第 2 章

程式的意義

用心體會，但不要思考！這就像用手指指向月亮，
如果將精神集中在手指，你將會錯過所有絕妙的榮耀。

— 李小龍

程式語言和以程式語言所編寫的程式，是我們作為軟體工程師工作的基礎。我們利用它們向我們自己釐清複雜的想法，向彼此溝通那些想法，最重要的是在我們的電腦內部實作那些想法。就如同人類社會沒有自然語言就無法運作，因此全球的程式設計者社群都依賴程式語言來傳達並實作我們的想法，他們會以每個成功的程式作為基礎，然後再在這些基礎建構下一層的想法。

程式設計者往往是現實的生物。我們學習新的程式語言常常是利用閱讀文件、依循教程、研究現有的程式、亂改自己的簡單程式，而且不會過度考慮那些程式的意義。學習的過程有時候很像就在不斷的摸索、嘗試錯誤：我們試著要檢視範例和文件來瞭解語言，然後試著以它來編寫一些程式，接著這一切卻又煙消雲散，使得我們必須回頭再試，一直到我們能夠統整大部分的工作。隨著電腦和電腦所支援的系統變得越來越複雜，往往讓人將程式視為只能表示它們自己而且只是偶然可以運作的暗黑咒語。

然而電腦程式設計的關鍵並非**程式**，它的核心其實是**想法**。程式展現的是凍結的想法、曾經存在程式設計者想像力裡的結構快照。程式的價值僅在於編寫，因為這樣才有意義。因此，是什麼連接了程式碼及其意義？而我們該如何更具體的賦予程式意義，而不只是嚷著『它只是做它所做的』？在本章，我們將檢視幾種技術，除了釐清電腦程式的意義，並且也瞭解如何將那些死的快照變成活的。

意義的『意義』

語言學裡的**語意學**（*semantic*）專門在研究字詞及其意義之間的關聯，例如：『狗』這個字是頁面上外形的約定，或者是某人聲帶造成的一組空氣震動；這些與實際的狗或一般對狗的想法截然不同。語意關注的是這些具體的意符如何關聯到它們的抽象意義及抽象意義本身的基本性質。

在電腦科學的領域，**形式語意**（*formal semantic*）涉及的是程式難以捉摸之意的尋求之道，以及利用它們發現或證明程式語言的趣事。形式語意的用途廣泛，從像是指定新語言和設計編譯器最佳化的具體應用，一直到諸如建構程式正確性的數學證明的抽象概念。

要完全具體詳述某種程式語言，我們需要提供兩件事：一是描述程式外觀的**語法**（*syntax*），再者是描述程式意義的語意[1]。

許多語言並沒有官方的書面規格，只有能運作的直譯器或編譯器。Ruby 本身就屬於這種『源自實作物的規格』（specification by implementation）的類型：雖然有很多 Ruby 應該如何運作的書籍和教程，但這些資訊的最終來源都是 Matz 的 Ruby Interpreter（MRI，譯註：Matz 是 Ruby 的原始創造者），這是 Ruby 的參考實作物。如果有任何 Ruby 文件的隻字片語不同意 MRI 實際的行為，這份就是錯的文件。諸如 JRuby、Rubinius、MacRuby 之類的第三方 Ruby 實作物都必須竭力且完全模仿 MRI 的行為，這樣它們才敢聲稱與 Ruby 語言相容。其他像是 PHP 和 Perl 5 之類的語言，則在語言定義共用了這種以實作物主導（implementation-led）的作法。

撰寫官方的一般規格（prose specification，通常以英文撰寫），是另一種描述程式語言的方式，C++、Java、ECMAScript（JavaScript 的標準版本）就是這種作法的例子：語言的標準化是根據專家委員會所撰寫、與實作物無關（implementation-agnostic）的文件，而那些標準存在著許多相容的實作物。以官方文件規範語言，會比以參考實作物規範語言更為嚴格（設計決策更有可能是深思過、理性選擇的結果，而不是特定實作物的意外後果），但這種規格通常難以閱讀，而且也很難分辨其中是不是包含了矛盾、遺漏、模稜兩可。尤其並沒有形式的方式來論究英語的規格，面對這種規格，我們只能一讀再讀、不斷思索，然後希望能夠瞭解所有的推論。

---

1 在程式語言理論的語境（context）裡，通常將字詞的語意視為單數：我們藉由語意的賦予來描述語言的意義。

確實有一份 Ruby 1.8.7 的一般規格，而且甚至已經成為 ISO 標準（ISO/IEC 30170[2]）。雖然 MRI 依然被視為 Ruby 語言源自實作物的規格，但是 mruby 專案（*https://github.com/mruby/mruby*）也試著建構一個輕量、可嵌入的 Ruby 實作物，它的目標很明確：符合 ISO 標準，而非 MRI 相容性。

第 3 種選擇是使用形式語意的數學技巧來精準的描述程式語言的意義。這裡的目標是完全清楚，並且以適合的方法分析或甚至自動分析的格式來撰寫規格，以便全面檢查一致性、矛盾、或疏漏。在看過如何處理語法之後，我們將會檢視這些語意規格的形式之道。

語法

傳統的電腦程式是一長串字元。每種程式語言都有自己的一組規則，用來描述該種語言會將哪一種字元字串視為合法的程式；這些規則指明了語言的**語法**（*syntax*）。

語言的語法規則能讓我們從無意義的一群程式當中（例如 `>/;x:1@4`），分辨出可能合法的程式（像是 `y = x + 1`）。它們也提供了如何閱讀模稜兩可程式的實用資訊：舉例來說，有關運算子優先權的規則能自動決定應該將 `1 + 2 * 3` 視為 `1 + (2 * 3)`，而非視為 `(1 + 2) * 3`。

電腦讀取程式當然在預期之中，而讀取程式需要**解析器**（*parser*）：這是一種程式，它能讀取代表程式的整串字元，為了確定讀取的內容合法有效，便依照語法規則加以檢查，為了進一步的處理，再將它轉換成程式適合的結構化呈現。

有好幾種可以自動將語言的語法規則轉換成解析器的工具，但如何指定這些規則的細節，以及將它們轉換成有用的解析器的技巧，並非本章的重點（概觀的介紹請見第 61 頁『實作解析器』），但大致而言，解析器應該讀取像是 `y = x + 1` 的字串，並將它轉換成**抽象語法樹**（*abstract syntax tree*，*AST*），這是一種捨棄像是空白字元的附屬細節，並聚焦在程式階層架構的原始碼表示方式。

最後，語法只關心程式的外表，不在意它們的意義。程式有可能語法正確，但毫無意義和用途。舉例來說，程式 `y = x + 1` 本身並沒有意義，因為它並沒有事先說明 `x` 為何；而程式 `z = true + 1` 可能根本無法執行，因為它試著相加數值和布林值（當然這還取決於程式語言的其他屬性）。

2 雖然取得 ISO/IEC 30170 必須付費，但同一規格的早發草案則能從 *http://www.ipa.go.jp/osc/english/ruby/* 免費下載。

一如我們所預期，沒有『一種真正的方式』能解釋程式語言的語法要怎麼對應到潛在的意義。實際上有好幾種不同的方式，可以具體討論程式的意義，所有的方式都在形式、抽象、表現、實際效率之間有不同的折衷。我們將在後續幾節檢視幾種主要的形式之道，並且討論它們彼此的關係。

操作語意學

思考程式意義最實際的方式就是**它能做什麼**。當我們執行程式時，我們期待會發生什麼？程式語言裡的不同建構物在執行階段的行為會是如何？將它們接在一起而形成更大的程式，又會有什麼效果？

這就是**操作語意**（*operational semantic*）的基礎，一種擷取程式語言意義的方式，而這種方式是由程式如何在某種裝置執行的規則所定義。其中的裝置通常就是**抽象機**（*abstract machine*）：虛構、理想化的電腦，為了解釋語言的程式將會如何執行的特定目的而設計。為了精巧的擷取到它們執行階段的行為，不同類型的程式語言通常需要不同設計的抽象機。

賦予了操作語意，就能相當嚴格且精確面對語言裡特定建構物的目的。不同於英文撰寫的語言規格，那種規格可能隱含了模糊，並且忽略重要的邊緣案例；為了讓人信服語言行為的傳達效果，形式操作規格必須明確且不能模糊。

小步語意

那麼，我們要如何設計抽象機，並以它來指定程式語言的操作語意？方式之一是想像有種能直接在其語法操作估算程式的機器，以較少的步驟重複**化簡**（*reduce*）這部機器，不論結果的意義，使其每一步驟都讓程式更接近它的最終結果。

這些小步驟化簡類似我們在學校學的代數運算式計算？舉例來說，若要計算 (1 × 2) + (3 × 4)，我們知道應該：

1. 執行左側的乘法（1 × 2 變成 2），即可將運算式化簡成 2 + (3 × 4)
2. 執行右側乘法（3 × 4 變成 12），即可將運算式化簡成 2 + 12
3. 執行加法（2 + 12 變成 14），最後便得到 14

我們可以將 14 視為結果，因為以此過程無法再進一步化簡 14（我們認為 14 是代數運算式的特殊類型，也就是**值** [*value*]，它有自己的意義，而且不需要任何進一步處理）。

藉由寫下每個化簡小步驟進行方式的形式規則，可以將這項非正式的過程轉變成操作語意。這些規則本身需要以某些語言（**元語言** [*metalanguage*]）撰寫，通常是數學的表示方式。

我們將會在本章探討一種玩具程式語言的語意，它的名字是 SIMPLE [3]。

SIMPLE 小步語意的數學描述看起來像是這樣：

$$\frac{\langle e_1, \sigma\rangle \rightsquigarrow_e e_1'}{\langle e_1 \texttt{ + } e_2, \sigma\rangle \rightsquigarrow_e e_1' \texttt{ + } e_2} \qquad \frac{\langle e_2, \sigma\rangle \rightsquigarrow_e e_2'}{\langle v_1 \texttt{ + } e_2, \sigma\rangle \rightsquigarrow_e v_1 \texttt{ + } e_2'}$$

$$\frac{}{\langle n_1 \texttt{ + } n_2, \sigma\rangle \rightsquigarrow_e n} \text{ if } n = n_1 + n_2$$

$$\frac{\langle e_1, \sigma\rangle \rightsquigarrow_e e_1'}{\langle e_1 \texttt{ * } e_2, \sigma\rangle \rightsquigarrow_e e_1' \texttt{ * } e_2} \qquad \frac{\langle e_2, \sigma\rangle \rightsquigarrow_e e_2'}{\langle v_1 \texttt{ * } e_2, \sigma\rangle \rightsquigarrow_e v_1 \texttt{ * } e_2'}$$

$$\frac{}{\langle n_1 \texttt{ * } n_2, \sigma\rangle \rightsquigarrow_e n} \text{ if } n = n_1 \times n_2$$

$$\frac{\langle e_1, \sigma\rangle \rightsquigarrow_e e_1'}{\langle e_1 \texttt{ < } e_2, \sigma\rangle \rightsquigarrow_e e_1' \texttt{ < } e_2} \qquad \frac{\langle e_2, \sigma\rangle \rightsquigarrow_e e_2'}{\langle v_1 \texttt{ < } e_2, \sigma\rangle \rightsquigarrow_e v_1 \texttt{ < } e_2'}$$

$$\frac{}{\langle n_1 \texttt{ < } n_2, \sigma\rangle \rightsquigarrow_e \texttt{true}} \text{ if } n_1 < n_2 \qquad \frac{}{\langle n_1 \texttt{ < } n_2, \sigma\rangle \rightsquigarrow_e \texttt{false}} \text{ if } n_1 \geq n_2$$

$$\frac{}{\langle x, \sigma\rangle \rightsquigarrow_e \sigma(x)} \text{ if } x \in \mathrm{dom}(\sigma)$$

$$\frac{\langle e, \sigma\rangle \rightsquigarrow_e e'}{\langle x \texttt{ = } e, \sigma\rangle \rightsquigarrow_s \langle x \texttt{ = } e', \sigma\rangle} \qquad \frac{}{\langle x \texttt{ = } v, \sigma\rangle \rightsquigarrow_s \langle \texttt{do-nothing}, \sigma[x \mapsto v]\rangle}$$

$$\frac{\langle e, \sigma\rangle \rightsquigarrow_e e'}{\langle \texttt{if (}e\texttt{) \{ }s_1\texttt{ \} else \{ }s_2\texttt{ \}}, \sigma\rangle \rightsquigarrow_s \langle \texttt{if (}e'\texttt{) \{ }s_1\texttt{ \} else \{ }s_2\texttt{ \}}, \sigma\rangle}$$

$$\frac{}{\langle \texttt{if (true) \{ }s_1\texttt{ \} else \{ }s_2\texttt{ \}}, \sigma\rangle \rightsquigarrow_s \langle s_1, \sigma\rangle} \qquad \frac{}{\langle \texttt{if (false) \{ }s_1\texttt{ \} else \{ }s_2\texttt{ \}}, \sigma\rangle \rightsquigarrow_s \langle s_2, \sigma\rangle}$$

$$\frac{\langle s_1, \sigma\rangle \rightsquigarrow_s \langle s_1', \sigma'\rangle}{\langle s_1\texttt{; }s_2, \sigma\rangle \rightsquigarrow_s \langle s_1'\texttt{; }s_2, \sigma'\rangle} \qquad \frac{}{\langle \texttt{do-nothing; }s_2, \sigma\rangle \rightsquigarrow_s \langle s_2, \sigma\rangle}$$

$$\frac{}{\langle \texttt{while (}e\texttt{) \{ }s\texttt{ \}}, \sigma\rangle \rightsquigarrow_s \langle \texttt{if (}e\texttt{) \{ }s\texttt{; while (}e\texttt{) \{ }s\texttt{ \} \} else \{ do-nothing \}}, \sigma\rangle}$$

3 如果你想知道的話，它可以是 simple imperative language（簡單命令式語言）的縮寫。

就數學的說法，這是一組在 SIMPLE 的抽象語法樹定義了**化簡關係**（*reduction relation*）的**推理規則**（*inference rule*）。而實際上那些是一群怪異的符號，無法表示任何可以理解的電腦程式的意義。

我們將研究如何以 Ruby 編寫相同的推理規則，而不是試著直接瞭解這種形式的表示法。利用 Ruby 作為元語言，對程式設計者來說，會更容易瞭解，而且它還提供了額外的優點，能夠執行規則來觀察它們是如何運作。

我們不會試著以『源自實作物的規格』來描述 SIMPLE 的語意。以 Ruby 取代數學表示法來描述小步語意的主要理由，是為了讓讀者更容易閱讀相關描述並融會貫通。這個語言最終的可執行實作物只是額外的獎品。

使用 Ruby 最大的缺點是它以更為複雜的語言來解釋簡單的語言，這讓如此的哲學目的因而受挫。我們應該記住，數學規則是語意的權威描述，而我們只是利用 Ruby 來瞭解那些規則的意義。

運算式

我們將以檢視 SIMPLE 運算式的語意作為起點。這裡的規則將可在這些運算式的抽象語法運作，所以我們需要能將 SIMPLE 運算式表示成 Ruby 物件。方式之一是根據 SIMPLE 的語法替每種不同類型的元素定義 Ruby 類別（Number 數值、Add 加法、Multiply 乘法等），然後將每個運算式表示成這些類別的實體樹狀結構。

舉例來說，這裡是 Number、Add、Multiply 等類別的定義：

```
class Number < Struct.new(:value)
end

class Add < Struct.new(:left, :right)
end

class Multiply < Struct.new(:left, :right)
end
```

我們可以自行產生這些類別的實體來建立抽象語法樹：

```
>> Add.new(
     Multiply.new(Number.new(1), Number.new(2)),
     Multiply.new(Number.new(3), Number.new(4))
)
=> #<struct Add
     left=#<struct Multiply
       left=#<struct Number value=1>,
       right=#<struct Number value=2>
```

```
    >,
    right=#<struct Multiply
      left=#<struct Number value=3>,
      right=#<struct Number value=4>
    >
  >
```

最後我們當然希望有個解析器能自動建立這些樹，第 61 頁的『實作解析器』會討論這部分的細節。

Number、Add、Multiply 等類別繼承了 Struct 的 #inspect 泛型定義，因此它們的實體呈現在 IRB 主控台裡的字串包含了許多不重要的細節。為了能更容易的在 IRB 觀察抽象語法樹的內容，我們將覆寫每個類別的 #inspect [4]，以便它傳回呈現用的自訂字串：

```
class Number
  def to_s
    value.to_s
  end

  def inspect
    "«#{self}»"
  end
end

class Add
  def to_s
    "#{left} + #{right}"
  end

  def inspect
    "«#{self}»"
  end
end

class Multiply
  def to_s
    "#{left} * #{right}"
    end

  def inspect
    "«#{self}»"
  end
end
```

4 為了簡單起見，我們不會將通用程式碼擷取到超類別或模組。

第 3 章

最簡單的電腦

我們在短短的幾年之內就被電腦包圍了。它們曾經安全的隱身在軍事研究中心和大學實驗室，但現在它們無所不在：在我們的桌上、口袋、汽車引擎蓋底下，甚至植入我們的身體。身為程式設計者，雖然每天都在使用複雜的計算裝置，但我們瞭解它們的運作方式有多深呢？

伴隨現代電腦的威力而來的是諸多的複雜性，使得電腦眾多子系統的每個細節變得難以瞭解，而這些子系統彼此如何相互建立整個系統的細節，更是難以瞭解。這種複雜性使得直接推論現實電腦的能力和行為變得不切實際，因此，擁有簡化過並能完全瞭解細節且與現實機器共享有用功能的電腦模型，將會相當有用。

我們會在本章剝去電腦的概念，讓電腦回到最原始的本質，看看它可以用來做什麼，並且探索這類簡單電腦的極限。

決定論有限自動機

現實生活裡的電腦通常擁有大量的揮發性記憶體（RAM）和非揮發性儲存空間（硬碟或 SSD）、許多 I/O 裝置，以及能夠同時執行多個指令的數個處理器核心。**有限狀態機**（*finite state machine*），也稱為**有限自動機**（*finite automaton*），是經過相當程度簡化的電腦模型，它拋出了這所有的功能，以換取易於理解、易於推論、易於以硬體或軟體實作。

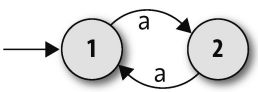
狀態、規則、輸入

有限自動機沒有永久固定的儲存空間，也沒有實質的 RAM。它是具有少數可能**狀態**（state）的小機器，並且能夠追蹤目前的狀態（可將它想作擁有足夠 RAM 的電腦，且能儲存單一的值）。

同樣的，有限自動機沒有接收輸入的鍵盤、滑鼠、網路介面，只有每次能讀取一個輸入字元的單一外部串流。

不同於可執行任意程式的通用型 CPU，每個有限自動機有 1 組已經寫死的**規則**（rule），這些規則決定了在回應輸入時，如何從某種狀態移到另 1 種狀態。此自動機在某種特定狀態下啟動，並從它的輸入串流讀取個別的字元，遵循的規則是每次讀取 1 個字元。

以下是某種特定有限自動機架構的視覺化方式：



上圖兩個圓表示自動機的兩種狀態，也就是狀態 1 和狀態 2，而不知從何而來的箭頭表示自動機始終從狀態 1 開始，這是它的**起始狀態**（*start state*）。 狀態之間的箭頭表示機器的規則，它們是：

- 如果是在狀態 1，且讀取了字元 a，就轉成狀態 2。
- 如果是在狀態 2，且讀取了字元 a，就轉成狀態 1。

這些資訊已經足以讓我們研究此機器如何處理輸入串流：

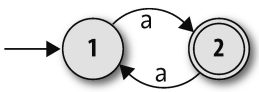
- 此機器起始於狀態 1。
- 此機器僅有從它的輸入串流讀取字元 a 的規則，因此那就是它唯一會發生的事情。當它讀取 a 時，會從狀態 1 轉成狀態 2。
- 當此機器讀取另一個 a，會轉回狀態 1。

一旦轉回狀態 1，它就會開始重複自己，這也正是這個特定機器行為的範圍。關於目前狀態的資訊，則假設是自動機的內部細節（它是當作『黑盒子』運作，不會揭露其內部作業），所以它的徒勞無用加重了這種行為的無聊程度，不會造成任何類型的顯著輸出。機器在狀態 1、2 之間反覆時，外部的世界沒有人能看到有任何事情發生，所以我們在這種情況可能也有單一狀態，並且完全不會影響任何內部結構。

輸出

要解決這個問題，有限自動機還有一種尚未完成的輸出產生方式，這種方式不像真的電腦的輸出能力那麼複雜，我們只需將一些狀態標示成特殊狀態，並且宣稱機器的單一位元輸出就是它目前是否處於特殊狀態的資訊。對此機器來說，讓我們使狀態 2 成為特殊

狀態，並在下圖裡以雙圓顯示：

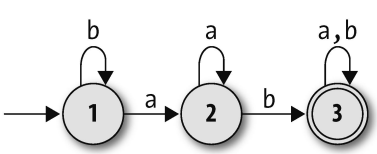


這些特殊狀態通常稱為**接受狀態**（*accept state*），這表示機器**接受**（*accepting*）或**拒絕**（*rejecting*）某些輸入序列的構想。如果這個自動機在狀態 1 起始並且讀取單一個 `a`，它將停留在狀態 2，那是接受狀態，所以我們可以說機器接受了字串 `'a'`。另一方面，如果它先讀 `a` 然後再讀取另一個 `a`，最後會回到狀態 1，但這不是接受狀態，因此機器便拒絕了字串 `'aa'`。事實上，很容易看出這個機器接受任何長度為奇數的字串：`'a'`、`'aaa'`、`'aaaaa'` 都會接受，而 `'aa'`、`'aaaa'` 和 `''`（空字串）則會遭到拒絕。

現在我們有一些更有用的東西：可以讀取一串字元的機器，並且提供 yes/no 輸出來指示是否接受該串字元。合理的說法是，這個 DFA 正在執行計算，因為我們向它提出問題（『這個字串的長度是奇數嗎？』），並得到有意義的回覆。這應該已經可以稱它為簡單的電腦，而且我們可以進一步比較它與真正電腦的功能：

	真正的電腦	有限自動機
永久儲存空間	硬碟或 SSD	無
暫時儲存空間	RAM	目前的狀態
輸入	鍵盤、滑鼠、網路等	內含字元的串流
輸出	顯示裝置、喇叭、網路等	目前的狀態是否為接受狀態（yes/no）
處理器	能執行任何程式的 CPU 核心	為回應輸入而改變狀態的寫死的規則

雖然這個特殊的自動機的確不做任何複雜或有用的事情，但我們可以建置擁有更多狀態並且能讀取數個字元的複雜自動機。以下這個自動機擁有 3 個狀態，並且能讀取輸入的 `a` 和 `b`：



這部機器可接受像是 `'ab'`、`'baba'`、`'aaaab'` 之類的字串，並且拒絕像是 `'a'`、`'baa'`、`'bbbba'` 之類的字串。若干實驗顯示，它只接受包含序列 `'ab'` 的字串，雖然這依然不是非常有用，但至少證明了一定程度的細微區別。我們將在本章的後續討論更多的實際應用。

決定論

重要的是，這種自動機屬於**決定論**（*deterministic*）：無論它目前的狀態為何，也不論它讀取的字元為何，一定會確定它將在哪個狀態結束。只要我們遵守兩項限制，就能保證這種確定：

- 無矛盾：因為規則衝突，因此無狀態機器的下一步並不明確（這意謂著沒有狀態可能對同一個輸入字元有多個規則）。
- 不遺漏：因為缺少規則，因此無狀態機器的下一步是未知的（這意謂著每個可能輸入字元的每個狀態必須擁有至少一條規則）。

總而言之，這些限制意謂著該機器的每一項狀態和輸入的組合都必須要有一條規則，而遵從決定論限制的機器，它的技術名稱就是**決定論有限自動機**（*deterministic finite automaton*，DFA）。

模擬物

決定論有限自動機的目的是作為抽象的計算模型。我們已經繪製了一些機器範例示意圖，並且思索它們的行為，但這些機器實際上不可能存在，因此我們無法實際將資料輸入給它們，並且觀察它們的行為。所幸 DFA 非常簡單，我們很容易以 Ruby 建置**模擬物**（*simulation*），並直接與它互動。

讓我們藉由實作我們稱為**規則手冊**（*rulebook*）的一組規則，來開始建置模擬：

```
class FARule < Struct.new(:state, :character, :next_state)
  def applies_to?(state, character)
    self.state == state && self.character == character
  end

  def follow
    next_state
  end

  def inspect
    "#<FARule #{state.inspect} --#{character}--> #{next_state.inspect}>"
  end
end

class DFARulebook < Struct.new(:rules)
  def next_state(state, character)
    rule_for(state, character).follow
  end
```

```
    def rule_for(state, character)
      rules.detect { |rule| rule.applies_to?(state, character) }
    end
  end
```

這段程式碼替規則建立了一個簡單的 API：每個規則都有一個 #applies_to? 方法和 #follow 方法，前者會傳回 true 或 false，來指示該規則是否適用特定情況；而 #follow 方法會傳回遵循規則時，機器應如何改變的相關資訊[1]。DFARulebook#next_state 使用這些方法來找出正確的規則和 DFA 的下一個狀態。

藉由使用 Enumerable#detect，DFARulebook#next_state 的實作物會假設永遠有一條規則一定適用於給定的狀態和字元。如果適用的規則超過一條，將只有第一條規則具有效果，而且會忽略其他的規則；如果沒有的話，#detect 呼叫將會傳回 nil，而且如果它嘗試呼叫 nil.follow，將會導致模擬物當機。

這就是為什麼此類別稱為 DFARulebook，而不只是稱為 FARulebook：因為它只有在遵守決定論的限制時，才能正常運作。

規則手冊能讓我們將諸多規則包進單一物件，並且能詢問它接下來的狀態：

```
>> rulebook = DFARulebook.new([
     FARule.new(1, 'a', 2), FARule.new(1, 'b', 1),
     FARule.new(2, 'a', 2), FARule.new(2, 'b', 3),
     FARule.new(3, 'a', 3), FARule.new(3, 'b', 3)
   ])
=> #<struct DFARulebook …>
>> rulebook.next_state(1, 'a')
=> 2
>> rulebook.next_state(1, 'b')
=> 1
>> rulebook.next_state(2, 'b')
=> 3
```

1 這種設計通常足以容納不同類型的機器和規則，因此當事情變得更複雜的時候，我們將能在本書後續重複它。

這裡我們有個選擇，事關如何以 Ruby 的值表示我們自動機的狀態。最重要的是分別告知狀態的能力：我們的 DFARulebook#next_state 實作物需要能夠比較兩種狀態來決定它們是否相同，不然的話，它不會在意那些物件是數值、符號、字串、雜湊、或 Object 類別的匿名實體。

在這種情況下，使用簡易、舊式的 Ruby 數值就最為清楚（它們與先前示意圖裡的數值狀態非常搭配），所以現在我們就將這麼做。

只要我們擁有規則手冊，就能以它來建置 DFA 物件，追蹤它目前的狀態，並能回報它目前是不是接受狀態：

```
class DFA < Struct.new(:current_state, :accept_states, :rulebook)
  def accepting?
    accept_states.include?(current_state)
  end
end

>> DFA.new(1, [1, 3], rulebook).accepting?
=> true
>> DFA.new(1, [3], rulebook).accepting?
=> false
```

現在我們可以編寫方法來讀取輸入字元、查詢規則手冊，並且適當的更改狀態：

```
class DFA
  def read_character(character)
    self.current_state = rulebook.next_state(current_state, character)
  end
end
```

這讓我們將字元輸入 DFA，並觀察它的輸出變化：

```
>> dfa = DFA.new(1, [3], rulebook); dfa.accepting?
=> false
>> dfa.read_character('b'); dfa.accepting?
=> false
>> 3.times do dfa.read_character('a') end; dfa.accepting?
=> false
>> dfa.read_character('b'); dfa.accepting?
=> true
```

一次輸入 1 個字元給 DFA 是有點笨拙，所以讓我們加入方便的方法來讀取整個輸入字串：

```
class DFA
  def read_string(string)
```

```
    string.chars.each do |character|
      read_character(character)
    end
  end
end
```

現在我們可以提供整個輸入字串給 DFA，而不需將個別的字元傳給它：

```
>> dfa = DFA.new(1, [3], rulebook); dfa.accepting?
=> false
>> dfa.read_string('baaab'); dfa.accepting?
=> true
```

一旦將資料提供給 DFA 物件，它可能就再也不是它的起始狀態，因此重複使用它來檢查全新的一組輸入就變得不可靠。這意謂著每次我們想要觀察它是否接受新字串，就必須以同前的起始狀態、接受狀態、規則手冊，全部重新建立它。如果我們想要檢查字串的接受情況，而且想要免去手動完成這些，只要依照以下作法即可：將其建構式的參數包進代表特定 DFA 的設計物件裡，並且根據該物件自動建置該 DFA 的一次性實體：

```
class DFADesign < Struct.new(:start_state, :accept_states, :rulebook)
  def to_dfa
    DFA.new(start_state, accept_states, rulebook)
  end

  def accepts?(string)
    to_dfa.tap { |dfa| dfa.read_string(string) }.accepting?
  end
end
```

#tap 方法會估算整個區塊，然後傳回它所呼叫的物件。

DFADesign#accepts? 會使用 DFADesign#to_dfa 方法來建立新的 DFA 實體，然後呼叫 #read_string?，即可將它放進接受狀態或拒絕狀態：

```
>> dfa_design = DFADesign.new(1, [3], rulebook)
=> #<struct DFADesign …>
>> dfa_design.accepts?('a')
=> false
>> dfa_design.accepts?('baa')
=> false
>> dfa_design.accepts?('baba')
=> true
```

非決定論有限自動機

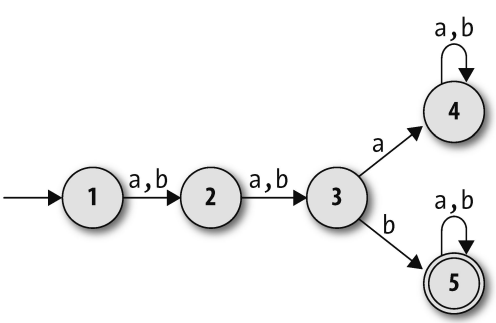
要瞭解 DFA 並加以實作是一件簡單的事情，但那是因為它們非常類似我們已經熟悉的機器。在剝離了真實電腦的所有複雜性之後，我們現在有機會嘗試較不常規的想法，讓我們遠離過去使用的機器，而不必處理隨著這些想法在真實系統運作而來的困難。

其中一種探索的方法是消除我們現有的假設和限制。首先，決定論似乎有其限制：也許我們不在意每種狀態的每個可能的輸入字元，因此在某些意外發生時，我們為何就不能忽略我們不在意的字元規則，並且假設該機器可以進入一般的故障狀態？更奇怪的是，允許機器擁有相互矛盾的規則以便可以擁有一條以上的執行路徑到底意謂著什麼？我們的方案也假設每個狀態更改都必須發生在從輸入串流讀取字元的回應，但如果該機器勿需讀取任何字元就可以改變狀態，這會發生什麼事？

我們將在本節研究這些想法，並觀察調整有限自動機的能力會開啟什麼新的可能性。

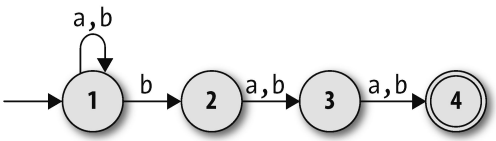
非決定論

假設我們想要一部只要第 3 個字元是 b 就會接受任何字串 a 和 b 的有限自動機（這裡的字串 a、b 裡面可以有數個 a 或 b 字元）。這很容易就能提出適合的 DFA 設計：



如果我們想要的機器所接受字串的**倒數第 3 個**字元是 b，這該怎麼辦？它會如何運作？似乎更為困難：上述會保證 DFA 在讀取第 3 個字元時會位於狀態 3，但是機器在讀取倒數第 3 個字元時並無法提前得知，因為它直到讀完字串才知道字串有多長。它可能無法馬上清楚這樣的 DFA 是否更為可行。

然而，如果我們放寬決定論限制並允許規則手冊針對給定的狀態和輸入包含多條規則（或完全沒有規則），那麼我們就可以設計一部執行該項任務的機器：



這種狀態機器，也就是**非決定論有限自動機**（*nondeterministic finite automaton*，NFA），它的每組輸入都不再有正確的執行路徑。當它處於狀態 1 並且讀取 b 作為輸入時，可能為了它而遵循著將其保持在狀態 1 的規則，但也可能為了它改而遵循使其進入狀態 2 的不同規則。反之，一旦進入狀態 4，就沒有可以遵循的規則，也因此無法讀取任何更多的輸入。DFA 的下一個狀態一定完全由它目前的狀態和輸入所決定，但是 NFA 的下一個狀態有時會有一種以上的可能，有時卻完全沒有。

如果 DFA 讀取字元並盲目遵循規則而造成機器最後處於接受狀態，那麼 DFA 就會接受字串，因此 NFA 接受或拒絕字串的意義為何？直覺的答案是，如果有一些方式可以讓 NFA 遵循它的某些規則而在最後處於接受狀態，就會接受字串；也就是說，即使並非必然，但可能達成接受狀態。

舉例來說，因為是從狀態 1 開始，因此這個 NFA 接受字串 `'baa'`，而規則裡提及了一種方式可以讓機器讀取 b 並進入狀態 2，然後讀取 a 再進入狀態 3，最後讀取另一個 a 並在接受狀態的狀態 4 結束。它也接受字串 `'bbbbb'`，因為 NFA 最初可以遵循不同的規則，並且在讀取前兩個 b 的時候停留在狀態 1，然後在讀取第 3 個 b 的時候僅使用進入狀態 2 的規則，接著讓它讀取字串剩餘的部分，然後一如先前在狀態 4 結束。

另一方面，它卻無法讀取 `'abb'`，並在最後處於狀態 4（根據它所遵循的規則，最後只能在狀態 1、2 或 3），所以這個 NFA 並不接受 `'abb'`。而且也不接受 `'bbabb'`，這只能到達狀態 3；如果它在讀取第 1 個 b 的時候直接進入狀態 2，最後會太早進入狀態 4，而它有兩個字元依然還得讀取，但沒有更多的規則要遵循。

特定機器接受的字串集合稱為**語言**（language）：也就是說該機器能夠**識別**（*recognize*）那個語言。並非所有可能的語言都擁有可以識別它們的 DFA 或 NFA（細節請見第 4 章），但有限自動機能夠識別的語言則稱為**正規語言**（*regular language*）。

放寬決定論限制所產生的虛構機器，和我們熟悉的真實、決定論的電腦非常不同。NFA 處理可能的事物而非確定的事物；我們是以**可能**發生什麼而非**將會**發生什麼作為談論其行為的用語。這似乎威力強大，但這樣的機器會如何在真實世界運作？它乍看之下就像 NFA 的真正實作物，為了在它讀取輸入時能得知數種可能性而加以選擇，因此需要某種遠見：為了有機會接受字串，我們的範例 NFA 必須停留在狀態 1，一直到它讀取倒數第 3 個字元，但它無法得知會接收到多少字元。我們要怎麼以無趣且決定論的 Ruby 模擬像這樣有趣好玩的機器呢？

在決定論電腦模擬 NFA 的關鍵，是找到探索機器**所有可能執行**的方式。這種暴力破解的作法會沿途以某種方式做出所有正確的決定，以剔除要求模擬僅一種可能執行方式的怪異遠見。當 NFA 讀取字元時，接下來能做的始終只有有限數量的可能性，所以我們可以模擬非決定論，作法是以某種方式試過它們每一個，並觀察它們是否有任何一個最終可允許它達到接受狀態。

我們可以遞迴的方式試過所有的可能性來完成它：每次模擬的 NFA 讀取 1 個字元，並且有 1 條以上的適用規則，遵循其中 1 條規則，然後嘗試讀取剩下的輸入；如果這不會讓機器處於接受狀態，就回到先前的狀態，將輸入轉回它先前的位置，再重新嘗試遵循不同的規則；不斷重複，一直到某些規則的選擇造成接受狀態，或者沒有成功而一直試過所有可能的選擇。

另一種策略是在機器每次擁有多個下次可以遵循的規則時，建立新的緒程來平行模擬所有可能性，進而以更有效率的方式複製模擬的 NFA，以便每個 NFA 的複製品可以嘗試不同的規則，來觀察它是如何成功。這些所有的緒程可以馬上執行，每次讀取它自己輸入字串的複製品，且若任何緒程以讀取每個字元並停在接受狀態的機器結束了，那麼我們就能說字串已經獲得接受。

這兩種實作物都可行，但是它們有點複雜且效率不彰。我們的 DFA 模擬物很簡單，而且在讀取個別字元的同時，還可以持續回報機器是否處於接受狀態，因此這種很棒的 NFA 模擬方式會帶給我們同樣的簡易和透明。

所幸有個簡單的 NFA 模擬方式，這種方式不需要反轉我們進度、不需要建立緒程，也不需要事先知道所有輸入的字元。事實上，就如同之前記錄 DFA 的目前狀態來模擬單一個 DFA，現在也可以記錄 NFA 所有**可能**的目前狀態來模擬單一個 NFA。相較在不同方向模擬多個 NFA，這種作法更簡單也更有效率，並且最後會完成相同的結果。如果我們之前真的模擬過許多個別的機器，那麼我們在意的是它們所處的狀態，但是相同狀態完全無法分辨出任何機器[2]，所以我們不會有所損失。