言，涉及的术语如表 1-1 所示。正则表达式（regular expression，RE）、正则文法（regular grammar，RG）是III型文法，与 DFA、NFA 分别都是正则语言（regular language，RL）的识别器。上下文无关文法（context-free grammar，CFG）是II型文法，与 PDA 都是上下文无关语言（context-free language，CFL）识别器。

表 1-1 形式语言与自动机术语

自动机	形式文法	形式语言
DFA、NFA	RE、RG	RL
PDA	CFG	CFL
TM	0 型文法	递归可枚举语言

表 1-1 的不同列各有其独有特点：自动机便于执行；形式文法便于推导、推理；形式语言便于枚举。乔姆斯基于 1956 年提出一个形式语言体系，确定了各型语言的真包含关系，被称为乔姆斯基体系，如图 1-7 所示。

乔姆斯基体系至今仍然作为计算机科学中刻画形式文法表达能力的一个分类谱系。

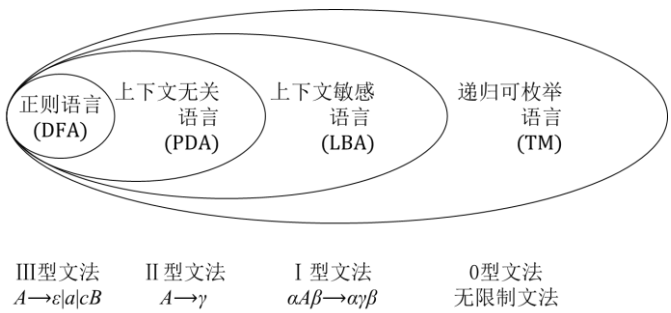


图 1-7 乔姆斯基体系

1.6 中心概念及系统

本节介绍贯穿本书的最重要概念的定义及对基础系统的描述，这些概念包括符号、

字母表、符号串、语言和问题，而基础系统是主文法和主符号系统。

1.6.1　中心概念

定义 1.1　符号。符号是指代事物用到的字母、数字、字符、标记或它们的组合体等。所谓指代意味着计算系统中出现的符号可以被引用，以知道它的指代物。事实上，对符号的来源范围无从限定，倒是它们的指代物可以是具体的事物，也可以是抽象的概念，都是认知域或计算系统的组成部分。当一个被指代的事物明确后，符号便作为这个事物的名字而存在于计算系统中。符号被当作名字是天经地义的，因为它必然成为指代物的名字，除非那些指代物不可知。因此，对合法符号的唯一判定依据就是能行就行，是事实标准，如果有的话。但是，基本点是明确的，从组成上，符号是原子的，符号之间没有组合关系，尽管对符号的概念描述里有“组合”二字，但那仅仅是与起名字相关，而不是名字与名字相关，因为起名字不属于计算系统范畴，正如写程序不属于写好的那个程序的范畴。

当我们关注一个计算系统的时候，所出现的符号是有穷个，所以对它们的名字构成可以要求有更好的一致性，以降低表示上的复杂性。对它们的指代物可以划分类别并将其反映在它们的命名构思上以提高可读性。当同时关注多个相互关联的计算系统的时候，一个计算系统的符号与另一个计算系统的符号没有相关性，除非它们之间有交互协议来允许共享某些符号或建立相互等价关系。

从符号定义可知符号是多模态的，但除非有排他性，文本名更符合形式表达上的简化原则，就是越简单越好，越便捷越好，越易读易用越好。如此看来，符号是设计的结果。比如，源程序的每一个字符都是一个符号，这种情况属于事实。而在设计主文法时，就可以做到让每个变元由一个拉丁字母作为名字，事实表明，这样做很符合形式上的简化原则。从编译过程看到，称源程序的各个符号都是关于词法分析这个计算系统的，当关注于语法分析这个计算系统时，它们什么都不是。

定义 1.2　字母表。字母表是感兴趣的符号的非空有穷集合。一个认知域或计算系统中的符号都是感兴趣的符号，也可以是其他感兴趣的符号，比如非空有穷的事实集合中的每个元素等。字母表习惯上写为希腊字母 Σ，也可写为其他字母。常见字母表有：

二进制数字集合：$\Sigma_{\mathrm{B}}=\{0,1\}$。

十进制数字集合：$\Sigma_{\mathrm{D}}=\{0,1,2,3,4,5,6,7,8,9\}$。

所有小写拉丁字母集合：$\Sigma_{a}=\{a,b,\cdots,z\}$。

所有 ASCII 字符的集合。

空白字符的集合：$S=\{\backslash t, \backslash n, \backslash r, \backslash 0, \backslash s\}$。

字母加特殊符号\$构成的集合：$\Sigma_{a\$}=\Sigma_a \cup \{\$\}$。

字母表的设计应该是有目的的，应该达到构建一个论域的要求，或者说基于它构建成功一个论域。计算上，字母表的设计目的以基于此成功构建一个计算系统为必要条件。就像 Σ_{B} 可能用于构建二进制系统，S 可能用于对源程序中的分隔进行处理等。

定义 1.3　符号串。符号串是构建在字母表上的最基础的结构，一个符号串是由字母表中的符号组成的长度有穷的序列，是相对于符号表而言的，所以也称为字母表上的符号串，在上下文清楚的情况下，简称为符号串。符号串的结构特征在于它是一个有穷长度的序列，根据序列元素可枚举、可索引的性质，符号串每个元素都是一个符号，每个元素都有一个前驱（第一个除外）和后继（最后一个除外），符号串的无前驱元素的索引一般设为 1，无后继元素的索引值为这个符号串的长度。例如：

$axaz$ 是 Σ_a 上的符号串；

999 是 Σ_{D} 上的一个符号串；

0011011 是 Σ_{B} 上的一个符号串；

$ab\$ba$ 是 $\Sigma_{a\$}$上的符号串；等等。

符号串还可简称为串。符号串的长度是指序列中符号出现的次数，例如以上符号串的长度依次为：4、3、7 和 5。字母表 Σ 上的符号串 w，它的长度记为$|w|$，可以是 0 到任意整数，但不是无穷大。对于长度为 0 的符号串特别记为 ε，它不含任何符号，所以它与字母表事实上没有关系，但它是任何字母表上的符号串。一个符号串 w 里的符号用基于索引来引用，对于 $1\leqslant i\leqslant|w|$，w 的第 i 个元素写为 w_i，$w_i\in\Sigma$。如果 $w_i = w_j$ 也是可能的，对于 $i\neq j$，也就是说同一个符号在符号串中出现多次，所以说符号串的一个元素是某个符号的一次出现。

符号串的序列特征还导致它有前缀部分和后缀部分，如果需要特别关注的话。使用函数 $\pi(w,i)$返回符号串 w 的 i 前缀，就是长度为 i 的前缀。$\pi(10111,2)=10$，$\pi(E{+}T,1)=E$，$\pi(10111,0)=\varepsilon$，$\pi(E{+}T,3)=E{+}T$，$\pi(E{+}T,4)=\perp$。习惯上为了完备性，一个特殊符号⊥表示出错。缺省第二个参数的 $\pi()$函数将返回一个包含所有前缀的集合（可索引集合），比如 $\pi(E{+}T)=\{\varepsilon,E,E{+},E{+}T\}$。索引为长度递增次序。

使用函数 $\tau(w,i)$返回符号串 w 的 i 后缀，就是长度为 i 的后缀。$\tau(10111,4)=0111$，$\tau(E{+}T,1)=T$，$\tau(10111,0)=\varepsilon$，$\tau(E{+}T,3)=E{+}T$，$\tau(E{+}T,4)=\perp$。缺省第二个参数的 $\tau()$函数将返回一个包含所有后缀的集合（可索引集合），比如 $\tau(E{+}T)=\{\varepsilon,T,{+}T,E{+}T\}$。

索引为长度递增次序。

字母表 Σ 上的符号串全集是任意由 Σ 中的符号组成的有穷长度的符号串的集合，采用 Σ^*来表示。比如 Σ_B^*是由 0 和 1 组成的长度有穷的所有符号串的集合。注意到 ε 是 Σ^* 的一个元素，可以写成 $\varepsilon\in\Sigma^*$。Σ 上任意符号串都是 Σ^*的一个元素。长度为 1 的符号串在表示上与组成它的符号没有区别，例如，9 是字母表 Σ_D 上的一个符号串，也是 Σ_D^*的元素。虽然形式上不能区分，但在上下文清楚的情况下，能知道前者是一个符号串而后者是一个符号。再如，对于 $w=xa=by$，有 $\pi(w,1)=b$，$\tau(w,1)=a$，在形式上 xa 隐含着 a 是一个长度为 1 的符号串，因为 x 是符号串（命名习惯），而单独的 a 是一个符号，$a\in\Sigma$；类似地，由 $by=w$ 可知，b 为符号串，只是在写法上与符号 b 没有区别，但含义依照上下文都是确定的。

定义 1.4　语言。简单来说，符号串集合就是语言。这个解释让语言不再神秘。具体地，由字母表 Σ 上的符号串构成的集合就是 Σ 上的语言，也就是说，Σ 上的语言是 Σ^* 的一个子集。如此就得到语言的一个定义如下：

如果 Σ 是字母表，且 $L\subseteq\Sigma^*$，则 L 是 Σ 上的语言。

自然语言可以看作已知字母表上的串集合，这些串就是这个自然语言的所有句子。程序设计语言可以看作 ASCII 字符集上所有合法的程序所构成的集合，其中每个程序是 ASCII 字符集上的一个符号串。当然，以上这两种语言都是难以枚举的，但是像 Σ_a 上的关键字集合、Σ_D 上的无符号整数构成的集合等都是容易枚举的语言。前者在高级语言被设计时确定下来，像 if、while 等都是，数目不大，容易罗列；后者虽然数量众多，好在有语言识别器擅长枚举它们，将于第 2 章和第 3 章介绍。

还可以认识到有符号整数是 $\Sigma_D\cup\{+,-\}$上的语言。实数是 $\Sigma_D\cup\{+,-,.\}$上的语言等等。

$L_1=\{w\in\Sigma_a^*\mid w$ 含有子串 $to\}$，是 Σ_a 上的含有子串 to 的所有串。

$L_2=\{x\in\Sigma_D^*\mid x$ 被 7 整除$\}$是字母表 Σ_D 上语言。

$L_3=\{s\$s^R\mid s\in\Sigma_a^*\}$是字母表 $\Sigma_{a\$}$上的语言。注意到上标 R 是求逆运算。

一些特殊的语言也是重要的。比如：空集 $\varnothing$ 是任意字母表上的语言，是空语言，该语言什么也没有，是一种极端语言；对任意字母表 Σ，Σ^*是语言，是另一极端语言；对任意字母表 Σ，$\{\varepsilon\}$是 Σ 上的语言，是单元素语言、原子语言；等等。对于 Σ 上的语言 L，也可说是 Σ'上的语言，其中 Σ'是 Σ 的真超集。然而，遵守最简原则，更倾向于将 L 默认为 Σ 上的语言。

定义 1.5　问题。在形式语言中，一个问题就是判定一个给定的串是否属于某语言

的提问。将要看到，任何在口头上称为“问题”的东西，竟然都可以表示成语言的成员性。更准确地说，如果 Σ 是字母表，L 是 Σ 上的语言，则问题就是：

给定 Σ^*中的一个串 w，判定 w 是否属于 L。

以下都是判定性的问题：

给定词 w，该词包含 *to* 子串吗？该问题描述为判定 $w \in L_1$ 为“是”还是为“否”。

给定整数 n，它能被 7 整除吗？该问题描述为判定 $n \in L_2$ 为“是”还是为“否”。

给定 Σ_a 上的符号串 s 和 t，它们相同吗？这个问题需要等价地转换为：s 与 t 的逆连接起来是否为回文？回文是这样的符号串：序列反序仍与原序列相同。这样，该问题被描述为判定 $s\$t^R \in L_3$ 为“是”还是为“否”。

理论上，所有问题都可以是或者可以被等价转化为判定性的问题，如查找、统计等任何问题都可以等价转化为判定性问题。对于判定性问题，答案只有“是”或“否”。

1.6.2 主文法与主符号系统

主文法是个代数系统，作为上下文无关语言（包括程序设计语言）的识别器。

主文法对于变元采用单个拉丁字母(包括扩展拉丁字母)表示。终结符除了保留字、字面常量和运算符采用原名外，都表示为单个小写拉丁字母。

主文法呈现为一个核心态，目的是用于识别命令式程序设计语言的主要语言构造，类似于 C 语言和 PASCAL 语言并去掉琐碎的、有重复的、容易扩展的部分，以降低规模并突出编译重点。

主文法核心态容易扩展用于展示有价值的编译理论与方法，不难产生多套文法变体并与语言在风格上建立对应，体现出其中蕴含的创造性思维痕迹。

主符号系统是符号系统的一种，专门针对编译任务构建而成，因此能够胜任编译理论的应用和编译过程的描述，并容易被转换为编译程序。

主符号系统遵守一致性的命名惯例。

主符号系统的基本结构借鉴 Lisp 语言的表结构，操作借鉴于 Lisp 语言的函数和宏，但不限于这些。

主符号系统采用了集合论代数和序列的概念、计算机算法描述以及独特的符号表抽象、代码抽象、面向方面的全局名等。

主符号系统的关注点在于“所以然”方面，没有追求计算的效率，因为计算的效率能够由编译优化技术来提升。

1.7　习题

习题 1.1　作为编译器的目标语言，汇编语言与机器语言相比有什么优势和劣势？以 MIPS 为例说明。

习题 1.2　设计+2 图灵机的状态转移规则，对纸带左端输入串+2，结果放在纸带上结束。

习题 1.3　图 1-8 中是一个滚木雷石玩具，在 A 或 B 垛口处扔下一个木球，机关 x_1、x_2 和 x_3 让木球落向左方或右方，每当一个木球遇到一个机关时，就引起这个机关在木球通过之后改变方向，所以下一个木球会走相反的暗道。

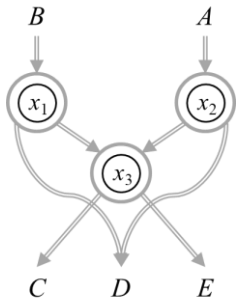


图 1-8　滚木雷石玩具

(1) 用有穷自动机为这个玩具建模。设输入为从垛口 A 或 B 扔进一个木球。设接受对应于木球从 D 或 E 垛口出来，不接受则表示木球从 C 垛口出来。

(2) 用自然语言描述这个自动机的语言。

习题 1.4　试对口香糖球机建模。如图 1-9 所示，口香糖球机中有口香糖球，有投币口可投入十分或五分硬币，一次投入一个硬币，当投入足够硬币同时按下释放键，那么就会吐出口香糖球若干个以及找回的钱。设一个口香糖球按 15 分硬币计价。

图 1-9　口香糖球机示意

(1) 用有穷自动机为这个玩具建模，要求符号规范、术语规范，与图灵机模型有一致性；

(2) 若希望一次成功购买到 3 个口香糖，试问：可能的输入串有哪些？

习题 1.5　分别写出下列语言的字母表并依次判断符号串 ε、123.、+5e6 是不是那个字母表上的符号串。

(1) 无符号八进制定点数；

(2) 有符号十进制定点数；

(3) 无符号十进制实数。

习题 1.6　已知变量 x 的值为符号串+12.34e5，分别写出 $\pi(x)$，$\pi(x,0)$，$\pi(x,5)$，$\pi(x,|x|)$；$\tau(x)$，$\tau(x,0)$，$\tau(x,5)$，$\tau(x,|x|)$。

习题 1.7　写出字母表$\{+,0,1\}$上同时满足如下条件的语言 B：

(1) 串的长度不超过 4；

(2) 串除以 5 余数为 0；

(3) 同时满足(1)和(2)的串一定属于 B。

第 2 章　有穷自动机

本章介绍分为确定有穷自动机（DFA）和非确定有穷自动机（NFA），统称为有穷自动机（finite automata，FA），。它们都是语言识别器，能够识别正则语言。正则语言的另外一种识别器，即正则表达式，将在第 3 章介绍。这些语言识别器都适合应用于编译的词法分析。本章还介绍正则语言识别器的等价性质，从而满足设计更优化词法分析器的需要。

回顾第 1 章中对于形式语言的概述，相对于图灵机，FA 是十分简单的计算模型。如图 2-1 所示，DFA 有一组状态及其状态转移规则，从初始状态开始，通过依次读入输入串的符号，实现状态转移控制，并根据完成时的状态得出判定结论：接受或拒绝。本章首先构建一个简单购物系统的有穷自动机模型模拟购物过程、验证无交易漏洞，展示 FA 的有用性。

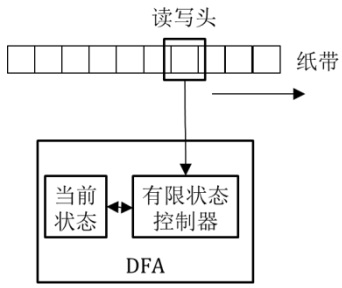


图 2-1　DFA 计算模型

FA 与图灵机的相似性表现在都是有穷个运行状态、用状态转移规则进行计算，以及对位于纸带上的输入串开始计算。

有穷自动机十分简单，是因为严格受限的读写头只能从左往右逐一读入输入串的符号，读完即控制结束。由于这个原因，谈论有穷自动机通常不提及纸带及读写头，只需要给出输入串，并从输入串左端往右端逐一符号地读入，伴随自动机运转过程，直到剩余串为空结束。

不同自动机的最大区别在于状态转移规则。这类规则在形式上用一个右箭头表示转移，

如下所示：

(状态 p，输入符号 c) → 状态 q

解释为自动机状态寄存器值为状态 p，且当前输入符号为 c，那么自动机发生状态转移，将状态寄存器内容更改为状态 q 并且消耗掉当前输入符号 c。换言之，处在“当前状态”的自动机遇到读入符号“当前输入符号”时发生状态转移，自动机的状态转移为“转移状态”。这样的转移规则有两种选择：一是是否允许有两个规则的左部相同，二是当前输入符号是否允许为 ε，意思是这条转移规则不需要消耗输入符号。基于此，将有穷自动机分为三种类型，如表 2-1 所示。

表 2-1 有穷自动机分类

FA	选择一	选择二
DFA	不允许	无 ε 转移
NFA	允许	无 ε 转移
ε-NFA	允许	有 ε 转移

本章中先通过一个简单购物系统的计算模型来介绍 DFA，然后从表 2-1 中“选择一”的角度扩展到 NFA，接下来从“选择二”的角度扩展到 ε-NFA（带 ε 转移弧的 NFA），最后我们看到，这三种类型在定义正则语言能力上是等价的。

从 2.2 节开始，介绍 DFA 定义及表示、如何判定输入串为接受还是拒绝、扩展转移函数以表示连续转移并用于表示 DFA 的语言等。接下来，类似的叙述方式用于 NFA 和 ε-NFA 的介绍，并引入 ε 闭包和 ε 闭集的概念，以及 NFA 转换为 DFA 的算法等。

2.1 一个简单购物系统的 DFA 模型

本节通过一个现实世界的问题建模的例子，来说明有穷自动机在解决这些问题时能起到重要作用。这个例子研究一种简单购物系统：用户选择商家的货品，基于银行签发的电子货币用户给商家付款并收到商家发来的货品，与此同时商家也获得与用户付款等值的电子货币。

这个购物系统建立在银行和商家诚信的基础上，以大大简化系统复杂度。银行的诚信表现在能够正确辨认电子货币并做等值兑换。商家的诚信表现在收到货款即保证用户收到所发送的货品。

一个简单的购物场景是：顾客通过网络订购商品并付款，商店将付款电子货币发

送给银行，银行对收到的电子货币进行认证并签发给商店等值电子货币，商店送货给顾客。此场景显示出，最后顾客收到所购买的货品，而商店则收到款项，一次购物成功得以完成。

分别对顾客、商店、银行进行 DFA 建模，确定出系统边界交互行为如图 2-2 所示。

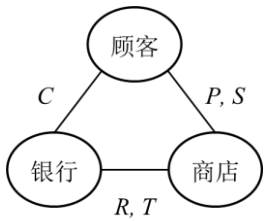


图 2-2　子系统边界

P（付款）：顾客把电子货币发给商店，商店收到。

S（送货）：商店送货给顾客，顾客收到。

C（取消）：顾客让银行取消电子货币，银行取消。

R（兑换）：商店将电子货币发给银行，银行收到并予以认证。

T（转账）：银行签发给商店电子货币，商店收到。

分别构建客户、商店、银行的自动机模型如图 2-3 所示。

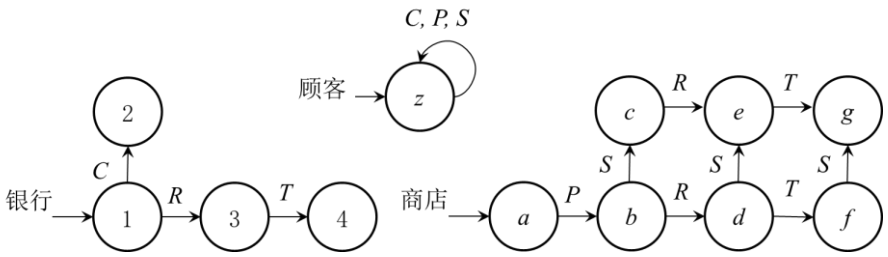


图 2-3　客户、商店、银行的自动机模型

这样分别构建的模型作为购物系统模型存在明显漏洞，如用户使用虚假电子货币、重复使用同一个电子货币、付款后又取消购物等，对商家而言，可能面临货品已发送却收不到货款的情况。应考虑到各自运行时它们之间的协同关系，以避免这个购物系统存在各种漏洞发生。

为了对图 2-3 进行改进，统一环境，各自动机对于原无关交互也要参与，只是仍然转移到自身即可，等价于不作出反应。银行和商店的改进模型如图 2-4 所示。

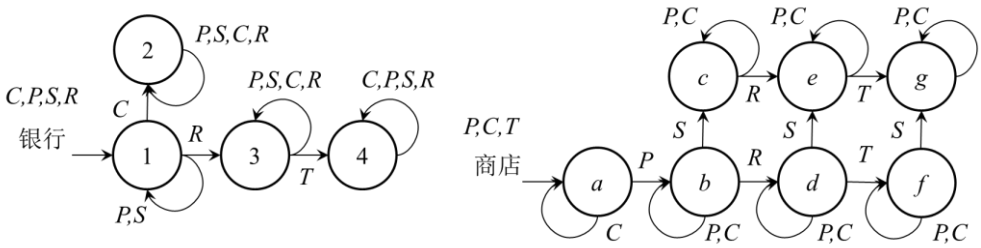


图 2-4　统一环境下银行与商店自动机模型

构建乘积自动机用于验证该设计是否存在上面分析过的漏洞，乘积自动机将在随后章节介绍。

银行、商店乘积自动机如图 2-5 所示，其中为了简明起见，去掉了从初始状态不可达的状态。图中状态 4g 为接受状态，可看到，在从初始状态到接受状态的路径上，保证没有漏洞发生，原因如下所述。

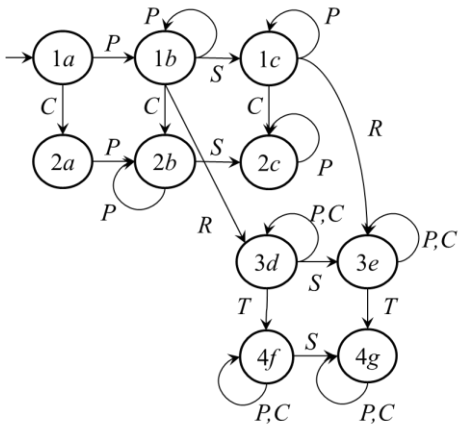


图 2-5　银行、商店乘积自动机（简化型）

商店送了货品却永远收不到货款，商店在 c、e、g 状态，但商店没有在 T 上的转移，从状态 1a 到状态 4g 的路径上都有一个 T 转移和一个 S 转移，说明商家收到货款，用户收到货品。

如果用户付款后又取消，可能商店收到货款而银行收到取消，导致商店依照诚信送了货而货款由于被银行取消（状态 2）而收不到。这种情况下系统要么处于 2a、2b 或 2c 状态，而从此不能达到接受状态，要么在购物成功的路径上。

2.2　确定型有穷自动机

在有穷自动机中，确定型有穷自动机是最为基本的一种。之所以这么说，是因为它既是所有其他有穷自动机在表示上的特例又是它们在功能上的等价物。所以，它往往被作为构建有穷自动机模型的终极目标。它的确定性表现在转移规则方面，形如$(p, a)\rightarrow q$的转移规则分为左部(p, a)和右部q，其中p和q为状态，a为输入符号，那么没有左部分相同的规则，也没有与输入符号无关的规则，这对于程序实现来说是幸运的。

2.2.1　DFA 定义及表示

定义 2.1　一个确定型有穷自动机 DFA A 是一个五元组 $(Q, \Sigma, \upsilon, q_0, F)$，其中：

Q 是有穷个状态的集合；

Σ 是字母表也是输入符号的有穷集合，$Q \cap \Sigma = \varnothing$；

υ: $Q \times \Sigma \rightarrow Q$ 是转移函数，$\upsilon(p, a) = q$ 表示 DFA 的当前状态为 p，当前读入符号为 a 时，DFA 发生状态转移，消耗掉 a，读写头右移一格，当前状态转变为状态 q；

q_0 是初始状态，$q_0 \in Q$；

F 是接受状态集合，$F \subseteq Q$。

在这个定义中，名称 A 指代该 DFA，写成 $A = (Q, \Sigma, \upsilon, q_0, F)$，这是为了便于引用。当然也可以没有名称，写成 DFA $(Q, \Sigma, \upsilon, q_0, F)$，隐含仅对引用模型元素感兴趣。转移函数 υ 满足约束：

$$\forall q \in Q \cdot \forall a \in \Sigma \cdot \upsilon(q, a) \in Q$$

由于 Q 和 Σ 都是有穷集合，所以 υ 可以被写成集合表示。注意到有穷自动机模型中，集合 υ 对应于状态转移规则集合，有时为了特别强调而写成 υ_D，也有把 υ_D 写成算法、程序的情形。

根据定义 2.1，字母表 Σ 被作为 DFA 输入符号的全集，表明 DFA 的输入串都是 Σ 上的符号串。每个 DFA 恰有一个初始状态，而有 0 到多个接受状态，且接受状态集 F 是 Q 的子集，即 $F \subseteq Q$。如果 F 为空集则表明不存在为该 DFA 所接受的输入串。如果初始状态也是接受状态，表明 DFA 接受空串 ε。关于如何判定一个输入串是被接受还是被拒绝将在 2.2.2 节中给出。

例 2.1　一个 DFA M 的代数表示。

已知 DFA $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \upsilon, q_0, \{q_1\})$，其中：

$$\upsilon(q_0, 0) = q_0$$

$$\upsilon(q_0, 1) = q_1$$

$$\upsilon(q_1, 0) = q_2$$

$$\upsilon(q_1, 1) = q_1$$

$$\upsilon(q_2, 0) = q_2$$

$$\upsilon(q_2, 1) = q_2$$

M 有三个状态 q_0, q_1, q_2，初始状态为 q_0，接受状态为 q_1。M 的字母表只有 0 和 1 两个元素，而转移函数为 υ，具体已经列出。

写出 υ 的集合表示就得到 M 的代数表示，具体是

$$\upsilon = \{((q_0, 0), q_0), ((q_0, 1), q_1), ((q_1, 0), q_2), ((q_1, 1), q_1), ((q_2, 0), q_2), ((q_2, 1), q_2)\}$$

一般来说，DFA 的代数表示便于在推理中使用。除了推理以外，还需要有直观的表示和便于程序实现的表示形式。幸运的是，状态转换图和状态转移表就是相应的表示形式。

定义 2.2 DFA $A = (Q, \Sigma, \upsilon, q_0, F)$的状态转换图是一个带权有向图$\mathcal{G}(A) = (Q, E, \Sigma)$，其中 Q 作为顶点集合，Σ 作为权集合，边的集合 $E = \{(p, q, a) \mid p, q \in Q, a \in \Sigma, q = \upsilon(p, a)\}$。有时将有向边$(p, q, a) \in E$ 解释为 p 顶点射出的、标记为 a 的、射入顶点 q 的一条箭弧，p 是这个箭弧的射出端，q 是射入端。更为简洁地，可以将边$(p, q, a) \in E$ 说成是 p 到 q 的 a 弧。图示$\mathcal{G}(A)$的顶点为圆圈表示，接受状态用双圈表示，在初始状态上可加短箭头示意，在上下文清楚的情况下短箭头可省略。

对于例 2.1 的 DFA M，它的状态转换图表示如图 2-6 所示。

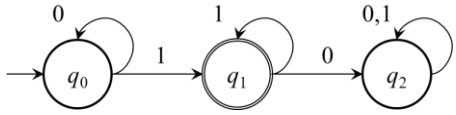


图 2-6 DFA M 的状态转换图表示

容易得出 M 的代数表示和状态图表示$\mathcal{G}(M)$存在一一对应关系：

M 的 Q 的每一个状态对应$\mathcal{G}(M)$中一个顶点，反之亦然；

对于 $q \in Q$ 和 $a \in \Sigma$，如果有 $\upsilon(q, a) = p, p \in Q$，则$\mathcal{G}(M)$有从 p 到 q 的 a 弧，反之亦然；

M 的初始状态 q_0，在$\mathcal{G}(M)$中有一个短箭头指向（或默认下标最小者），反之亦然；

对于 M 的 F 中的每一个状态，在$\mathcal{G}(M)$中都图示为双圈顶点，反之亦然。

因此，这两种表示对于 M 来说是等价的。状态转移图简称为转移图。可以证明，对于任意 DFA，它的代数和转移图在表示上是等价的，证明过程在此省略。

定义 2.3 DFA $A = (Q, \Sigma, \upsilon, q_0, F)$的状态转移表为一个 $n \times m$ 表格$\mathbb{T}$，其中 $n = |Q| + 1$ 而

$m=|\Sigma|+1$，符号$|S|$表示集合 S 的元素个数。对 Q 和 Σ 分别建立索引，使得$\{Q[i] \mid 1\leqslant i\leqslant n\}=Q$ 且$\{\Sigma[i] \mid 1\leqslant i\leqslant m\}=\Sigma$；从而对于 $1\leqslant i\leqslant n$，令$\mathbb{T}[i,0]=Q[i]$，且对于 $1\leqslant j\leqslant m$，令$\mathbb{T}[0,j]=\Sigma[j]$，则对于任意 $1\leqslant i\leqslant n$ 和 $1\leqslant j\leqslant m$，令$\mathbb{T}[i,j]=\upsilon(\mathbb{T}[i,0],\mathbb{T}[0,j])$。此外，习惯上令$\mathbb{T}[1,0]$为初始状态 q_0 且带有短箭头标记，同时对于任意 $1\leqslant i\leqslant n$，若$\mathbb{T}[i,0]\in F$，则它的元素带有星号标记。

对于例 2.1 的 M，它的状态转移表如表 2-2 所示。

表 2-2　DFA M 的转移表$\mathbb{T}$

	0	1
$\rightarrow q_0$	q_0	q_1
$^*q_1$	q_2	q_1
q_2	q_2	q_2

容易找到 DFA $(Q,\Sigma,\upsilon,q_0,F)$ 的代数表示和转移表表示的一一对应关系。

首先恢复索引表，即若$\mathbb{T}[i,0]=q\in Q$，那么 $\mathrm{ind}(q)=i$，其中 $1\leqslant i\leqslant|Q|$；同时，若$(\mathbb{T}[j,0]=a)\in\Sigma$，那么 $\mathrm{ind}(a)=j$，其中 $1\leqslant j\leqslant|\Sigma|$。这里 $\mathrm{ind}(x)$返回 x 的索引值，比如表 2-2 中 $\mathrm{ind}(q_0)=1$，$\mathrm{ind}(q_1)=2$，$\mathrm{ind}(q_2)=3$，$\mathrm{ind}(0)=1$，$\mathrm{ind}(1)=2$。那么，有以下断言：

每一个 Q 元素对应一个表元素$\mathbb{T}[i,0], i\neq 0$，即$\forall q\in Q\cdot q=\mathbb{T}[\mathrm{ind}(q),0]$，反之亦然；

对于 $q\in Q$ 和 $a\in\Sigma$，如果有 $\upsilon(q,a)=p$，$p\in Q$，于是$\mathbb{T}[\mathrm{ind}(q),\mathrm{ind}(a)]=p$，反之亦然；

对于 q_0，有 $\mathrm{ind}(q_0)=1$，可见$\mathbb{T}[1,0]$元素带有一个短箭头标记（或默认下标最小者，即也可以默认表元素$\mathbb{T}[1,0]$就是 q_0），反之亦然；

对于 F 中的每一个状态 q，表元素$\mathbb{T}[\mathrm{ind}(q),0]$带有星号标记，反之亦然。

可以证明，一个 DFA 的这两种表示是相互等价的，在此省略。归纳一下，DFA 的代数、转移图和转移表三种表示是相互等价的，对应关系罗列到表 2-3 中。

表 2-3　DFA 的三种表示之间对应关系

代数 A	转移图$\mathcal{G}(A)$	转移表$\mathbb{T}$
Q 元素 q	顶点（圆圈）	$\mathbb{T}[\mathrm{ind}(q),0]$
Σ 元素 a	权（弧标记）	$\mathbb{T}[0,\mathrm{ind}(a)]$
υ 元素	边（转移弧）	$\mathbb{T}[i,j], i,j\neq 0$
q_0	箭头标记顶点	箭头标记$\mathbb{T}[1,0]$
F 元素	双圈顶点	*标记$\mathbb{T}[i,0], i\neq 0$

2.2.2 DFA 的判定性质

回答 DFA A 的作用、原理等方面的问题，需要知道 A 是如何接受或拒绝输入串的。将字母表上的符号串作为输入，通过运行 A 可知哪个符号串接受，哪个不接受，这属于 DFA 的判定性质的作用范畴。判定性质告诉我们 Σ^*被划分为 A 接受的串集合与拒绝的串集合。如果知道 Σ 上的这些符号串 $L\subseteq\Sigma^*$都为 A 接受，而 $\Sigma^*\backslash L$ 都为 A 拒绝，就说 A 识别 L，是 L 的识别器。

1. 接受、拒绝输入串

给定输入串，DFA A 自动运行，得出接受或拒绝的结论。对于接受的输入串，A 有这样的运行过程：一次读入一个符号，并根据转移函数求出转移状态，A 发生一次状态转移，转移状态成为当前状态；重复此过程直到输入串中符号被依次读完；作为初始化，令 A 的初始状态作为当前状态，输入串的第一个符号为当前输入符号；运行过程成功时，当前状态是接受状态且输入串已消耗完。

形式地，DFA $(Q,\Sigma,\upsilon,q_0,F)$接受 $w\in\Sigma^*$，如果存在一个 Q 的状态转移序列 $r_0,r_1,\ldots,r_n$ 满足如下三条件：

(1) $r_0=q_0$；

(2) $\upsilon(r_i,w_{i+1})=r_{i+1}$，其中 $i=0,1,\ldots,n$；

(3) $r_n\in F$。

识别过程初始化为 $r_0=q_0$ 是当前状态，w_1 是当前输入符号，经历了 n 次状态转移，其中转移状态依次更替为 $r_1,r_2,\ldots,r_n$。当前输入符号依次更替为 $w_1,w_2,\ldots,w_n$。过程终止于 $r_n\in F$。

不被接受的输入串，称其被拒绝。对于被拒绝的输入串，DFA 一直可以运行到这个过程终止于输入串消耗完，却发现当前状态不是接受状态。特别地，如果原输入串为空，则没有读入符号，依据初始状态是否为接受状态决定接受还是拒绝。

接受、拒绝输入串的判定过程可通过算法 2.1 精确地表达。

算法 2.1 DFA 接受、拒绝输入串。

输入： DFA $A=(Q,\Sigma,\upsilon,q_0,F)$，$w\in\Sigma^*$。

输出： 返回真表示 A 接受 w，否则拒绝。

$q=q_0$

$x=w$

while $x \neq \varepsilon$ do {

$x = ay$

$q = \upsilon(q, a)$

$x = y$

}

return ($q \in F$)

算法中的变量 q 表示 A 的当前状态，a 表示 A 的当前输入符号，x 表示剩余输入串。算法初始化 q 为初始状态 q_0，初始化 x 为输入串，在 x 不空的时候它的第一个符号将作为当前输入符号 a。while 循环体为 A 的一次状态转移，其中，根据 q 和 a 计算 $\upsilon(q, a)$ 作为 q 最新值，该循环在输入串消耗完，即 $x = \varepsilon$ 时结束。

在算法 2.1 中，式子 $x = ay$ 表示已知 x 求 a 和 y 使得 a 和 y 连接等于 x，其中 a 是符号，x 和 y 是符号串。这里在上下文清楚的情况下直接使用 ay 这种连接运算，将符号 a 和符号串 a 灵活运用。这个式子实现了将剩余串 x 的第一个符号作为当前输入符号 a 并求出余下的剩余串 y 这一运算。

对于例 2.1 的 DFA M，模拟算法执行过程，其中算法的输入为 M 和输入串 $w = 00011$。算法执行过程的踪迹数据如表 2-4 所示。

表 2-4 DFA M 模拟算法执行过程的踪迹数据

算法步骤	当前状态 q	当前输入符号 a	剩余输入串 x
初始化	q_0	0	0011
第一次循环	q_0	0	011
第二次循环	q_0	0	11
第三次循环	q_1	1	1
最后一次循环	q_1	1	ε

如此重复直到输入串消耗完。若没有下一个输入符号，表明已读完输入串，根据自动机的当前状态得出判定结果。

在 M 的状态转换图上，算法 2.1 的执行过程有更为直观的表现。对于输入串 00011，得到状态转移序列为

$$q_0\text{-}0\text{-}q_0\text{-}0\text{-}q_0\text{-}0\text{-}q_0\text{-}1\text{-}q_1\text{-}1\text{-}q_1$$

这个序列的表示形式解释为：初始化当前状态为 q_0；对于当前输入符号为 0，发生一次状态转移，当前状态变为 q_0；对于当前输入符号 0，发生一次状态转移，当前状态变为 q_0；对于当前输入符号 1，发生一次状态转移，当前状态变为 q_1；对于当前输入符号 1，

发生一次状态转移，当前状态变为 q_1。至此已消耗完毕输入串，并因 q_1 为接受状态得出接受 00011 串的结论。

进一步把这个转移序列表示为$\mathcal{G}(M)$上的一条带权路径，得到始端是 q_0、末端是 q_1、标记为 00011 的路径。转移路径是一种全局视觉。考虑输入串 0101，表示为以它为标记的$\mathcal{G}(M)$上的一条始端为 q_0 末端为 q_2 的带权路径，根据末端不是接受状态得出被拒绝的结论。

一般地，对于 DFA $A=(Q,\Sigma,\upsilon,q_0,F)$，$w\in\Sigma^*$，$p,q\in Q$，采用 PATH$[q,p,w]$表示转移图$\mathcal{G}(M)$上始端为 q、末端为 p、标记为 w 的路径。显然，这个路径的长度等于$|w|$，顶点个数为$|w|+1$。PATH[]被方便地用于表示输入串 w 是否为 A 所接受。

2. DFA 的扩展转移函数。

判定 DFA 是否接受给定输入串的过程中，使用的转移序列可以有更为简洁的表示，事实上是扩展的转移函数。对于转移函数 υ，扩展转移函数记为 $\tilde{\upsilon}$。扩展 υ 的目的是理性表示在符号串上的连续转移，尽管已有转移序列和转移路径均有同样的效果。

定义 2.4 扩展转移函数 υ 为 $\tilde{\upsilon}$。

通过对输入串的长度进行归纳来定义 $\tilde{\upsilon}$。

基础：$\tilde{\upsilon}(q,\varepsilon)=q$。

归纳：$\tilde{\upsilon}(q,wa)=\upsilon(\tilde{\upsilon}(q,w),a)$。

基础部分解释为：如果在状态 q 下不读输入，当前状态就还处在状态 q，即 PATH$[q,q,\varepsilon]$。对归纳部分有如下解释：从 q 状态开始消耗输入串 wa 后到达的状态，等价于从 q 状态开始消耗掉输入串 w 到达的状态并再以 a 为当前输入符号发生转移一次最终到达的状态，也就是 PATH$[q,r,wa]$等价于 PATH$[q,p,w]\wedge$PATH$[p,r,a]$。

假设 w 是形如 xa 的串，即 a 是 w 的结尾符号，x 是包含除结尾符号外的所有符号的串，有 $\tilde{\upsilon}(q,w)=\upsilon(\tilde{\upsilon}(q,x),a)$。

换言之，给定状态 q 和输入串 $w=a_1a_2\ldots a_n$，$\tilde{\upsilon}$ 是这样计算的：从 q 开始依次选择标记为 $a_1,a_2,\ldots,a_n$ 的弧构成一条路径，路径末端就是 $\tilde{\upsilon}(q,w)$。即有

$$\tilde{\upsilon}(q,w)=p，\text{当且仅当 PATH}[q,p,w]$$

对于例 2.1 的 M，依据扩展转移函数的定义，计算输入串 011 的转移状态：

$$\begin{aligned}&\tilde{\upsilon}(q_0,011)\\&=\upsilon(\tilde{\upsilon}(q_0,01),1)\\&=\upsilon(\upsilon(\tilde{\upsilon}(q_0,0),1),1)\\&=\upsilon(\upsilon(\upsilon(\tilde{\upsilon}(q_0,\varepsilon),0),1),1)\\&=\upsilon(\upsilon(\upsilon(q_0,0),1),1)\end{aligned}$$

$$= \upsilon(\upsilon(q_0, 1), 1)$$
$$= \upsilon(q_1, 1)$$
$$= q_1$$

对于任意 $w \in \Sigma^*$，扩展转移函数可用于计算指定始端的、标记为 w 的转移路径的末端。比如：

$$\tilde{\upsilon}(q_1, 011)$$
$$= \upsilon(\tilde{\upsilon}(q_1, 01), 1)$$
$$= \upsilon(\upsilon(\tilde{\upsilon}(q_1, 0), 1), 1)$$
$$= \upsilon(\upsilon(q_2, 1), 1)$$
$$= \upsilon(q_2\ 1)$$
$$= q_2$$

不过，这样的转移不具有判定意义，因为起点 q_1 不是初始状态。

图 2-7 所示为例 2.1 的 M 的简化型，它的初始状态和接受状态没有变化，而省去了状态 q_2 以及以它为端点的边，从而状态总数减少 1 个，而且 q_1 上不再有 0 弧射出。

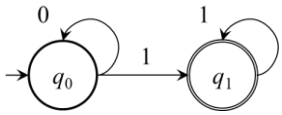


图 2-7　简化型 DFA M'

回顾习题 1.4 的口香糖球机问题，可能的 DFA 模型如图 2-8 所示，符合定义 2.1。事实上，按照 2.1 节采用的建模思路，容易得出不含 d 状态的 DFA，就是图 2-8 中去掉 d 节点及其邻接边。

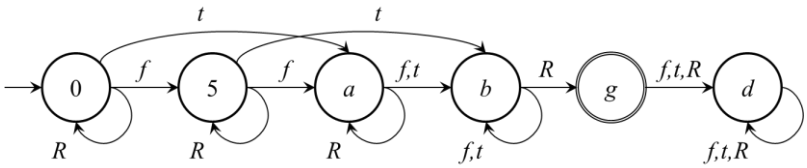


图 2-8　口香糖球机问题的完全形 DFA

d 是陷阱状态，对于一次成功的购买活动之后多余的操作都掉入陷阱而失败，如同画蛇添足，所以后一个模型通过 g 无射出弧更为简洁。总之，简化型 DFA 的使用使得模型简洁度得以提升，降低了有关算法的复杂度，将在 2.4 节予以分析。必须要做的是

扩展 DFA 的定义与表示，以包含这种简化表示。

3. 状态的可达性

回顾 DFA $A=(Q,\Sigma,\upsilon,q_0,F)$定义，因为需要满足$\forall q\in Q\cdot\forall a\in\Sigma\cdot\upsilon(q,a)\in Q$，所以$\mathcal{G}(A)$的状态顶点 $q\in Q$ 的可达性存在如下三种情形：

(1) $\forall w\in\Sigma^*\cdot q\neq\tilde{\upsilon}(q_0,w)$；

(2) $\forall p\in F,w\in\Sigma^*\cdot p\neq\tilde{\upsilon}(q,w)$；

(3) $\exists p\in F,\exists x,y\in\Sigma^*\cdot q=\tilde{\upsilon}(q_0,x)\wedge p=\tilde{\upsilon}(q,y)$。

状态可达性情形(1)表明$\mathcal{G}(A)$不存在始端为 q_0、末端为 q 的路径，表明 q 不是可达的。而情形(2)意味着不存在始端为 q、末端为接受状态的路径，表明 q 是失败的，除非 $q\in F$。对于判定性质而言，不可达的状态或者失败的状态都是无用状态。因为无用状态不在任何一条成功的转移路径上，所以与识别所有接受串都无关。状态可达性情形(3)表明，q 至少在一条成功的转移路径上，推导如下：

如果$\exists p\in F$，$\exists x,y\in\Sigma^*\cdot q=\tilde{\upsilon}(q_0,x)\wedge p=\tilde{\upsilon}(q,y)$

那么$\exists p\in F$，$\exists x,y\in\Sigma^*\cdot p=\tilde{\upsilon}(\tilde{\upsilon}(q_0,x),y)=\tilde{\upsilon}(q_0,xy)$

令 $w=xy$，得到$\exists w\in\Sigma^*\cdot\tilde{\upsilon}(q_0,w)\in F$，那么 w 为 A 接受。

因此称 q 为有用状态。

按照状态的可达性对 DFA 进行分类，分成三种类别：

第一类的状态都是可达的，即没有属于可达性情形(1)的状态；

第二类没有失败的状态，即没有属于可达性情形(2)的状态；

第三类的状态都是有用的，即既无情形(1)又无情形(2)的状态，都是属于情形(3)的状态。

从第一类 DFA 去除不可达状态，所识别的串集合不发生变化，但是状态数目减少了，因此这是一种化简，得到较为简化的 DFA。

将第二类 DFA 中的所有失败状态合并为一个状态，所识别的串集合不发生变化，但是状态数目减少了，因此这也是一种化简。

只有一个失败状态的 DFA A，就将这个状态称为陷阱状态，而 A 被称为陷入式 DFA。将 A 中的陷阱状态去除，所识别的串集合不发生变化，但是状态数目减少了一个，因此这也是一种化简。

反复进行上述化简动作，直到不能再简化为止，就得到最简 DFA，既不含无用状态又与原 DFA 等价。

然而不幸的是，化简可能导致转移函数不符合定义 2.1。这包括去除不可达状态和去除陷阱状态都导致转移函数不再是 $Q\times\Sigma$ 到 Q 的映射。不过合并失败状态为一个陷

阱状态所得到的仍然符合定义 2.1。基于此分析，最简 DFA 的判定性质需要包含突然死亡的情形，这是在形象地说，因为 $\upsilon()$值不存在而导致自动机的自主运行无法继续。

对于例 2.1，自动机 M 的 q_2 是失败状态，也是陷阱状态，可化简为

$$\text{DFA } M' = (\{q_0, q_1\}, \{0, 1\}, \upsilon, q_0, \{q_1\})$$

其中：

$\upsilon(q_0, 0) = q_0$

$\upsilon(q_0, 1) = q_1$

$\upsilon(q_1, 0) = \perp$

$\upsilon(q_1, 1) = q_1$

相比于 M，M'减少了一个状态，相应地 $\upsilon(q_1, 0)$不存在。从完备性上，使用值$\perp$指值无定义，那么有 DFA $M'' = (\{q_0, q_1, \perp\}, \{0, 1\}, \upsilon, q_0, \{q_1\})$，其中，$\upsilon(q_0, 0) = q_0$，$\upsilon(q_0, 1) = q_1$，$\upsilon(q_1, 0) = \perp$，$\upsilon(q_1, 1) = q_1$，$\upsilon(q_1, \perp) = \perp$，$\upsilon(q_1, \perp) = \perp$。

M''符合定义 2.1，但是，事实上 M''又是陷入式的，除非对$\perp$解释为 M''突然死亡，并立即得出判定结论为拒绝。注意，当 M''突然死亡时剩余串不一定为空串。因此有针对于最简 DFA 的判定性质如算法 2.2 所示。

定义 2.5　最简 DFA 的转移函数 υ 定义为

$$\forall q \in Q \cdot \forall a \in \Sigma \cdot \upsilon(q, a) \in Q \cup \{\perp\}$$

其中：$\upsilon(q, a) = \perp$表示 $\upsilon(q, a)$无定义。

与定义 2.1 对比，定义 2.5 允许转移函数的值为$\perp$，符号$\perp$不属于 Q。对于转移图和转移表表示，允许一些状态没针对每一个符号的射出弧，或允许表格元素为空（考虑到完备性时，采用符号$\perp$作为表元素）。

例 2.2　字母表{0, 1}上只接受串 010 和串 1 的 DFA。

结果如图 2-9(a)所示。这个 DFA 有一个陷阱状态 $q_\perp$，若将此删除就得到图 2-9(b)所示，二者仍然保持相互等价。

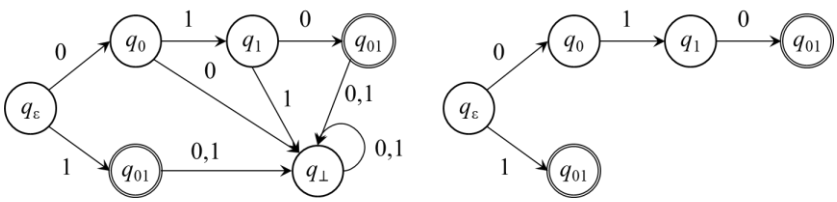


图 2-9　一个 DFA 的两种极端表示

$q_\perp$状态的命名表明自动机运行到此状态实际已经死亡了，所以，相比于图 2-9(a)，图 2-9(b)中缺失的射出弧意味着自动机因为无法转移而突然死亡。

再次考虑例 2.1 的 M，它的简化型 M'的代数表示为

$$\text{DFA } M' = (\{q_0, q_1\}, \{0, 1\}, \upsilon, q_0, \{q_1\})$$

其中：

$\upsilon(q_0, 0) = q_0$

$\upsilon(q_0, 1) = q_1$

$\upsilon(q_1, 0) = \perp$

$\upsilon(q_1, 1) = q_1$

对照完全形的判定性质，对最简 DFA，考虑到处理转移函数无定义情形，从而得到判定性质，即算法 2.2。

算法 2. 2　最简 DFA 的判定性质。

输入：DFA $A' = (Q, \Sigma, \upsilon, q_0, F)$，$w \in \Sigma^*$。

输出：返回真表示 A'接受 w，否则拒绝。

$q = q_0$

$x = w$

while x !=ε do {

　　$x = ay$

　　if (($q = \upsilon(q, a)$)==$\perp$) return 0

　　$x = y$

　　}

return ($q \in F$)

算法中变量 q 表示 A'的当前状态，a 表示当前输入符号，x 表示剩余输入串。while 循环体对应一次转移，根据 q 和 a 计算 $\upsilon(q, a)$作为 q 的最新值。该循环在输入串消耗完，即 $x = \varepsilon$时结束。特别地，如果转移函数的值无定义则算法直接返回 FALSE，表示拒绝。

完全形和最简形是 DFA 的两种极端表示，还有介于二者之间的表示，就是既不满足可达性(1)也不满足可达性(2)。这种表示更为一般化，同时也包含两个极端表示形式在内。所以以后除非有必要特别指出，都采用一般化的 DFA 表示，所用判定性质也如算法 2.2 所示。

示例 M'判定输入串 101 的运行过程：

$\tilde{\upsilon}(q_0, 101) = \upsilon(\tilde{\upsilon}(q_0, 10), 1) = \upsilon(\upsilon(\tilde{\upsilon}(q_0, 1), 0), 1) = \upsilon(\upsilon(q_1, 0), 1)$

此时，因 $\upsilon(q_1, 0) = \perp$而导致运行意外终止，判定结论为拒绝。

2.2.3　DFA 的语言

利用不同形式的判定性质，接受、拒绝输入串，成功的转移路径，扩展转移函数和算法 2.2 等，都能得出任意 DFA A 定义的语言，简称 A 的语言，记为 $L(A)$。其意义不仅在于 A 是 $L(A)$的语言识别器，A 识别 $L(A)$、拒绝 $\Sigma^*\backslash L(A)$，更重要的是，A 是 $L(A)$的模型，应用于期望语言的建模和应用。

定义 2.6　DFA $A = (Q, \Sigma, \upsilon, q_0, F)$的语言

$$L(A) = \{w \mid \tilde{\upsilon}(q_0, w) \in F\}$$

这个形式定义准确地描述了 A 的语言就是 A 所接受的符号串的全集。下面给出一些例子，示意应用 DFA 对语言建模的一些关注点。

例 2.3　给出$\{0, 1\}$上的以 11 为前缀的串。

设计状态用于记忆，记住已消耗串是否为 11。各状态用于局部性地判定已消耗串是输入串的前缀，并依据其记忆来决定后续运行。

初始状态 q_ε 记住 0 前缀，依据其记忆不能判定输入串，所以 q_0 不能是接受状态；

q_1 记住 1 前缀为 1，依据其记忆不能判定输入串，所以 q_1 不能是接受状态；

q_0 记住 1 前缀为 0，依据其记忆不能判定输入串，所以 q_0 不能是接受状态；

q_{00} 记住 2 前缀 00，依据其记忆得出拒绝输入串之结论；

q_{01} 记住 2 前缀 01，依据其记忆得出拒绝输入串之结论；

q_{10} 记住 2 前缀 10，依据其记忆得出拒绝输入串之结论；

q_{11} 记住 2 前缀 11，依据其记忆得出接受输入串之结论。

从这个设计结果再求出最简 DFA，得到如下答案：

	0	1
$\rightarrow q_\varepsilon$	$\perp$	q_1
q_1	$\perp$	q_{11}
$^*q_{11}$	q_{11}	q_{11}

例 2.4　给出$\{0, 1\}$上的含有子串 11 的串。

一种基于模式匹配的设计思路是，在扫描输入串的过程中找出一个模式 11 并得出接受结论，或者因剩余串为空为拒绝。利用状态做记忆，记住已消耗串的个别汇总结果，即找到子串 11。答案如下：

	0	1
$\rightarrow q_0$	q_0	q_1
q_1	q_0	q_2
$*q_2$	q_2	q_2

此答案与例 2.3 的原始设计本质不同的是只需记住个别情形（接受）而无需记住所有情形。

例 2.5 给出{0, 1}上最多有两个 1 的串。

利用状态做记忆，记住对已消耗串的全面汇总结果。而状态转移有逐个符号进行汇总统计的效果。答案如下：

	0	1
$\rightarrow *q_0$	q_0	q_1
$*q_1$	q_1	q_2
$*q_2$	q_2	q_{3+}
q_{3+}	q_{3+}	q_{3+}

例 2.6 构建 DFA 接受含有偶数个 0 和偶数个 1 的 0-1 串。

利用状态做记忆，全面汇总为 4 种情况。具体分析已消耗掉的输入串部分，只有如下 4 种情形，分别用状态来记住：

ee：含有偶数个 0 和偶数个 1 的串；

eo：含有偶数个 0 和奇数个 1 的串；

oe：含有奇数个 0 和偶数个 1 的串；

oo：含有奇数个 0 和奇数个 1 的串。

继续读入符号将继续发生状态转移，但是转移状态还是这 4 个中的 1 个。当输入串消耗完以后，从到达的状态可知原输入串的情形。比如到达 ee 的话，原输入串就是偶数个 0 和偶数个 1，如图 2-10 所示。

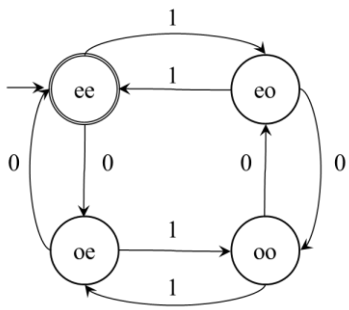


图 2-10 含有偶数个 0 和偶数个 1 的 0-1 串

例 2.7　不包含一对 1 且中间被奇数个 0 隔开的任何串。

一种建模思路为，排除包含模式 10...01 的串，该模式中的 0 为奇数个。通过 F 与 $Q\backslash F$ 互换角色完成排除任务，结果如图 2-11 所示。

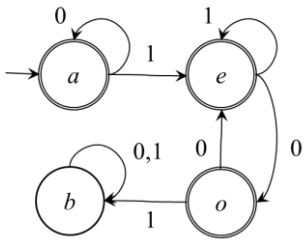


图 2-11　含有模式 1<偶数个 0>1 的 0-1 串

对图 2-11 所示化简，得到的最简 DFA 为删除图中 b 状态及其邻边而剩下的部分，并作为答案。

例 2.8　给出能被 5 整除的二进制数。

令 $L=\{w\in\{0,1\}^* \mid w$ 为能被 5 整除的二进制数$\}$，对如图 2-12 所示的建模结果进行分析解释。一个数除以 5，其余数只能是 0，…，4 这 5 种，因此用 q_0，…，q_4 表示这 5 种状态。因为接受能被 5 整除的数，故状态 q_0 既为初始状态，又为接受状态。接着，考虑一个二进制数的串与一个 0 或 1 连接时，状态的转化情况。在二进制串后增添 1 位，可理解为将先前的数值乘 2 再加上所添的数值。那么，计算串尾添数后新的数值模 5 的余数，不外乎这 5 种情形，即得到添 0 或 1 后的新的状态。如图 2-12 所示，每个状态都是有用的，所以是最简 DFA。

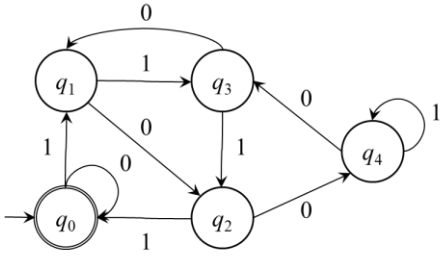


图 2-12　能被 5 整除的 0-1 串

例 2.9　给出后缀为 01 的串。

利用状态做记忆，全面汇总，记住已消耗串的暂时汇总结果。其中 4 个状态 q_{00}、q_{01}、q_{10} 和 q_{11} 记住已消耗串的最后两位，直到输入串消耗完，从成功结束于它们中的哪一个就可知输入串是以该状态下标所示结尾，当然输入串 ε、0 和 1 被排除在外，因

为它们没有后两位。结果如图 2-13 所示。

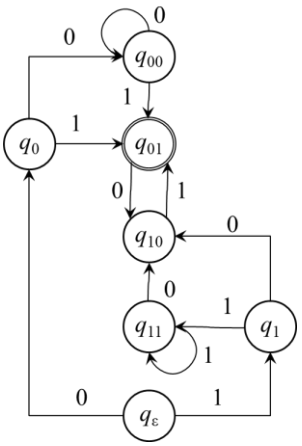


图 2-13　以 01 为后缀的 0-1 串

例 2. 10　给出以 101 为后缀的 0-1 串。

仍然利用状态做记忆，全面汇总，记住已消耗串的暂时汇总结果。这个例子显示出与前例同样问题但存在组合爆炸情形。需要用 8 个状态 $q_{000}, \cdots, q_{111}$ 记住已消耗串的最后 3 位，得到如图 2-14 所示的设计结果。

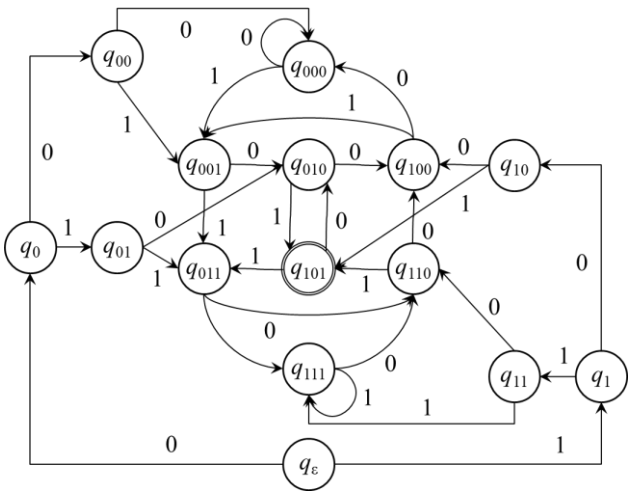


图 2-14　识别以 101 为后缀的 0-1 串

从图 2-14 中可见，只有已消耗串后缀为 101 才能使得 DFA 当前状态为 q_{101}。所以输入串消耗完时当前状态若是 q_{101} 则表明输入串以 101 为后缀。

观察例 2.9，将状态作为记忆体，记住当自己成为当前状态时，已消耗掉的串的两位后缀为何。使用 4 个状态分别记住消耗掉的串的后缀为 00、01、10 和 11，比如用

q_{00}、q_{01}、q_{10} 和 q_{11} 依次记住消耗掉的串的两位后缀，这样，当输入串消耗完时这 4 个状态就记录了整个输入串的两位后缀为何，其中选定 q_{01} 为接受状态，那么接受后缀为 01 的输入串，符合题意。

可看到例 2.10 采用类似的思路，由于 3 位后缀有 8 种情况，故使用 8 个状态分别记忆之，并将表示消耗串后缀为 101 的状态作为接受状态即可。按照这样的思路，随着问题所要求后缀长度的增加，要用状态记忆的情形数量以指数形式增大，导致状态爆炸。为了避免状态爆炸，需要换一种建模思路，允许自动机猜测剩余串长度达到给定值的情形，一旦猜中，则状态下标就示意输入串的期望后缀是什么。

如图 2-15 所示，如果一个 FA 对于输入串能够猜测到什么时候到达剩余串为 3 个符号，则接下来就如图中的转移所示，直接得出判定结果。为了实现自动机的猜测功能，我们将 DFA 的转移函数做出扩展，允许一个状态有多条标记相同的弧射出，这是给自动机引入了一种不确定性，如图 2-16 所示，在状态 q_0 上对于当前输入符号 0 自动机既转移到自身又转移到状态 q_1。设想如果自动机足够聪明，能猜测到剩余串长度为 2 的话，就在剩余串长度大于 2 时，只在 q_0 上转移并消耗输入串。当剩余串为 2 个符号的时候，才从 q_0 往后转移，直到对于后缀为 01 的输入串到达 q_2 返回接受，其他情况则拒绝。当然输入串长度小于 2 的情况下，都因为到不了接受状态而被拒绝。

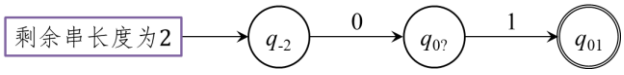


图 2-15　FA 猜测示意

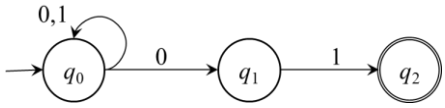


图 2-16　借助不确定性猜测

这种状态转移有不确定性的 FA 就是非确定有穷自动机 NFA。对于猜测剩余串长度等于 2 这样的行为，图 2-16 所示 NFA 是这样来实现的。在状态转移过程中，某个状态因有多个同标记射出弧而导致转移路径发生分叉，产生多个支路。随后的转移路径还可能多次分叉，其结果是让 NFA 经历了始端都为初始状态的多个转移路径。我们把这些路径称为相对于同一个输入串的不同线索。

之所以采用“线索”这个术语，是因为这个判定过程可类比于破案行为。破案

行为描述为顺着多条破案线索分别进行下去，只要有一条线索最终完成破案，那么破案行为就成功结束了。以此类比，NFA 的判定性质也是多条线索中只要有一条线索取得成功就得出接受之结论。考虑图 2-17 所示的 3 条线索：中间的一条线索以输入串 10101 为标记且末端为接受状态，是一条成功的转移路径；其余的两条线索中一条因为自动机突然死亡而没有消耗完输入串，另一条虽然消耗完了输入串但是末端不是接受状态，都不是成功的转移路径。那么根据前述原则即可得出接受 10101 的结论。

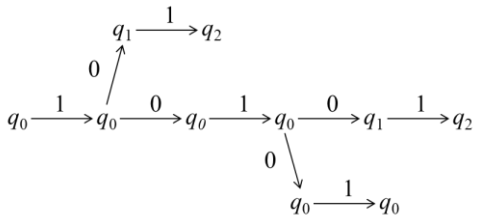
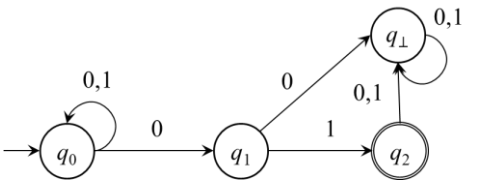
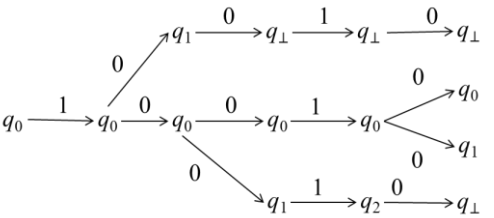


图 2-17　输入串 10101 的线索穷举

另一种设计思路是让 NFA 运行的每条线索上的标记都是同一个输入串。以图 2-18 为例，由于借鉴了 DFA 使用的陷阱状态，可以让 NFA 轻松消耗完输入串。如图 2-18(a)所示，相比于图 2-16 的设计，增加了陷阱状态 $q_\perp$。如图 2-18(b)所示，运行输入串 10010 经历 4 条线索，每条线索的标记都是 10010，都不能结束于接受状态，所以该串被拒绝。



(a) 陷入式 NFA



(b) 10010 的线索穷举树

图 2-18　另一种 NFA 设计

归纳一下：对于输入串，线索是自动机的一条以初始状态为始端的状态转移路径，该路径的标记正好是输入串的某个前缀；对于陷入式 NFA 而言，线索是整个输入串。对于任意 NFA，判定 w 为接受，当且仅当运行在一条始端为初始状态的线索，线索标记是 w 且末端是接受状态。

线索穷举是 FA 的运行方式。DFA 运行时只有一条线索（路径），NFA 运行时有有穷条线索。对于给定的输入串，只关注是否有一条线索成功就够了。成功的线索满足三个条件：以初始状态为始端，以接受状态为末端，以输入串为路径标记。

定义 2.7　陷入式 NFA 的输入串 w 的线索穷举树是这样一棵树：

树根为初始状态，位于树的第 0 层；

从树根到每一片叶子的路径都是一条运行线索；

任意两个运行线索都有一个最大公共前缀，如果含有 i 个状态，$i>0$，其最后一个状态记为 q_i，那么 q_i 在树的第 $i-1$ 层，它有两个孩子，分别属于那两条线索上的第 $i+1$ 状态，而且都属于树的第 i 层。

将树的第 i 层（$i \geqslant 0$）的所有节点（为状态）用集合表示，记为 T_i，那么在这个输入串上的运行过程描述为：以根为起点，并行地沿着每条线索，都消耗掉输入串的 i 前缀（长度为 i 的前缀），并转移到 T_i 中每一个状态。当输入串 w 消耗完时，根据 $T_{|w|} \cap F \neq \varnothing$ 决定接受。

称 T_i 为活动状态集，即消耗完输入串 w 的 i 前缀时的当前状态的集合。假如把活动状态集合当作一个状态来看待的话，输入串的转移过程就是确定的了。采用归纳思想描述如下。

基础：$i=0$，$T_0=\{q_0\}$，因为 q_0 是所有运行线索的始端。

归纳：$i=k$，$0<k\leqslant|w|$，$T_k=\bigcup q\in T_{k-1}\cdot\{p \mid$ 有标记为 w_k 的转移弧从 q 射出射入 $p\}$。

对于图 2-16 所示 NFA 在输入串 10101 上的线索穷举树，用活动状态集合表示运行过程为

$$\{q_0\},1,\{q_0\},0,\{q_0,q_1\},1,\{q_0,q_2\},0,\{q_0,q_1\},1,\{q_0,q_2\}$$

可见路径末端 $T_5=\{q_0,q_2\}$ 包含接受状态 q_2，所以 10101 被接受。

对于图 2-18 所示陷入式 NFA 在输入串 10010 上的线索穷举树，用活动状态集合表示运行过程为

$$\{q_0\},1,\{q_0\},0,\{q_0,q_1\},0,\{q_0,q_1,q_\perp\},1,\{q_0,q_2,q_\perp\},0,\{q_0,q_1,q_\perp\}$$

可见路径末端 $T_5=\{q_0,q_1,q_\perp\}$ 中没有接受状态，所以 10010 被拒绝。

2.3 非确定型有穷自动机

非确定型有穷自动机（NFA）在识别输入串过程中，由于状态有多条同标记射出弧，导致当前状态不是一个状态，而是多个状态，表明 NFA 自主运行时有同时处于多个状态的能力。通常把这种能力说成是对输入串进行“猜测”的能力。以后我们用活动状态集合表示运行时的每一个并行转移步骤中，转移状态变化就是活动状态集合的改变。

例 2.11 设计 NFA，识别以 01 结尾的 0-1 串。

设计结果如图 2-16 所示。对于长度大于 2 的输入串，可以认为，只要结尾两个符号还没有遇到，就处在 q_0 状态。在 q_0 状态有两条射出的弧都标记为 0，意味着当前输入符号是 0 时，产生两个猜测：一个是没有到达最后两个符号，状态还是转移至 q_0；另一个是到达最后两个符号，那么状态转移至 q_1。

猜测导致识别输入串过程中产生多条线索，每一条线索就是一系列状态转移。一些线索没有成功的原因可以是因为缺少对应的箭弧而死亡，或者输入串消耗完但是没有到达接受状态。如果有一条线索成功，也就是输入串消耗完且到达接受状态，那么就断定该输入串被接受。

在本例中，如果输入串是 01001，则识别过程中一共产生 3 条线索：

q_0-0- q_0-1- q_0-0- q_0-0- q_0-1- q_0（没有成功）

q_0-0- q_0-1- q_1-0-（死亡）

q_0-0- q_0-1- q_0-0- q_0-0- q_1-1- q_2（成功）

第一条线索虽然消耗完输入串但没有到达接受状态，所以不成功；第二条线索到达 q_1 状态后因为没有标记为 0 的箭弧射出故而死亡；第三条线索消耗完输入串且到达接受状态，所以成功。结论是这个 NFA 识别符号串为 01011。

如果输入串是 01110，识别过程一共产生 5 条线索：

q_0-0- q_0-1- q_0-1- q_0-1- q_0-0- q_0（没有成功）

q_0-0- q_1-1- q_2-1-（死亡）

q_0-0- q_0-1- q_1-1- q_2-1-（死亡）

q_0-0- q_0-1- q_0-1- q_1-1- q_2-0- （死亡）

q_0-0- q_0-1- q_0-1- q_0-1- q_1-0-（死亡）

由于没有成功的线索，所以这个 NFA 拒绝该串。

2.3.1　定义及表示

定义 2.8　一个非确定型有穷自动机 NFA A 是一个五元组$(Q, \Sigma, \upsilon, q_0, F)$，其中：

Q 是一个有穷状态集合；

Σ 是字母表，一个有穷的输入符号的集合；

υ 是转移函数，记为 $\upsilon: Q \times \Sigma \to 2\text{^}Q$，函数值 $\upsilon(q, a) = S$ 是 Q 的子集，表示在 q 状态输入符号为 a 时同时转移到 S 中每一个状态；

$q_0 \in Q$ 是初始状态；

$F \subseteq Q$ 是接受状态的集合，接受状态有时也称为接受状态。

与定义 2.1 比较，区别仅在于转移函数的返回值不同，DFA 的返回值是一个状态，而 NFA 的返回值是状态集合。NFA 的转移函数值也正好反映了其不确定性。

NFA 的转移表表示与 DFA 相似，除了表元素是状态集合以外，其他部分含义相同。如用转移表来表示例 2.11，有

	0	1
$\to q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$
q_2	Ø	Ø

对于某状态在给定输入符号上没有箭弧射出，用空集 Ø 表示。将此写成代数形式，有 NFA $(\{q_0, q_1, q_2\}, \{0, 1\}, \upsilon, q_0, \{q_2\})$，其中 $\upsilon = \{(q_0, 0, \{q_0, q_1\}), (q_0, 1, \{q_0\}), (q_1, 1, \{q_2\})\}$。注意，$\upsilon$ 可被写成集合形式，也可写成函数形式：

$\upsilon(q_0, 0) = \{q_0, q_1\}$

$\upsilon(q_0, 1) = \{q_0\}$

$\upsilon(q_1, 1) = \{q_2\}$

类似于 DFA，NFA 也有状态转移图表示，唯一区别在于 NFA 允许从状态有多条同标记弧射出，而 DFA 的同一状态只允许最多一条同标记弧射出。参见图 2-16，状态 q_0 有两条 1 标记弧射出。

NFA 的代数、转移图和转移表三种表示是等价的，因此我们可以采用其中任意一种来指明所说的。一般地，代数表示适合于推理、证明等过程，转移图表示比较直观，便于人观看、分析，而转移表表示便于程序实现。

2.3.2 扩展转移函数

与 DFA 一样，为了方便表达 NFA 中发生的一系列状态转移现象，扩展它的转移函数 $\upsilon(q,a)$为 $\tilde{\upsilon}(q,w)$，用以表示从状态 q 开始读入符号串 w 后到达的那些状态，记为 S。从猜测线索上解释，S 中的元素都是那些始端为 q、标记为 w 的线索的末端。一些线索中途死亡了，从 S 中不会反映出来。

定义 2.9　NFA 的扩展转移函数 $\tilde{\upsilon}()$是一个类型为 $Q\times\Sigma^{*}\to 2\hat{}Q$ 的函数。

基础：$\tilde{\upsilon}(q,\varepsilon)=\{q\}$。仅一条长度为 0 的路径，始端和末端都是 q。也就是说，不读入任何符号，就只能处在原来的状态。

归纳：$\tilde{\upsilon}(q,wa)=\cup p\in\tilde{\upsilon}(q,w)\cdot\upsilon(p,a)$。集合 $\tilde{\upsilon}(q,w)$的每一个状态射出的 a 弧所射入的全部状态就是 $\tilde{\upsilon}(q,wa)$。是始端为 q、标记为 wa、末端为 $\tilde{\upsilon}(q,wa)$元素的线索。

从输入串 w 的线索穷举树解释扩展转移函数 $\tilde{\upsilon}(q,w)$：

树根为 q；

$T_0=\tilde{\upsilon}(q,\varepsilon)$，$T_1=\tilde{\upsilon}(q,\pi(w,1))$，$T_i=\tilde{\upsilon}(q,\pi(w,1))$，…，$T_{|w|}=\tilde{\upsilon}(q,\pi(w,|w|))$; $T_i=\cup p\in T_{i-1}\cdot\upsilon(p,w_i)$，$0\leqslant i\leqslant|w|$。

也就是说，对于前一层的状态集合中每个状态，标记为 w_i 的射出箭弧到达的所有状态，这些状态就组成后一层状态集合。工具函数 $\pi(x,i)$返回符号串 x 的长度为 i 的前缀。

针对例 2.11，存在代数推导方式的计算过程：

$$\begin{aligned}
&\tilde{\upsilon}(q_0,01011)=\cup p\in\tilde{\upsilon}(q_0,0101)\cdot\upsilon(p,1)\\
&=\cup p\in[\cup p\in\tilde{\upsilon}(q_0,010)\cdot\upsilon(p,1)]\cdot\upsilon(p,1)\\
&=\cup p\in[\cup p\in[\cup p\in\tilde{\upsilon}(q_0,01)\cdot\upsilon(p,0)]\cdot\upsilon(p,1)]\cdot\upsilon(p,1)\\
&=\cup p\in[\cup p\in[\cup p\in[\cup p\in\tilde{\upsilon}(q_0,0)\cdot\upsilon(p,1)]\cdot\upsilon(p,0)]\cdot\upsilon(p,1)]\cdot\upsilon(p,1)\\
&=\cup p\in[\cup p\in[\cup p\in[\cup p\in[\cup p\in\tilde{\upsilon}(q_0,\varepsilon)\cdot\upsilon(p,0)]\cdot\upsilon(p,1)]\cdot\upsilon(p,0)]\cdot\upsilon(p,1)]\cdot\upsilon(p,1)\\
&=\cup p\in[\cup p\in[\cup p\in[\cup p\in[\cup p\in\{q_0\}\cdot\upsilon(p,0)]\cdot\upsilon(p,1)]\cdot\upsilon(p,0)]\cdot\upsilon(p,1)]\cdot\upsilon(p,1)\\
&=\cup p\in[\cup p\in[\cup p\in[\cup p\in\{q_0,q_1\}\cdot\upsilon(p,1)]\cdot\upsilon(p,0)]\cdot\upsilon(p,1)]\cdot\upsilon(p,1)\\
&=\cup p\in[\cup p\in[\cup p\in\{q_0,q_2\}\cdot\upsilon(p,0)]\cdot\upsilon(p,1)]\cdot\upsilon(p,1)\\
&=\cup p\in[\cup p\in\{q_0,q_1\}\cdot\upsilon(p,1)]\cdot\upsilon(p,1)\\
&=\cup p\in\{q_0,q_2\}\cdot\upsilon(p,1)\\
&=\{q_0\}
\end{aligned}$$

也存在归纳方式的计算过程，即依次计算出中间结果，最后得到最终结果：

$$\tilde{\upsilon}(q_0,\varepsilon)=\{q_0\}$$

$$\tilde{\upsilon}(q_0,0)=\cup p\in\tilde{\upsilon}(q_0,\varepsilon)\cdot\upsilon(p,0)=\{q_0,q_1\}$$

$\tilde{\upsilon}(q_0, 01) = \cup p \in \tilde{\upsilon}(q_0, 0) \cdot \upsilon(p, 1) = \upsilon(q_0, 1) \cup \upsilon(q_1, 1) = \{q_0, q_2\}$

$\tilde{\upsilon}(q_0, 010) = \cup p \in \tilde{\upsilon}(q_0, 01) \cdot \upsilon(p, 0) = \upsilon(q_0, 0) \cup \upsilon(q_2, 0) = \{q_0, q_1\}$

$\tilde{\upsilon}(q_0, 0101) = \cup p \in \tilde{\upsilon}(q_0, 010) \cdot \upsilon(p, 1) = \upsilon(q_0, 1) \cup \upsilon(q_1, 1) = \{q_0, q_2\}$

$\tilde{\upsilon}(q_0, 01011) = \cup p \in \tilde{\upsilon}(q_0, 0101) \cdot \upsilon(p, 1) = \upsilon(q_0, 1) \cup \upsilon(q_2, 1) = \{q_0\}$

2.3.3　NFA 的语言

作为语言识别器的 NFA 用来定义语言，基于判定性质，所有接受的符号串集合就是所定义的语言，简称 NFA 的语言。形式地，NFA $A = (Q, \Sigma, \upsilon, q_0, F)$的语言为

$$L(A) = \{w \in \Sigma^* \mid \tilde{\upsilon}(q_0, w) \cap F \neq \emptyset\}$$

这个定义也显示，只要有一条成功线索，就表示接受该输入串。

例 2.12　证明例 2.11 定义的 NFA 接受后缀为 01 的语言。

根据状态转移函数的这样一个特点，当前状态与当前输入符号共同决定转移状态，那么当输入串最后一个符号为当前符号时，当前状态是什么？随后到达哪一个状态？这样，证明过程就分解为针对以下三个命题，注意仅针对本例成立，不具有一般性。

命题一：$w \in \Sigma^*, q_0 \in \tilde{\upsilon}(q_0, w)$。

命题二：$q_1 \in \tilde{\upsilon}(q_0, w)$，当且仅当 w 以 0 结尾。

命题三：$q_2 \in \tilde{\upsilon}(q_0, w)$，当且仅当 w 以 01 结尾。

由于这个自动机的接受状态只有一个，即 q_2，所以它的语言是满足 $q_2 \in \tilde{\upsilon}(q_0, w)$的全部 w，其中 $w \in \Sigma^*$。基于 w 的长度进行归纳。

基础：如果 $|w| = 0$，即 $w = \varepsilon$。根据 $\tilde{\upsilon}$ 定义的基础 $\tilde{\upsilon}(q_0, \varepsilon) = \{q_0\}$，命题一满足。对于命题二和命题三，$\varepsilon$ 既不以 1 结尾也不以 01 结尾，所以都满足。

归纳：令 $w = xa$，因为$|w|>0$，$w \in \Sigma^*$，那么 x 是长度大于等于 0 的 0-1 串，而 a 是符号 0 或 1。设$|w| = n+1$，那么假设命题一到命题三对于长度小于 n 的输入串都成立，即对 x 成立。下面证明对 w 成立。

(1) 初始状态 q_0 同时有标记为 0 和 1 的箭弧射出并射入自身，所以 $q_0 \in \tilde{\upsilon}(q_0, w)$。

(2) （当）若 w 以 0 结尾，即 $a = 0$，根据命题一，$q_0 \in \tilde{\upsilon}(q_0, x)$。由于 q_0 上有标记为 0 的箭弧射出到 q_1，所以 $q_1 \in \tilde{\upsilon}(q_0, w)$。

（仅当）若 $q_1 \in \tilde{\upsilon}(q_0, w)$，由于射入 q_1 的箭弧只有一条从 q_0 射出的标记为 0 的箭弧，所以 w 中最后一个符号为 0 才能到达 q_1，否则为 1，不能到达 q_1。

(3) （当）若 w 以 1 结尾，即 $a = 1$ 且 x 以 0 结尾，根据命题二，$q_1 \in \tilde{\upsilon}(q_0, x)$，由于有标记为 1 的转移弧从 q_1 到 q_2，所以 $q_2 \in \tilde{\upsilon}(q_0, w)$。

（仅当）若 $q_2 \in \tilde{\upsilon}(q_0, w)$，由于射入 q_2 的箭弧只有一条从 q_1 射出的标记为 1 的箭弧，所以 w 中最后一个符号为 1 才能到达 q_2，否则为 0，不能到达 q_2。这时 $w = x1$，且 $q_1 \in \tilde{\upsilon}(q_0, x)$。根据命题二，$x$ 的最后一个符号是 0，据此 w 以 01 结尾。

2.3.4 NFA 的等价性质

对于“以 01 结尾的 0-1 串”这样的语言，前文已给出它的一个 DFA，如图 2-13 所示。对于任意输入串 $w \in \{0, 1\}^*$，从初始状态 q_ε 开始，都有一条长度为 $|w|$ 的转移路径，正好消耗完 w，最后是否到达接受状态 q_{01} 取决于串的结尾是否为 01。

前文也给出了这种语言的一个 NFA，如图 2-16 所示。对比这种语言的两种 FA，可见前者较为容易构造而且状态数较少。随着关注 0-1 串结尾符号个数的增加，DFA 的状态数目呈现为指数（2 的幂）增加趋势，而 NFA 的仅呈现出线性增加趋势。

有鉴于此，提出一个有趣的问题：NFA 和 DFA 在定义语言的能力上是否等价呢？换言之，对于任意一种语言，如果为 DFA 所定义，那么是否能为 NFA 定义呢？反过来也一样。这就是 NFA 与 DFA 二者等价性问题。

定理 2.1　对于任意一个 NFA N 存在一个等价的 DFA D 使得 $L(N) = L(D)$。

采用构造性证明，就是给出具体的方法，将 NFA 等价地转换为 DFA。这个转换方法命名为子集构造法，步骤如下。

初始化，令 $N = (Q_N, \Sigma, \upsilon_N, q_0, F_N)$，那么 $D = (Q_D, \Sigma, \upsilon_D, \{q_0\}, F_D)$。其中二者的输入符号集合相同，用 N 的初始状态 q_0 构成的集合$\{q_0\}$表示 D 的初始状态。本方法中，对 D 采用 Q_N 的子集来表示其状态，也是子集构造法名称来由。

(1) 构造 Q_D。首先说明 $Q_D \subseteq 2\text{^}Q_N$，即 Q_D 是 Q_N 的幂集的子集。如果 Q_N 的状态数目为 n 那么 Q_D 的状态数目最坏情况下可达 2^n。当然，一般情况下 $2\text{^}Q_N$ 中有许多死状态(即从 q_0 出发不可达的状态)，故而不必出现在 Q_D 中。

(2) 构造 F_D。对于 $S \in Q_D$，若 $S \cap F_N \neq \emptyset$，则 $S \in F_D$。也就是说，对于 Q_D 中那些包含有 NFA N 的接受状态的元素，令其为 Q_D 的接受状态。

(3) 构造 υ_D。对于每一个 $S \subseteq Q_N$ 以及每一个 $a \in \Sigma$，令 $\upsilon_D(S, a) = \bigcup p \in S \cdot \upsilon_N(p, a)$。换言之，为了计算 $\upsilon_D(S, a)$，检查 S 中每一状态，看看从其射出的带有标记 a 的箭弧射入哪些状态，所有这些状态构成的集合就是结果。

回顾例 2.11 和例 2.12，给出语言“以 01 结尾的 0-1 串”的 NFA 和 DFA，作为例子，这里采用直接从 NFA 转换的办法得到 DFA，再与直接构造的 DFA 对照一下，是令人期待的。

此 NFA 已在前文给出，见图 2-16。令 $Q_N = \{q_0, q_1, q_2\}$，构造 $Q_D = 2\text{^}Q_N$，共有 8 个状态，如转移表中第一列所示。注意是用集合来代表 DFA 状态的。

其次，构造 F_D。逐一判断中 Q_D 的元素，比如 S，若与 F_N 相交不为空，则 S 为 F_D 元素。具体而言，凡是含有 q_2 的那些 Q_D 的元素就是 DFA 的接受状态。

最后，构造 υ_D。对于转移表最左一列每一集合 S，标记为 0 的箭弧射入哪些状态呢？需要根据 υ_N 计算。比如 $S = \{q_0, q_1\}$，$\upsilon_N(q_0, 0) = \{q_0\}$，$\upsilon_N(q_1, 0) = \emptyset$。

如果状态用集合来指代的话，表 2-5 就是 DFA 的转移表表示。

表 2-5　后缀为 01 的 0-1 串的子集法示例

	0	1
Ø	Ø	Ø
→{q_0}	{q_0, q_1}	{q_0}
{q_1}	Ø	{q_2}
*{q_2}	Ø	Ø
{q_0, q_1}	{q_0, q_1}	{q_0, q_2}
*{q_0, q_2}	{q_0, q_1}	{q_0}
*{q_1, q_2}	Ø	{q_2}
*{q_0, q_1, q_2}	{q_0, q_1}	{q_0, q_2}

进一步通过去除无用状态求出最简 DFA：

	0	1
→{q_0}	{q_0, q_1}	{q_0}
{q_0, q_1}	{q_0, q_1}	{q_0, q_2}
*{q_0, q_2}	{q_0, q_1}	{q_0}
*{q_0, q_1, q_2}	{q_0, q_1}	{q_0, q_2}

状态数目减少 4 个。化简过程启发我们逐行构造 DFA 转移表可以避免无用状态出现，从而使计算量减少。

这种改进的方法中，首先在第一行第一列写上 DFA 的初始状态，并计算出该行的第二和第三列。若其没有出现在第一列则加入第一列最后。

然后重复计算新的一行并完成比较和添加工作。直到不再有状态被加入第一列，构造过程结束，返回转移表。

最后对最简 DFA 的状态命名，重新引进符号命名原本用集合表示的状态，得到常规形式的 Q_D 和 υ_D。在上下文清楚的情况下，这一步可省略。

将上述过程进行归纳并形式化表示。已知NFA转为等价的DFA过程分为两个阶段，第一阶段得到等价 DFA，第二阶段去除无用状态，得到最简 DFA。

设有 NFA $(Q, \Sigma, \upsilon, q_0, F)$，把 $S, T \subseteq Q$ 作为 DFA 的状态（子集构造法），若有

$$\exists q \in S, p \in T, a \in \Sigma \cdot p \in \upsilon(q, a)$$

那么令 $T = \upsilon'(S, a)$，$\upsilon' = \{(S, a, T) \mid S \subseteq Q, a \in \Sigma, T = \cup q \in S \cdot \upsilon(q, a)\}$，$F' = \{S \subseteq Q \mid S \cap F \neq \varnothing\}$，得到 DFA $(2^{\wedge}Q, \Sigma, \upsilon', \{q_0\}, F')$。这是与原 NFA 等价的。

最后，去除结果中的无用状态，使得 $2^{\wedge}Q$ 变更为 Q_D，υ'变为 υ_D，F'变为 F_D。这样就得到最简 DFA $(Q_D, \Sigma, \upsilon_D, \{q_0\}, F_D)$。此过程进一步优化为子集构造法，如算法 2.3 所示。

算法 2.3 NFA 等价转为 DFA 的子集构造法。

输入： NFA $N = (Q, \Sigma, \upsilon, q_0, F)$。

输出： DFA $A = (\mathbb{Q}, \Sigma, \text{move}[], \{q_0\}, \{S \in \mathbb{Q} \mid S \cap F \neq \varnothing\})$。

```
ℚ = Ø
move[] = NIL
将{q0}加入 ℚ 且未标记
while (ℚ 有一个未标记元素 S) {
    标记 S
    for (a∈Σ) {
        T = ∪q∈S · υ(q, a)
        if (T ∉ ℚ)T 加入 ℚ 中且未标记
        move[S, a] = T}
}
```

算法构建的 $\mathbb{Q}$ 是 A 状态集，而 A 的每个状态都用 Q 的一个子集来表示，这是为了借助于 N 转移函数构建 A 的转移函数。对于构建的转移表，在算法中使用一个二维表格 move[S, a]来表示，它实际上是 A 的转移表。该算法的主要任务是构造此表。A 的初始状态是确定的，即仅由 N 初始状态构成的单元素集合。而接受状态是那些与 N 接受状态集合 F 有交集的 A 状态。

算法中使用集合 $\mathbb{Q}$ 及其元素标记功能和查找功能实现无遗漏填表 move[]，并借此控制 while 循环正常结束。具体来说，算法很好地模拟了惰性子集法的计算过程，即先将 A 初始状态{q0}加入 $\mathbb{Q}$ 并设为未标记，然后重复对 $\mathbb{Q}$ 中一个未标记元素 S 进行迭代计算，首先将 S 设置标记，并计算它的标记为 a 的转移弧射入状态集合 T，得到 A 的 S

状态经 a 弧转移到 T 状态，并将此转移关系记录到 move[]中，同时检查 T 若不在 $\mathbb{Q}$ 中则加入以便后续计算从它到另一 A 状态的转移关系。对字母表中每个符号 a 都进行上述处理，对应于算法中的 for 循环。

算法中一旦 while 循环结束，则算法结束，并返回 A 的转移表。当然在这个 move[]表中还需指出初始状态和接受状态，那就是容易的事了。

定理 2.2　对于 NFA $N=(Q_N,\Sigma,\upsilon_N,q_0,F_N)$，采用子集构造法得到 DFA $D=(Q_D,\Sigma,\upsilon_D,\{q_0\},F_D)$，那么 $L(D)=L(N)$。

证明：依照 w 长度归纳证明命题 $\tilde{\upsilon}_D(\{q_0\},w)=\tilde{\upsilon}_N(q_0,w)$。

基础：对于 $w=\varepsilon$。根据 DFA 和 NFA 的定义，$\tilde{\upsilon}_D(\{q_0\},\varepsilon)=\tilde{\upsilon}_N(q_0,\varepsilon)=\{q_0\}$。

归纳：假定归纳假设，对于短于 w 的串命题成立，令 $w=xa$，$a\in\Sigma$，即有

$\tilde{\upsilon}_D(\{q_0\},x)=\tilde{\upsilon}_N(q_0,x)$

由 NFA 的扩展转移函数知

$\tilde{\upsilon}_N(q_0,xa)=\bigcup p\in\tilde{\upsilon}_N(q_0,x)\cdot\upsilon_N(p,a)$

由子集构造法知

$\upsilon_D(\tilde{\upsilon}_N(q_0,x),a)=\bigcup p\in\tilde{\upsilon}_N(q_0,x)\cdot\upsilon_N(p,a)$

根据以上两式有

$\tilde{\upsilon}_D(\{q_0\},xa)=\upsilon_D(\tilde{\upsilon}_D(\{q_0\},x),a)=\upsilon_D(\tilde{\upsilon}_N(q_0,x),a)=\bigcup p\in\tilde{\upsilon}_N(q_0,x)\cdot\upsilon_N(p,a)$

由此得到 $\tilde{\upsilon}_D(\{q_0\},xa)=\tilde{\upsilon}_N(q_0,xa)$的结论，因此命题成立。

注意，D 和 N 都接受 w 当且仅当 $\tilde{\upsilon}_D(\{q_0\},w)$和 $\tilde{\upsilon}_N(q_0,w)$各自都包含一个 F_N 中的状态，由此得 $L(D)=L(N)$。

定理 2.3　语言 L 为某个 DFA 定义，当且仅当 L 为某个 NFA 定义。

证明：（当）由子集构造法和定理 2.2 得证。

（仅当）将 DFA $D=(Q,\Sigma,\upsilon_D,\{q_0\},F)$等价地转化为 NFA $N=(Q,\Sigma,\upsilon_N,q_0,F)$，如果 $\tilde{\upsilon}_D(q,a)=p$，那么令 $\tilde{\upsilon}_N(q,a)=\{p\}$。采用归纳证明 D 和 N 是等价的，即基于 $w\in\Sigma^*$长度归纳，如果 $\tilde{\upsilon}_D(q,w)=p$，那么 $\tilde{\upsilon}_N(q,w)=\{p\}$。

把这部分的证明留给读者。结论是：D 接受 w 当且仅当 N 接受 w，即 $L(D)=L(N)$。

2.4　带 ε 转移的有穷自动机

就一个状态上的射出弧而言，DFA 允许同标记的至多一条，而 NFA 允许同标记的可以有多条，因此，它的状态转移有不确定性。除此之外，NFA 还有一种不确定

性源自 ε 转移，指不需要输入符号就发生状态转移。ε 转移丰富了 NFA 的不确定性内涵，允许 ε 转移的 NFA 记为 ε-NFA，是更为有用的模型，但仍然与不带 ε 转移的 NFA 等价。

2.4.1 ε-NFA 的定义

除了转移函数中考虑 ε 转移外，ε-NFA 的定义与 NFA $(Q, \Sigma, \upsilon, q_0, F)$的定义没有更多区别，可设想为将 ε 转移添加到 NFA 的转移函数中而得

$$\upsilon: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^{\wedge} Q$$

ε 作为转移函数的第二个参数时，特别表示 ε 转移。ε 专用于指代空串且 $\varepsilon \in \Sigma^*$，所以 ε 不是 Σ 中的符号，也就是说，在构建字母表时考虑避开使用 ε。转移函数的其他参数取值与 NFA 无异。

例 2.13 给出接受十进制定点数的 ε-NFA。

把由十进制数字和一个小数点组成的数称为无符号定点数，这里对于小数点的出现做了限定，为必须出现，即小数点可以位于数的最左端、数字串的中间或数的最右端。带有正负号的无符号定点数或者无符号定点数本身都是定点数。

状态图如图 2-19 所示。这个 ε-NFA 在 q_1 状态时已消耗掉正负号，如果有的话。经过 q_2 的路径表明小数点在数字前部或中间，而经过 q_3 的路径表明小数点在数字中间或最右端。这个转移图使用了 ε 转移，分别是 q_0 转移到 q_1，q_4 转移到 q_5。前一个 ε 转移为了表达不带正负号的数，后一个 ε 转移表达这个事实即到达 q_4 如同到达 q_5，实际上这一个 ε 转移是多余的，只要将 q_4 设为接受状态，q_5 状态可以省去。冗余的设计提示优化设计的可能性。

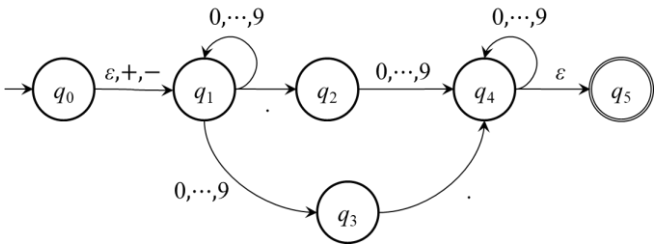


图 2-19 十进制定点数 ε-NFA

除了用转移图表示外，还可以用转移表和代数表示。转移表表示如表 2-6 所示。代数表示在后文描述。

表 2-6　定点数转移表表示

	+, −	.	0, 1, ⋯, 9	ε
$\rightarrow q_0$	$\{q_1\}$	Ø	Ø	$\{q_1\}$
q_1	Ø	$\{q_2\}$	$\{q_1, q_4\}$	Ø
q_2	Ø	Ø	$\{q_3\}$	Ø
q_3	Ø	Ø	$\{q_3\}$	$\{q_5\}$
q_4	Ø	$\{q_3\}$	Ø	Ø
$^*q_5$	Ø	Ø	Ø	Ø

设置表的 ε 列来表示 ε 转移，其余列含义与 NFA 相同。如同转移图的一条弧上多个标记等价于平行的多个单标记弧一样，表中也有多列合并展示，都是在不会引起混淆的前提下，为了追求更为简洁的表达所做的变通。

对应于图 2-19，写出代数表示形式：ε-NFA $(Q, \Sigma, \upsilon, q_0, \{q_5\})$

其中：

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$

$\Sigma = \{+, -, ., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$\upsilon = \{(q_0, +, \{q_1\}), (q_0, -, \{q_1\}), (q_0, \varepsilon, \{q_1\}), (q_1, ., \{q_2\}), \cdots\}$

ε-NFA 的转移图、转移表和代数表示是等价的，都有各自最为适合的应用场合，可类比于 NFA、DFA 的情形。

ε 转移在一些场合非常有用，因为它带来表达上的方便性。比如识别金额和账号这样的问题，已经有了识别金额的自动机（不管是 DFA、NFA，还是 ε-NFA），也有了识别账号的自动机，在此基础上，只需要利用 ε 转移将二者集成为一个 ε-NFA：

	$a \in \Sigma$	ε
$\rightarrow r_0$	r_0	$\{q_0, p_0\}$

<账号自动机的转移表>　//初始状态 q_0 不再是，接受状态（可多个）仍是

<金额自动机的转移表>　//初始状态 p_0 不再是，接受状态（可多个）仍是

集成起来的 ε-NFA，对于已有的账号自动机和金额自动机仍然保持完整，只是原有初始状态不再是初始状态，但接受状态仍然保持不变。这个 ε-NFA 在输入符号串上的运行过程可描述为并行地经历三类线索：识别为账号的线索，识别为金额的线索，以及不是账号也不是金额而拒绝的线索。如果输入串是账号或金额，依照它们的接受状态做出结论，如果不是则拒绝。

回顾例 2.7，如果打算构造一个 ε-NFA 识别“或者包含偶数个 0 或者包含奇数个 1

的串”，同时已经有了识别“偶数个 0 的串”的 DFA 和识别“奇数个 1 的串”的 DFA，那么这个 ε-NFA 就很容易设计，结果如下：

	0	1	ε
$\rightarrow q_0$	Ø	Ø	$\{e_0, o\}$
$*e_0$	$\{e\}$	$\{e_0\}$	Ø
e	$\{e_0\}$	$\{e\}$	Ø
o	$\{o\}$	$\{o_1\}$	Ø
$*o_1$	$\{o_1\}$	$\{o\}$	Ø

对于 0-1 输入串，永远接受，并根据接受状态可知其为哪一种。

2.4.2 状态的 ε 闭包

在 ε-NFA 中，从某状态出发只沿 ε 转移就可达的所有状态称为这个状态的 ε 闭包。状态 q 的 ε 闭包记为 $\omega(q)$。换言之，$\omega(q)$指 q 自身以及以 q 为始端且以 ε 为标记的任意路径的末端之集合。总之，ε 转移过程中不消耗输入。

定义 2.10 ε-NFA $(Q, \Sigma, \upsilon, q_0, F)$的状态的 ε 闭包 $\omega(q)$，$q \in Q$：

基础：$q \in \omega(q)$。

归纳：若 $p \in \omega(q)$且从 p 有 ε 转移弧射出到状态 r，即 $r \in \upsilon(p, \varepsilon)$，那么 $r \in \omega(q)$。

由定义中的归纳部分看到，从 q 经连续 ε 弧所到达的状态包含在 q 的 ε 闭包中，计算 ε 闭包的算法强调了这一计算过程。

算法 2.4 状态的 ε 闭包。

输入：ε-NFA $E = (Q, \Sigma, \upsilon, q_0, F)$，$q \in Q$。

输出：$S = \omega(q)$。

```
1   S = {q}
2   while(S 中有未标记的元素 p) {
3       标记 S 中的这个元素 p
4       T = υ(p, ε)
5       for(r∈T) if(r∉S)  将 r 加入 S 中}
```

行 1 对应于定义的基础步，行 2 至行 6 对应于定义的归纳步。算法中，S 从最初包含一个 q 状态，到把 S 中状态经过连续 ε 弧转移到的状态都包含进去，采用的是不动点算法框架，即 S 中每一状态做 ε 转移并添加到 S，直到 S 不再变化为止，所得 S 就是结果。

例 2.14　给定一个 ε-NFA：

	0	1	ε
$\rightarrow q_0$	Ø	$\{q_1\}$	$\{q_1\}$
q_1	$\{q_0, q_1\}$	Ø	$\{q_2\}$
$^*q_2$	Ø	Ø	Ø

求各状态的 ε 闭包。模拟不动点算法的计算过程。

基于归纳基础步进行初始化：$\omega(q_0)=\{q_0\}$；$\omega(q_1)=\{q_1\}$；$\omega(q_2)=\{q_2\}$

归纳步第一遍更新：$\omega(q_0)=\{q_0, q_1\}$；$\omega(q_1)=\{q_1, q_2\}$；$\omega(q_2)=\{q_2\}$

归纳步第二遍更新：$\omega(q_0)=\{q_0, q_1, q_2\}$；$\omega(q_1)=\{q_1, q_2\}$；$\omega(q_2)=\{q_2\}$

归纳步第三遍更新：因无变化则最新值即为结果。

状态的 ε 闭包概念推广到状态集合的 ε 闭包，仍使用函数 $\omega()$计算，可理解为它是一个多态函数，计算结果为这个集合中的每一个状态的 ε 闭包的并集，即把集合 S 的 ε 闭包写为 $\omega(S)=\cup p\in S\cdot\omega(p)$。对于例 2.14，有

$$\omega(\{q_1, q_2\})=\cup p\in\{q_1, q_2\}\cdot\omega(p)=\omega(q_1)\cup\omega(q_2)=\{q_1, q_2\}\cup\{q_2\}=\{q_1, q_2\}$$

理论上会有这样的计算任务：

$$\forall S\in 2\hat{}Q\cdot\omega(S)=\cup q\in S\cdot\omega(q)$$

其复杂度是指数的，实际中需要进行优化处理。

在求集合的 ε 闭包过程中，常常会遇到一种现象，就是求 ε 闭包前后结果不变，即对于状态集合 S，$\omega(S)=S$。这样的状态集合被称为 ε 闭集。例 2.14 的$\{q_1, q_2\}$是 ε 闭集，此外还有$\{q_2\}$和$\{q_0, q_1, q_2\}$都是 ε 闭集。显然，对任意状态集合 S，$\omega(S)$是 ε 闭集，因为 $\omega(\omega(S))=\omega(S)$。$\varepsilon$ 闭集的这个性质可用于检查 ε 闭包的计算结果，无论是状态的还是状态集合的 ε 闭包，对计算出来的结果再求一次 ε 闭包，如果没有变化则知其为正确的结果。

ε闭集概念扩展了活动状态集概念，每个活动状态集都是ε闭集。让ε闭集用于ε-NFA 判定性质，使用归纳思想来表达运行输入串 w 得出判定结论。

（基础）已消耗串为 ε，剩余串等于输入串 w，活动状态集为 $\omega(\{q_0\})$。依次初始化并行转移步。

（归纳）假设经过 n 个并行转移步，已消耗串为 x，剩余串为 ay，活动状态集为 S。那么再一个并行转移步，已消耗串为 xa，剩余串为 y，活动状态集为 $\omega(\cup p\in S\cdot v(p, a))$。由于每一并行转移步消耗一个输入符号，第$|w|$个并行转移步过后，剩余串 $y=\varepsilon$，活动状态集为 S，依据 $S\cap F\neq\emptyset$ 判定为接受，否则拒绝。

2.4.3 ε-NFA 的扩展转移函数

与 NFA 一样，为了方便描述连续的状态转移过程，扩展转移函数 $\upsilon(q,a)$为 $\tilde{\upsilon}(q,w)$，表示一个活动状态集，是从初始化活动状态集 $\omega(\{q\})$到经过|w|个并行转移步骤同步消耗完 w 后的那个活动状态集。

ε-NFA 的扩展转移函数是在 NFA 的扩展转移函数基础上考虑了 ε 转移，因此，每一步计算都要考虑计算 ε 闭包。ε 闭包对于构造扩展转移函数而言是必要的。

定义 2.11 ε-NFA $(Q, \Sigma, \upsilon, q_0, F)$的扩展转移函数 $\tilde{\upsilon}()$是一个类型为 2^Q×Σ^*→2^Q 的函数，并且

基础： $\tilde{\upsilon}(q,\varepsilon)=\omega(\{q\})$。不读入任何符号，经连续 ε 弧所能到达的状态，包括自身。

归纳： 设 $w=xa$，$a\in\Sigma$，$\tilde{\upsilon}(q,x)$已知，那么

$$\tilde{\upsilon}(q,w)=\omega(\cup p\in\tilde{\upsilon}(q,x)\cdot\upsilon(p,a))$$

也就是说，根据归纳假设，已知 $\tilde{\upsilon}(q,x)$是一个状态集合，那么这个集合中的每一个状态射出的带有标记 a 的弧所射入的状态全集就是 $\tilde{\upsilon}(q,wa)$，其中包括连续 ε 弧到达的状态在内。可见扩展转移函数的返回值是一个 ε 闭集。所以我们对集合 $\tilde{\upsilon}(q,w)$在每一个状态 p 计算出 $\upsilon(p,a)$之后立即求 ε 闭包，所得结果的并集就是 $\tilde{\upsilon}(q,wa)$。还有另一种计算途径就是对集合 $\tilde{\upsilon}(q,w)$在每一个状态 p 计算出 $\upsilon(p,a)$，对所得结果的并集求 ε 闭包作为 $\tilde{\upsilon}(q,wa)$的值。后一种方法为定义 2.10 所采用。这两种计算的结果相同，所以在后文任意使用其中一种计算方式。

从活动状态集概念来解释扩展转移函数 $\tilde{\upsilon}(q,w)$，初始化活动状态集 $T_0=\tilde{\upsilon}(q,\varepsilon)=\omega(q)$，第一个并行转移步得到 $T_1=\tilde{\upsilon}(q,\pi(w,1))$，第 i 个并行转移步 $T_i=\tilde{\upsilon}(q,\pi(w,i))$，直到第|$w$|并行转移步 $T_{|w|}=\tilde{\upsilon}(q,\pi(w,|w|))=\tilde{\upsilon}(q,w)$即为结果。相邻两个并行步之间，即 T_{i-1} 与 T_i 之间的关系是 $T_i=\cup p\in T_{i-1}\cdot\upsilon(p,w_i)$，其中 $1\leqslant i\leqslant|w|$。也就是说，对于前一步的活动状态集中的每个状态，经过标记为 w_i 的射出弧所能到达(包括再经连续 ε 弧所能到达)的每个状态，这些状态组成后一步的活动状态集。

回顾例 2.13 中十进制数的例子，计算 $\tilde{\upsilon}(q_0,-2.6)$，采用递推方式的计算过程可以连续推导下去，上下文关系清晰。

$$\begin{aligned}
&\tilde{\upsilon}(q_0,-2.6)=\omega(\cup p\in\tilde{\upsilon}(q_0,-2.)\cdot\upsilon(p,6))\\
&=\omega(\cup p\in[\omega(\cup p\in\tilde{\upsilon}(q_0,-2)\cdot\upsilon(p,.))]\cdot\upsilon(p,6))\\
&=\omega(\cup p\in[\omega(\cup p\in[\omega(\cup p\in\tilde{\upsilon}(q_0,-)\cdot\upsilon(p,2))]\cdot\upsilon(p,.))]\cdot\upsilon(p,6))\\
&=\omega(\cup p\in[\omega(\cup p\in[\omega(\cup p\in[\omega(\cup p\in\tilde{\upsilon}(q_0,\varepsilon)\cdot\upsilon(p,-))]\cdot\upsilon(p,2))]\cdot\upsilon(p,.))]\cdot\upsilon(p,6))\\
&=\omega(\cup p\in[\omega(\cup p\in[\omega(\cup p\in[\omega(\cup p\in\{q_0,q_1\}\cdot\upsilon(p,-))]\cdot\upsilon(p,2))]\cdot\upsilon(p,.))]\cdot\upsilon(p,6))
\end{aligned}$$

$$= \omega(\cup p \in [\omega(\cup p \in [\omega(\cup p \in \{q_1\} \cdot \upsilon(p, 2))] \cdot \upsilon(p, .))] \cdot \upsilon(p, 6))$$
$$= \omega(\cup p \in [\omega(\cup p \in [\omega(\{q_1, q_4\})] \cdot \upsilon(p, .))] \cdot \upsilon(p, 6))$$
$$= \omega(\cup p \in [\omega(\cup p \in \{q_1, q_4\} \cdot \upsilon(p, .))] \cdot \upsilon(p, 6))$$
$$= \omega(\cup p \in [\omega(\{q_2, q_3\}] \cdot \upsilon(p, 6))$$
$$= \omega(\cup p \in \{q_2, q_3, q_5\} \cdot \upsilon(p, 6))$$
$$= \omega(\{q_3\})$$
$$= \{q_3, q_5\}$$

采用归纳方式计算，依次计算出中间结果，最后得到最终结果，逻辑严密。

$$\tilde{\upsilon}(q_0, \varepsilon) = \omega(\{q_0\}) = \{q_0, q_1\}$$
$$\tilde{\upsilon}(q_0, -) = \omega(\cup p \in \tilde{\upsilon}(q_0, \varepsilon) \cdot \upsilon(p, -)) = \omega(\cup p \in \{q_0, q_1\} \cdot \upsilon(p, -)) = \{q_1\}$$
$$\tilde{\upsilon}(q_0, -2) = \omega(\cup p \in \tilde{\upsilon}(q_0, -) \cdot \upsilon(p, 2)) = \omega(\{q_1, q_4\}) = \{q_1, q_4\}$$
$$\tilde{\upsilon}(q_0, -2.) = \omega(\cup p \in \tilde{\upsilon}(q_0, -2) \cdot \upsilon(p, .)) = \omega(\{q_2, q_3\}) = \{q_2, q_3, q_5\}$$
$$\tilde{\upsilon}(q_0, -2.6) = \omega(\cup p \in \tilde{\upsilon}(q_0, -2.) \cdot \upsilon(p, 6)) = \omega(\{q_3\}) = \{q_3, q_5\}$$

利用扩展转移函数定义 ε-NFA $E = (Q, \Sigma, \upsilon, q_0, F)$ 的语言：

$$L(E) = \{w \mid w \in \tilde{\upsilon}(q_0, w) \cap F \neq \varnothing\}$$

2.4.4　消除 ε 转移

假若去除 ε-NFA $(Q, \Sigma, \upsilon, q_0, F)$中的 ε 弧，结果是一个等价的 NFA $(Q', \Sigma, \upsilon', q_0, F')$，二者的语言相同，当且仅当

$$\upsilon'(q, a) = \omega(\cup p \in \omega(q) \cdot \upsilon(p, a)), q \in Q, a \in \Sigma$$

并且 Q'和 F'既不含从初始状态不可达的状态也不含不能到达接受状态的状态。

例 2.15　将下列 ε-NFA 转换为 NFA。

	0	1	ε
$\rightarrow q_0$	Ø	$\{q_1\}$	$\{q_1\}$
q_1	$\{q_0, q_1\}$	Ø	$\{q_2\}$
$*q_2$	Ø	Ø	Ø

转换结果中不再有 ε 转移，并且不改变所定义的语言。结果如下：

	0	1
$\rightarrow q_0$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$
q_1	$\{q_0, q_1, q_2\}$	Ø
$*q_2$	Ø	Ø

对一个 ε-NFA 在表示上进行优化是可能的，当然不改变所定义的语言。如果以下两个步骤至少有一个得到执行，就表明产生了更为优化的结果。

(1) 将由 ε 弧组成的单一路径（标记为 ε 的单一路径）用一个 ε 弧取代，该弧从路径始端直至末端。所谓单一路径是指除了端点外的节点只有唯一射出弧和唯一射入弧。

(2) 如果单一路径的末端没有射出弧，那么末端节点可以合并到始端节点。如果始端节点没有射入弧那么可以合并到末端节点。但是可以合并的前提是两个状态都是同质的，都是接受状态，或者都是非接受状态。

(3) 经 ε 弧到达原接受状态的所有状态都被更改为接受状态并删除当事 ε 弧。

作为例子，对图 2-19 实施优化步骤(3)的话，q_4 变成接受状态。接着发现 q_4 和 q_5 是同质的且 q_5 没有射出弧，所以进一步按照优化步骤(2)删除 q_5。结果是一个优化的并且等价的 ε-NFA，优化效果是自动机更小了。

给定一个 ε-NFA E 可以找到一个 DFA D，使得 $L(E)=L(D)$。这是 ε-NFA 的等价性质所提供的可能性。那么，仍然采用一种基于 E 的状态子集的构造方法，其中对于 ε 转移采用 ε 闭包处理。

给定 ε-NFA (Q,Σ,v,q_0,F)那么存在一个等价的 DFA 如下所示：

$$\text{DFA}\ (2\text{^}Q,\Sigma,v_D,\omega(q_0),\{S\in 2\text{^}Q \mid S\cap F\neq\varnothing\})$$

$$\forall a\in\Sigma,S\in 2\text{^}Q\cdot v_D(S,a)=\omega(\cup p\in S\cdot v(p,a))$$

可以看出，式子中的 S 就是 ε-NFA 的所有可能的活动状态集。

例 2.16 将例 2.13 的十进制定点数 ε-NFA 等价转化为 DFA A。

改进的子集构造过程是惰性求值技术，从 A 的初始状态 $\omega(q_0)$开始逐行构建 A 的状态转移表，这个表初始化为 1 行$|\Sigma|+1$ 列，第一列是 A 的初始状态，而其余列对应于 Σ 中的每一个符号：

	$+,-$	.	$0,1,\cdots,9$
$\rightarrow\{q_0,q_1\}$			

设该行为当前行，那么重复进行如下过程直到表格中内容不再发生变化为止。结果就得到 A 的状态转移表。

迭代的过程是：根据当前行的第一列内容 S 和其余各列对应的输入符号 a 计算当前行各列元素，为 $\omega(\cup p\in S\cdot v(p,a))$；检查计算出的元素没有在表格第一列中出现过，若没有则给表尾添加新的一行，并将该元素填入第一列。

当一行各元素都计算完成时，就接着如法炮制计算下一行，依次类推。第一行计算完成时，有

	$+,-$	$.$	$0,1,\cdots,9$
$\rightarrow\{q_0,q_1\}$	$\{q_1\}$	$\{q_2\}$	$\{q_1,q_4\}$
$\{q_1\}$			
$\{q_2\}$			
$\{q_1,q_4\}$			

这是计算完成时所得 A 的转移表，其中用 Q 子集表示 A 的一个状态：

	$+,-$	$.$	$0,1,\cdots,9$
$\rightarrow\{q_0,q_1\}$	$\{q_1\}$	$\{q_2\}$	$\{q_1,q_4\}$
$\{q_1\}$	$\varnothing$	$\{q_2\}$	$\{q_1,q_4\}$
$\{q_2\}$	$\varnothing$	$\varnothing$	$\{q_3,q_5\}$
$\{q_1,q_4\}$	$\varnothing$	$\{q_2,q_3,q_5\}$	$\{q_1,q_4\}$
$\{q_3,q_5\}$	$\varnothing$	$\varnothing$	$\{q_3,q_5\}$
$\{q_2,q_3,q_5\}$	$\varnothing$	$\varnothing$	$\{q_3,q_5\}$

容易得出子集构造过程的算法描述。

算法 2.5　ε-NFA 到 DFA 的子集构造算法。

输入：ε-NFA $E=(Q,\Sigma,\upsilon,q_0,F)$。

输出：DFA $D=(\mathbb{Q},\Sigma,\text{move}[],\omega(q_0),\{S\in\mathbb{Q}\mid S\cap F\neq\varnothing\})$。

$\mathbb{Q}=\varnothing$; move[] = NIL
将 $\omega(\{q_0\})$加入 $\mathbb{Q}$ 且未标记
while $\mathbb{Q}$ 中存在一个未标记元素 S
{　标记 S
　for($a\in\Sigma$) {
　　$T=\omega(\bigcup q\in S\cdot\upsilon(q,a))$
　　if($T\notin\mathbb{Q}$) 将 T 加入 $\mathbb{Q}$ 中且未标记
　　move[S,a] = T
}

算法中 $\mathbb{Q}$ 表示 E 状态 Q 的幂集的子集，其元素用于表示 D 的状态，以便于借助于 E 转移函数构建 D 状态间的转移关系。对于 D 的转移表，使用了一个二维表格 move[S, a]来表示，它实际上是 D 的转移表，算法的主要任务是构造此表。D 的初始状态是确定的，即 E 初始状态的 ε 闭包。而接受状态是那些与 E 接受状态集合 F 有交集的 D 状态。

算法中使用集合 $\mathbb{Q}$ 及其元素标记功能和查找功能实现填表 move[]无遗漏，并借此控制 while 循环正常结束。具体来说，算法很好地模拟了惰性子集法的计算过程，即先将 D 初始状态$\{q_0\}$的 ε 闭包，即 $\omega(\{q_0\})$加入 $\mathbb{Q}$ 并设为未标记，然后对 $\mathbb{Q}$ 中一个未标记元素 S 进行循环计算，即将 S 设置标记，并计算它的标记为 a 的转移弧射入状态集合 T，得到 D 的 S 状态经 a 弧转移到 $\omega(T)$状态，并记录 move$[S, a] = \omega(T)$到表格 move[]中，同时检查 $\omega(T)$，若其不在 $\mathbb{Q}$ 中则加入以便后续计算从它到 D 的另一状态的转移。对字母表中每个符号 a 都进行上述处理，对应于 for 循环。

一旦 while 循环结束，算法结束，得到了 D 的转移表 move[]。当然在这个 move[]表中还需指出初始状态和接受状态，那就容易了。

可以看出，例 2.16 展示的是算法执行过程。对于子集法需要理论证明其正确性。

定理 2.4 语言 L 被某个 ε-NFA 接受，当且仅当其被某个 DFA 接受。

证明：（当）证明是基于对 DFA $D = (Q, \Sigma, \upsilon_D, q_0, F)$构造等价的 ε-NFA $E = (Q, \Sigma, \upsilon_E, q_0, F)$，这是容易做到的，因为只需要定义 υ_E。

对于 $q \in Q_D$，$a \in \Sigma$，若有 $\upsilon_D(q, a) = p$，则令 $\upsilon_E(q, a) = \{p\}$。

对于 $q \in Q_D$，令 $\upsilon_E(q, \varepsilon) = \emptyset$。

（仅当）证明是基于对 ε-NFA $E = (Q_E, \Sigma, \upsilon_E, q_E, F_E)$ 采用子集法构造等价的 DFA $D = (Q_D, \Sigma, \upsilon_D, q_D, F_D)$。$E$ 和 D 的等价性也就是 $\tilde{\upsilon}_E(q_E, w) = \delta_D(q_D, w), w \in \Sigma^*$。采用归纳法，有

基础：对于$|w| = 0$，即 $w = \varepsilon$，根据子集构造法和 D 的初始状态的定义有 $q_D = \omega(q_E)$，而根据扩展转移函数和 ε 闭包的定义有 $\tilde{\upsilon}_E(q_E, \varepsilon) = \omega(q_E)$。因此

$$\tilde{\upsilon}_D(q_D, \varepsilon) = q_D = \omega(q_E) = \tilde{\upsilon}_E(q_E, \varepsilon)$$

归纳：假设该命题对于 $x \in \Sigma^*$成立，即有 $\tilde{\upsilon}_E(q_E, x) = \tilde{\upsilon}_D(q_D, x) = S$，那么当 $w = xa$ 时，令 $\tilde{\upsilon}_E(q_E, x) = \tilde{\upsilon}_D(q_D, x) = S$，根据扩展转移函数定义有 $\tilde{\upsilon}_E(q_E, w) = \tilde{\upsilon}_E(q_E, xa) = \omega(\bigcup p \in S \cdot \upsilon_E(p, a))$。

然而，根据子集构造法，采用 Q_E 子集作为 D 的状态，且 D 中有从状态 S 经标记为 a 的弧到达状态 $\omega(\bigcup p \in S \cdot \upsilon_E(p, a))$，即 $\upsilon_D(S, a) = \omega(\bigcup p \in S \cdot \upsilon_E(p, a))$。因此

$$\tilde{\upsilon}_D(q_D, w) = \tilde{\upsilon}_D(q_D, xa) = \upsilon_D(\tilde{\upsilon}_D(q_D, x), a) = \upsilon_D(\tilde{\upsilon}_E(q_E, x), a) = \upsilon_D(S, a)$$

$$= \omega(\bigcup p \in \tilde{\upsilon}_E(q_E, x) \cdot \upsilon_E(p, a)) = \tilde{\upsilon}_E(q_E, xa) = \tilde{\upsilon}_E(q_E, w)$$

证明完毕。

2.5　习题

习题 2.1　分别写出字母表{0, 1}上下列语言的 DFA：

(1) 有符号的二进制整数，不含前 0。

(2) 无符号二进制定点数，不含后 0。

(3) 带有偶数个 0 做子串的串的集合。

(4) 串中 0 的个数是 3 的串的集合。

习题 2.2　设 A 是一个 DFA，q 是 A 的一个特定状态，使得对于所有输入符号 a，$\upsilon(q, a) = q$。通过对输入串长度进行归纳，证明：对所有输入 w，$\tilde{\upsilon}(q, w) = q$。

习题 2.3　设 A 是一个 DFA，a 是 A 的一个输入符号，使得对于 A 的所有状态 q，有 $\upsilon(q, a) = q$。

(1) 通过对 n 进行归纳，证明：对所有 $n \geqslant 0$，$\tilde{\upsilon}(q, a^n) = q$，其中 a^n 是由 n 个 a 组成的串。

(2) 证明：要么$\{a\}^* \subseteq L(A)$，要么$\{a\}^* \cap L(A) = \varnothing$。

习题 2.4　考虑 DFA $(\{A, B\}, (0, 1), \{(A, 0, A), (A, 1, B), (B, 0, B), (B, 1, A)\}, A, \{B\})$，描述这个 DFA 的语言，通过对输入串的长度进行归纳，证明该描述是正确的。

习题 2.5　将下列 NFA 转换为 DFA：

	0	1
$\rightarrow s$	$\{s\}$	$\{s, q\}$
q	$\{p\}$	$\{p\}$
p	$\{\}$	$\{r\}$
$*r$	$\{p\}$	$\{p\}$

习题 2.6　考虑下列 ε-NFA：

	a	b	c	ε
$\rightarrow q$	$\{\}$	$\{q\}$	$\{p\}$	$\{r\}$
p	$\{q\}$	$\{p\}$	$\{r\}$	$\{\}$
$*r$	$\{p\}$	$\{r\}$	$\{\}$	$\{q\}$

(1) 计算每个状态的 ε 闭包。

(2) 给出这个自动机所接受的长度不大于 3 的串。

(3) 把这个自动机转换为 DFA。

习题 2.7 已知 NFA 如转移表所示：

	0	1	ε
→1			{2, 6}
2	{3}		{1}
3			{4}
4		{5}	
5			{2, 8}
6	{7}		
7			{8}
*8			

(1) 计算状态 5 的 ε 闭包。

(2) 用自然语言描述这个自动机所识别的语言。

(3) 把这个自动机转换为 DFA。

习题 2.8 为下列语言设计 ε-NFA，并转换为最小 DFA：

(1) 由 0 到多个 a，后面跟 0 到多个 b，再后面跟 0 到多个 c 的串的集合。

(2) 包含着 10 重复一次到多次或包含 101 重复一次到多次的串的集合。

(3) 使得 3 后缀至少含有一个 1 的 0-1 串。