

第一部分

程序和机器

什么是计算？这个词对于不同人来说意思不同，但是每个人都会赞同这样一种理解：在一台计算机读取程序、运行程序、读入一些输入，并且最后产生一些输出的时候，肯定发生了某种计算。因此我们可以这样认为：计算就是指计算机所做的事情。

为了创造一个环境让这种熟悉的计算发生，需要三个基本要素：

- 一台机器，能够执行计算；
- 一种语言，用来编写这台机器能够理解的指令；
- 一个程序，用这种语言编写，描述机器应该具体执行哪些计算。

这部分内容就是关于机器、语言和程序的：它们是什么，行为如何，我们如何对其建模并展开研究，以及如何利用它们完成实际工作。通过研究这三要素，我们将对计算的含义以及它是如何发生的有更好的理解。

在第 2 章，我们将设计和实现一种简单的编程语言，并用几种不同的方法来研究这种语言的含义。理解了一种语言的含义，就可以把一段没有生命的源代码和一个动态的、正在执行的进程联系起来。每一种方法都能带给我们一个把程序运行起来的特定策略，而我们最终将用几种不同的方式来实现同一语言。

我们会发现编程是一门把一个准确定义的结构组装起来的艺术，这个结构能拆卸、分析，并最终被一台机器解释执行从而完成一次计算。更重要的是，我们还会发现实现编程语言既简单又有趣：尽管语法分析、解释和编译看起来很吓人，但实际摆弄起来其实会感觉简单又愉快。

如果没有机器来运行，程序本身没有多大用处。所以在第 3 章里，我们会设计非常简单的机器，以便执行基本的、硬编码的任务。有了这个简单的基础，我们在第 4 章会向更复杂的机器努力前进，并在第 5 章介绍如何设计能被软件控制的通用计算装置。

到第二部分的时候，我们将了解拥有计算能力的机器的全景：一些机器拥有非常有限的能力，一些机器用处更大但仍然令人沮丧地有一些限制，最后还有一些机器是我们知道如何构建的最强大的机器。

第2章

程序的含义



不准想，快点！就像直觉地把手指向月亮。记住，反应慢了就只能看到手指，而绝不能看到月亮的光华了。

——电影《龙争虎斗》，李小龙

编程语言，以及我们用编程语言所写的程序，这些都是软件工程师工作的基础。我们用编程语言和程序阐明复杂的想法，并在彼此之间交流这些想法，当然最重要的是在计算机中实现这些想法。就像人类社会没有自然语言就难以运转一样，全球的程序员都依赖编程语言传递和实现自己的想法，每一个有成效的程序都是实现更高层思想的基础。

程序员是注重实际的生物。程序员经常通过阅读文档、学习教程、研究现有的程序以及修改自己的简单程序来学习新的编程语言，而不会过多地思考那些程序有什么含义。有时候，学习的过程就像试错：我们试图通过看例子和文档来理解一个语言片段，然后会努力用这种语言写点什么，之后所有问题就都爆发了，而我们只得回头重试，直到成功组装了一个大部分情况下都能工作的东西。随着程序支持的计算机和系统越来越复杂，它们很容易被看成是一些难懂的符咒，这些符咒只代表它们自己而看不出有什么含义，并且它们只是偶尔才能正常工作。

但是计算机编程不单是与程序相关，重要的是程序员要表达的思想。程序只是思想的静态表示，是曾经存在于程序员脑海中的某个结构的快照。程序是因为有了含义才值得写下来。那么是什么把代码和它的含义连接在一起呢？除了说“它做了该做的事”，怎样才能将一个程序的含义说得更具体一点呢？本章，你将会看到一些确定计算机程序含义的方法，了解如何给那些死板的“静态快照”注入生命气息。

2.1 “含义”的含义

在语言学中，语义学（semantics）研究的是单词和它们含义之间的关系：单词“dog”是纸上一些符号的组合，或是由某个人声带引起的一系列空气振动，这与真正的狗或者通常意义上狗的概念极为不同。语义不止关注抽象含义本身的基本性质，还关注具体的记号如何与它们的抽象含义关联起来。

计算机科学里，形式语义学注重找到确定程序难以捉摸的含义的方法，并利用这些方法发现或者证明编程语言中有趣的东西。形式语义学得到了广泛应用，从定义新的语言和进行编译优化这种具体的应用，到构造程序正确性的数学证明这样更抽象的领域不一而足。

为了完整地定义编程语言，我们需要：语法，描述程序看起来是什么样的；语义（semantics）¹，描述程序的含义。

许多语言都没有官方的书面规范，而只有一个可用的解释器或者编译器。Ruby 本身算是“靠实现规范”这一类：尽管有很多关于 Ruby 应该如何工作的书和教程，但这些资料的最终源头都是松本行弘先生（Matz）的 Ruby 解释器（MRI，Matz's Ruby Interpreter），这是 Ruby 的参考实现。如果任何一份 Ruby 文档与 MRI 的实际行为不一致，那必然是文档错了；JRuby、Rubinius 以及 MacRuby 这些第三方实现都只能努力地精准模拟 MRI 的行为，只有如此，它们才可以声称自己与 Ruby 语言有效地兼容。其他像 PHP 和 Perl 5 这样的语言，也使用了这种以实现为主导的语言定义方法。

另一种描述编程语言的方法，就是写一份平实的官方规范（一般是英语的）。C++、Java 以及 ECMAScript（JavaScript 的标准版本）都使用了这种方法：这些语言的标准化通过由专家委员会写成的、与实现无关的文档来完成，而且会存在很多与这些标准兼容的实现。比起只是依赖于一个参考实现，用官方文档规范定义一种语言更为严谨：这样所做的设计决策更有可能是经过深思熟虑、进行理性选择之后的，而不是某一个特定实现的意外结果。但是，规范通常非常难懂，而且很难讲规范中是不是含有矛盾、疏漏和有歧义的地方。特别是一份英语规范没有形式化的方法可以进行推导，我们只能完整彻底地阅读规范，大量地思考，然后寄希望于这样就可以掌握所有的前因后果。



Ruby 1.8.7 的规范确实存在，甚至已经被接受为 ISO 标准了（ISO/IEC 30170）²。尽管 mruby 工程（<https://github.com/mruby/mruby>）尝试构建一份轻量级、嵌入式的 Ruby 实现，并且明确声明将与 ISO 标准而不是 MRI 兼容，但 MRI 仍然被认为是 Ruby 语言由实现定义的权威规范。

注 1：在讨论编程语言理论的环境下，单词 semantics 通常被当作单数对待：我们通过为语言赋予语义来描述这种语言的含义。

注 2：尽管访问 ISO/IEC 30170 需要支付费用，但这一规范的一份早期草案可以免费下载：<http://ipa.go.jp/osc/english/ruby/>。

第三种方法是使用形式语义学中的数学方法准确描述编程语言的含义。它的目标是不仅能够用适合系统分析甚至自动化分析的格式写出规范，还能保证其完全没有歧义，这样就可以对规范是否一致、是否含有冲突，以及是否有疏漏进行全面检查。在介绍如何处理语法之后，我们将会看到语义规范的这些形式化方法。

2.2 语法

传统的计算机程序是长长的字符串。每一种编程语言都有一系列规则，描述在那种语言中什么样的字符串被认为是有效程序。这些规则定义了这种语言的语法。

通过语言的语法规则，我们能把像 $y = x + 1$ 这样可能有效的程序与像 $>/;x:1@4$ 这样毫无意义的字符串区分开。语法规则还为如何阅读一些具有二义性的程序提供了有用信息，例如运算符优先级的规则能够自动判定 $1 + 2 * 3$ 按其本意 $1 + (2 * 3)$ 处理，而不是按 $(1 + 2) * 3$ 处理。

当然，计算机程序的预期用途是被计算机读取，而要读程序就需要语法解析器：这个分析器程序能够读取代表程序的字符串，根据语法规则检查它是否有效，然后把它转换成一个适合被进一步处理的结构化表示。

有各种各样的工具能把一种语言的语法规则自动转换成一个语法解析器。具体如何对这些规则进行定义，以及把它们转成可用语法解析器的技术，并不是本章的讲解重点（2.6 节进行了简单介绍），但总体来讲一个语法解析器应该读入像 $y = x + 1$ 这样的字符串，然后把它转换成抽象语法树（AST）。抽象语法树是源代码的一种表示，去掉了空格之类的无关细节，而只关注程序的分层结构。

语法关心的只是程序的表面是什么样的，而不是它的含义。程序有可能语法正确但没有任何实际意义。例如，程序 $y = x + 1$ 本身可能没有任何意义，因为并没有事先说明 x 是什么，而程序 $z = \text{true} + 1$ 可能在运行时候报错，因为它试图在一个布尔型值上加数字。（当然，这依赖于具体编程语言的其他属性。）

正如我们所料，能说明如何把一种编程语言的语法与这个语法暗含的语义对应起来的“唯一正途”并不存在。实际上，关于程序的含义有几种不同的研究方法，它们都在形式化（formality）、抽象度（abstraction）、可表达性（expressiveness）和实际效率（efficiency）之间做了权衡。在接下来的几节里，我们将看到这些主要的形式化方法，并了解它们之间的联系。

2.3 操作语义

考虑程序含义的最实际方法是思考它做了些什么：在运行程序的时候，我们期望发生什么

呢？在运行时编程语言中不同的结构都是如何表现的？把它们放到一起组成更大的程序时会是什么效果？

这是操作语义学（operational semantic）的基础，这种方法为程序在某种机器上的执行定义一些规则，以此来捕捉编程语言的含义。这个机器常常是一种抽象的机器：为了解释这种语言所写的程序如何执行而设计出来的一个想象的、理想化的计算机。为了更好地捕获编程语言的运行时行为，通常需要针对不同种类的编程语言设计不同的抽象机器。

有了操作语义，我们可以朝着严谨而准确地研究语言中特定结构的目标前进了。用英语写成的语言规范可能暗藏着二义性，并且可能遗漏边缘情况，但一个形式化的操作性规范不会如此，为了令人信服地传达语言的行为，它必须明确而且无二义性。

2.3.1 小步语义

那么，我们如何设计一台抽象机器，并使用它定义一种编程语言的操作语义呢？一种方法就是假想一台机器，用这台机器直接按照这种语言的语法进行操作一小步一小步地对其进行反复规约，从而对一个程序求值。不管最后得到的结果含义是什么，我们每一步都能让程序更接近最终结果。

这种小步规约类似于对代数式求值的方式。例如，为了对 $(1 \times 2) + (3 \times 4)$ 求值，我们知道应该：

- (1) 执行左侧的乘法 (1×2 变成了 2)，这样表达式就规约成了 $2 + (3 \times 4)$ ；
- (2) 执行右侧的乘法 (3×4 变成了 12)，这样表达式规约成了 $2 + 12$ ；
- (3) 执行加法 ($2 + 12$ 变成了 14)，最终得到 14 。

我们可以认为 14 就是结果，因为通过上面步骤已经不能再进一步规约了；我们认为 14 是一个特殊代数表达式，它是一个值，有自己的含义，不需要进一步的努力了。

把如何进行每一小步的规约写成形式化规则，这个非形式化的过程就可以转换成一个操作语义。这些规则本身需要用某种语言（元语言）写下来，而这种语言通常是数学符号。

本章，我们将探索一个玩具级编程语言的语义，姑且将这种语言叫作 Simple³。

Simple 的小步语义（small-step semantic）的数学化描述如下所示：

注 3：你可以把它看成简单命令式语言（simple imperative language）的缩写。

$$\begin{array}{c}
\frac{\langle e_1, \sigma \rangle \rightsquigarrow_e e'_1}{\langle e_1 + e_2, \sigma \rangle \rightsquigarrow_e e'_1 + e_2} \quad \frac{\langle e_2, \sigma \rangle \rightsquigarrow_e e'_2}{\langle v_1 + e_2, \sigma \rangle \rightsquigarrow_e v_1 + e'_2} \\
\frac{}{\langle n_1 + n_2, \sigma \rangle \rightsquigarrow_e n} \text{ if } n = n_1 + n_2 \\[10pt]
\frac{\langle e_1, \sigma \rangle \rightsquigarrow_e e'_1}{\langle e_1 * e_2, \sigma \rangle \rightsquigarrow_e e'_1 * e_2} \quad \frac{\langle e_2, \sigma \rangle \rightsquigarrow_e e'_2}{\langle v_1 * e_2, \sigma \rangle \rightsquigarrow_e v_1 * e'_2} \\
\frac{}{\langle n_1 * n_2, \sigma \rangle \rightsquigarrow_e n} \text{ if } n = n_1 * n_2 \\[10pt]
\frac{\langle e_1, \sigma \rangle \rightsquigarrow_e e'_1}{\langle e_1 < e_2, \sigma \rangle \rightsquigarrow_e e'_1 < e_2} \quad \frac{\langle e_2, \sigma \rangle \rightsquigarrow_e e'_2}{\langle v_1 < e_2, \sigma \rangle \rightsquigarrow_e v_1 < e'_2} \\
\frac{\langle n_1 < n_2, \sigma \rangle \rightsquigarrow_e \text{true}}{\langle n_1 < n_2, \sigma \rangle \rightsquigarrow_e \text{false}} \text{ if } n_1 < n_2 \quad \frac{}{\langle n_1 < n_2, \sigma \rangle \rightsquigarrow_e \text{false}} \text{ if } n_1 \geq n_2 \\[10pt]
\frac{}{\langle x, \sigma \rangle \rightsquigarrow_e \sigma(x)} \text{ if } x \in \text{dom}(\sigma) \\[10pt]
\frac{\langle e, \sigma \rangle \rightsquigarrow_e e'}{\langle x = e, \sigma \rangle \rightsquigarrow_s \langle x = e', \sigma \rangle} \quad \frac{}{\langle x = v, \sigma \rangle \rightsquigarrow_s \langle \text{do-nothing}, \sigma[x \mapsto v] \rangle} \\[10pt]
\frac{\langle e, \sigma \rangle \rightsquigarrow_e e'}{\langle \text{if } (e) \{ s_1 \} \text{ else } \{ s_2 \}, \sigma \rangle \rightsquigarrow_s \langle \text{if } (e') \{ s_1 \} \text{ else } \{ s_2 \}, \sigma \rangle} \\
\frac{\langle \text{if } (\text{true}) \{ s_1 \} \text{ else } \{ s_2 \}, \sigma \rangle \rightsquigarrow_s \langle s_1, \sigma \rangle}{\langle \text{if } (\text{false}) \{ s_1 \} \text{ else } \{ s_2 \}, \sigma \rangle \rightsquigarrow_s \langle s_2, \sigma \rangle} \\[10pt]
\frac{\langle s_1, \sigma \rangle \rightsquigarrow_s \langle s'_1, \sigma' \rangle}{\langle s_1; s_2, \sigma \rangle \rightsquigarrow_s \langle s'_1; s_2, \sigma' \rangle} \quad \frac{}{\langle \text{do-nothing}; s_2, \sigma \rangle \rightsquigarrow_s \langle s_2, \sigma \rangle} \\[10pt]
\frac{}{\langle \text{while } (e) \{ s \}, \sigma \rangle \rightsquigarrow_s \langle \text{if } (e) \{ s; \text{while } (e) \{ s \} \} \text{ else } \{ \text{do-nothing} \}, \sigma \rangle}
\end{array}$$

从数学上讲，这是一个推理规则的集合，它定义了基于 Simple 抽象语法树的一个规约关系。实际点儿讲，这是一堆怪异的符号，关于计算机程序的含义它没有讲任何能让人理解的东西。

我们不会试图直接理解这种形式化的符号，而是研究如何用 Ruby 编写同样的推导规则。对程序员来说使用 Ruby 做元语言更容易理解，而且这样还有一个优点，就是这些规则可以执行，我们能看到它们是如何工作的。



我们并不打算尝试用“靠实现来规范”的方式描述 Simple 的语义。使用 Ruby 而不是用数学符号来描述小步语义，主要是为了使描述更容易被人们所理解。最终得到一个这种语言的可执行实现，只是这么做的额外好处。

使用 Ruby 有一大缺点：这是在使用一种更复杂的语言解释一种简单的语言，从哲学上来说这可能很失败。我们应该记住，数学化的规则是语义的权威描述，而使用 Ruby 只是为了更容易地理解这些规则的含义。

1. 表达式

首先来研究一下 Simple 语言中表达式的语义。规则将作用于这些表达式的抽象语法树，所以我们必须把 Simple 表达式表示成 Ruby 对象。要做到这一点，一种方式就是为 Simple 语法中每一种不同的元素都定义一个 Ruby 类，包括数字 (number)、加法 (add)、乘法 (multiply) 等，然后把每一个表达式表示成由这些类的实例构成的一棵树。

例如，下面是 Number、Add 和 Multiply 三个类的定义：

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

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

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

实例化这些类来手工构造抽象语法树：

```
>> 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>
    >
    >
```



当然，最终我们想通过一个语法解析器自动构建这些树。2.6 节将介绍如何完成这件事情。

三个类 (Number、Add 和 Multiply) 都继承了 Struct 对 #inspect 的通用定义，所以在 IRB 中它们实例的字符串表示会含有大量不重要的细节。为了方便在 IRB 中查看抽象语法树的内容，我们将覆盖每个类的 #inspect 方法⁴，让它返回自定义的字符串表示：

```
class Number
  def to_s
    value.to_s
  end

  def inspect
    "#<#{self}>"
  end
```

注 4：为了让代码保持简单，我们将抑制住把公共代码提取到超类或者模块中的欲望。

```

    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

```

这样每个抽象语法树都将在 IRB 中以 Simple 源代码的形式呈现，外边会加上书名号（«）以便与正常的 Ruby 值区分。

```

>> Add.new(
      Multiply.new(Number.new(1), Number.new(2)),
      Multiply.new(Number.new(3), Number.new(4))
    )
=> «1 * 2 + 3 * 4»
>> Number.new(5)
=> «5»

```



我们对 `#to_s` 的基本实现并没有把运算优先级考虑进来，所以有时候如果按照传统的优先级规则（例如 * 通常比 + 优先级更高）它们的输出是不正确的。以下面的抽象语法树为例：

```

>> Multiply.new(
      Number.new(1),
      Multiply.new(
        Add.new(Number.new(2), Number.new(3)),
        Number.new(4)
      )
    )
=> «1 * 2 + 3 * 4»

```

这棵树表示 « $1 * (2 + 3) * 4$ » 与 « $1 * 2 + 3 * 4$ » 不是一个表达式（具有不同的含义），但字符串表示并没有反映出这一点。

这个问题很严重，但与我们关于语义的讨论完全无关。为简单起见，暂时先忽略此事，避开可能拥有不正确字符串描述的表达式。我们将在 3.3.1 节为另一种语言给出更合适的实现。

现在为抽象语法树定义规约方法，这将是我们实现一个小步操作语义的起点。也就是说，代码可以以一个抽象语法树作为输入，然后生成一个规约树作为输出。

在实现规约本身之前，我们先要区分什么样的表达式能规约，什么样的表达式不能规约。Add 和 Multiply 表达式总是能规约的（它们的每一个表达式都表示一个操作，并能够通过那种操作对应的计算变成一个结果），但是 Number 表达式总是代表一个值，它就不能规约成任何其他东西了。

原则上，我们可以使用简单的 `#reducible?` 判断把这两种表达式区分开，它能判断参数是否可规约，并返回 `true` 或者 `false`：

```
def reducible?(expression)
  case expression
    when Number
      false
    when Add, Multiply
      true
    end
  end
```



在 Ruby 的 `case` 语句里，控制表达式与 `case` 值是否匹配，是通过将控制表达式的值作为参数调用每个 `case` 值的 `#==` 方法来判断的。方法 `#==` 的实现会检查它的参数是否是那个类或者那个子类的实例，这样我们可以使用“`case 对象 when 类名`”这样的语法为一个类匹配一个对象。

但是，在一种面向对象语言里这么写代码通常被认为是不好的做法⁵；如果一些运算的行为依赖于它参数的类型，典型的做法是将这种每个类都有的行为实现为它们的实例方法，从而让语言隐式地决定调用哪个方法，而不是使用显式的 `case` 语句。

因此，我们将分别为 Number、Add 和 Multiply 实现 `#reducible?` 方法：

```
class Number
  def reducible?
    false
  end
end

class Add
  def reducible?
    true
  end
end

class Multiply
  def reducible?
```

注 5：尽管我们用 Haskell 或者 ML 这样的函数式语言写 `#reducible?` 时就是这么写的。

```
    true
  end
end
```

这回的表现正是我们想要的：

```
>> Number.new(1).reducible?
=> false
>> Add.new(Number.new(1), Number.new(2)).reducible?
=> true
```

现在可以为这些表达式实现规约了：像上面一样，我们为 Add 和 Multiply 定义一个 #reduce 方法。既然数字不能再规约，那就没有必要定义 Number#reduce 了，因此除非确切知道一个表达式能够规约，否则不要对其调用 #reduce 方法。

那么规约加法表达式的规则是什么呢？如果左右参数都是数字，那我们就能把它们加到一起，但如果其中一个或者所有参数需要规约怎么办？既然我们在考虑一小步一小步地进行规约，那就有必要在它们都符合规约条件的时候决定哪个参数先进行规约⁶。一个常用的策略是按照从左到右的顺序对参数进行规约，规则是这样的：

- 如果加法左边的参数能够规约，就规约左边的参数；
- 如果加法左边的参数不能规约，但是右边的参数可以规约，就规约右边的参数；
- 如果两边都不能规约，它们应该都是数字了，就把它们加到一起。

上面这些规则的结构是小步规约操作语义的特征。每一个规则都提供了它能得以应用的表达式模式（左边参数可规约的加法，右边参数可规约的加法，两边参数分别都不能规约的加法），还有对当模式匹配上之后如何构建一个规约后的新表达式的描述。选择了这些特定的规则之后，我们不仅确定了那些参数分别规约好之后应该如何合并到一起，还特别指出了一个 Simple 表达式要使用从左到右求值的方法对参数进行规约。

我们可以把这些规则直接翻译成一个 Add#reduce 的实现，同样的代码对 Multiply#reduce 也适用（别忘了要把参数乘起来而不是加起来）：

```
class Add
  def reduce
    if left.reducible?
      Add.new(left.reduce, right)
    elsif right.reducible?
      Add.new(left, right.reduce)
    else
      Number.new(left.value + right.value)
    end
  end
end
```

注 6：选择什么顺序并没有区别，但是在这个时候我们必须做出决策。

```
class Multiply
  def reduce
    if left.reducible?
      Multiply.new(left.reduce, right)
    elsif right.reducible?
      Multiply.new(left, right.reduce)
    else
      Number.new(left.value * right.value)
    end
  end
end
```



方法 #reduce 总是构建出新的表达式，而不是对已有的表达式进行修改。

为这几种表达式实现了 #reduce 方法之后，我们可以反复对其进行调用，从而通过很多的小步来完整地求出表达式的值：

```
>> expression =
  Add.new(
    Multiply.new(Number.new(1), Number.new(2)),
    Multiply.new(Number.new(3), Number.new(4))
  )
=> «1 * 2 + 3 * 4»
>> expression.reducerible?
=> true
>> expression = expression.reduce
=> «2 + 3 * 4»
>> expression.reducerible?
=> true
>> expression = expression.reduce
=> «2 + 12»
>> expression.reducerible?
=> true
>> expression = expression.reduce
=> «14»
>> expression.reducerible?
=> false
```



注意，#reduce 总是把一个表达式转换成另一个表达式，这正是小步规约操作语义应该遵守的规则。特别要注意的是，Add.new(Number.new(2),Number.new(12)).reduce 返回的 Number.new(14) 表示 Simple 表达式，而不仅仅是 14 这个 Ruby 中的数字。

Simple 语言（我们正在为其定义语义）和 Ruby 元语言（我们正在使用它定义语义）在明显不同的时候区分起来很容易——就像元语言是数学符号而不是一种程序设计语言时一样容易区分——但是这里因为两种语言看起来很像，所以需要更加小心。

我们在维护着一个状态——也就是当前表达式——并且对其反复调用 `#reducible?` 和 `#reduce`, 直到得到了一个值为止, 通过这种方式, 可以手工模拟一个抽象机器对表达式求值的操作。为了节省点力气, 也为了让这个抽象机器的思想更为具体, 我们可以轻松地写些 Ruby 代码。把这些代码和状态封装到一个类里, 并称为虚拟机:

```
class Machine < Struct.new(:expression)
  def step
    self.expression = expression.reduce
  end

  def run
    while expression.reducible?
      puts expression
      step
    end
    puts expression
  end
end
```

这允许我们用一个表达式实例化一个虚拟机, 让它运行 (`#run`), 并观察逐渐规约的各个步骤:

```
>> Machine.new(
  Add.new(
    Multiply.new(Number.new(1), Number.new(2)),
    Multiply.new(Number.new(3), Number.new(4)))
  ).run
1 * 2 + 3 * 4
2 + 3 * 4
2 + 12
14
=> nil
```

要扩展这个实现以支持其他简单的值和运算并不难: 减法和除法, 布尔值 `true` 和 `false`, 布尔运算 `and`、`or` 和 `not`, 对数字进行比较并返回布尔值的运算, 等等。例如, 下面是一个布尔值以及小于运算的实现:

```
class Boolean < Struct.new(:value)
  def to_s
    value.to_s
  end

  def inspect
    "#{self}"
  end

  def reducible?
    false
  end
end
```

```

class LessThan < Struct.new(:left, :right)
  def to_s
    "#{{left} < #{{right}}"
  end

  def inspect
    "#{{self}}"
  end

  def reducible?
    true
  end

  def reduce
    if left.reducible?
      LessThan.new(left.reduce, right)
    elsif right.reducible?
      LessThan.new(left, right.reduce)
    else
      Boolean.new(left.value < right.value)
    end
  end
  end
end

```

这仍然允许我们一小步一小步地规约布尔表达式：

```

>> Machine.new(
  LessThan.new(Number.new(5), Add.new(Number.new(2), Number.new(2)))
).run
5 < 2 + 2
5 < 4
false
=> nil

```

目前为止都是直截了当的东西：我们通过实现能对一种语言求值的虚拟机来定义它的操作语义。虚拟机当前的状态就是当前的表达式，而机器的行为是由一个规则集合来描述的，这个规则集合负责管理机器运行时的状态切换。我们已经把机器实现成了程序，这个程序跟踪当前表达式，持续对其进行规约，并随之更新表达式，直到没有更进一步的规约可以继续执行为止。

但是这种由简单代数表达式组成的语言不是十分有趣，这种语言没有几个我们期望拥有的哪怕是最简单编程语言中的特性。接下来我们把它构建得更复杂一些，让它看起来更像是一种能写出有用程序的语言。

首先，Simple 有一个明显缺失的东西：变量。在任何有用的语言中，我们都期望在讨论值时能够使用有意义的名字而不是它们本身的字面值。这些名字提供了一个间接层，这样同一个代码可以用来处理很多不同的值——包括来自于程序外部因而在写代码时甚至都不知道的值。

我们可以引入一个新的表达式类 Variable 来表示 Simple 中的变量：

```
class Variable < Struct.new(:name)
  def to_s
    name.to_s
  end

  def inspect
    "#<#{self}>"
  end

  def reducible?
    true
  end
end
```

为了能规约一个变量，抽象机器不仅仅需要存储当前表达式，还要存储从变量名称到它们值的映射——环境（environment）。在 Ruby 中，我们可以把这个映射实现成一个散列表（hash），其中用符号作为键，用表达式对象作为值；例如，散列表 {x:Number.new(2), y:Boolean.new(false) } 是一个环境，它分别把变量 x 和 y 与 Simple 的数字和布尔值进行了关联。



对这种语言来说，环境的目的只是把变量名映射到 Number.new(2) 这样不可规约的值上，而不是映射到 Add.new(Number.new(1), Number.new(2)) 这样可以规约的表达式。稍后我们编写能改变环境的规则时要注意这个约束。

有了环境，我们很容易实现 Variable#reduce：它只是在环境里查找变量的名字并返回其值。

```
class Variable
  def reduce(environment)
    environment[name]
  end
end
```

注意，我们正在把一个环境作为参数传进 #reduce，所以需要修改其他类的 #reduce 的实现，以便能接受和提供这个参数：

```
class Add
  def reduce(environment)
    if left.reducible?
      Add.new(left.reduce(environment), right)
    elsif right.reducible?
      Add.new(left, right.reduce(environment))
    else
      Number.new(left.value + right.value)
    end
  end
end
```

```

class Multiply
  def reduce(environment)
    if left.reducible?
      Multiply.new(left.reduce(environment), right)
    elsif right.reducible?
      Multiply.new(left, right.reduce(environment))
    else
      Number.new(left.value * right.value)
    end
  end
end

class LessThan
  def reduce(environment)
    if left.reducible?
      LessThan.new(left.reduce(environment), right)
    elsif right.reducible?
      LessThan.new(left, right.reduce(environment))
    else
      Boolean.new(left.value < right.value)
    end
  end
end

```

现在 #reduce 的所有实现更新之后都已经能支持环境了，因此还需要重新定义虚拟机，以便维持一个环境并把它提供给 #reduce：

```

Object.send(:remove_const, :Machine) # 忘记原来的 Machine 类

class Machine < Struct.new(:expression, :environment)
  def step
    self.expression = expression.reduce(environment)
  end

  def run
    while expression.reducible?
      puts expression
      step
    end
    puts expression
  end
end

```

机器对 #run 的定义仍然没变，但它有了一个新的环境属性，这个属性提供给 #step 方法新的实现使用。

现在只要我们也提供一个包含变量值的环境，就可以对包含变量的表达式进行规约了：

```

>> Machine.new(
  Add.new(Variable.new(:x), Variable.new(:y)),
  { x: Number.new(3), y: Number.new(4) }
).run

```

```
x + y  
3 + y  
3 + 4  
7  
=> nil
```

环境的引入完成了表达式的操作语义。我们已经设计了抽象机器，它由一个初始表达式和环境开始，然后在每次规约的一小步中使用当前的表达式和环境生成一个新的表达式，这个过程中环境始终没有改变。

2. 语句

现在我们可以看一下另一种程序结构的实现：语句。它是一个表达式，用来求值生成另一个表达式；换句话说，一个语句能够通过求值改变抽象机器的状态。机器唯一的状态（除了当前程序）就是环境，因此我们将允许 Simple 的语句生成一个新的环境以替换当前环境。

最简单的语句就是什么都不做的语句：它不能规约，因为对环境没有任何影响。这实现起来很简单：

```
class DoNothing ❶  
  def to_s  
    'do-nothing'  
  end  
  
  def inspect  
    "#{self}"  
  end  
  
  def ==(other_statement) ❷  
    other_statement.instance_of?(DoNothing)  
  end  
  
  def reducible?  
    false  
  end  
end
```

- ❶ 其他所有语法类都从 Struct 类继承，但是 DoNothing 没有继承任何类。这是因为 DoNothing 什么属性都没有，而且遗憾的是，Struct.new 还不让我们传一个空的属性名称列表。
- ❷ 想要比较任意两个语句是否相等。其他类都从 Struct 继承了 #== 的实现，但 DoNothing 只能定义它自己的了。

一个什么都不做的语句可能看起来没什么意义，但是能有一个特殊的语句表示程序已经执行成功会非常方便。其他语句完成了它们的工作之后，我们会将它们最终规约成 «do-nothing»。

要看个实用语句的例子，最简单的就是像 «`x = x + 1`» 这样的赋值语句，但在实现赋值语句之前，我们还需要决定它的规约规则。

一个赋值语句由一个变量名（`x`）、一个等号和一个表达式（«`x + 1`»）组成。如果赋值语句中的表达式是可规约的，我们就可以按照表达式规约规则对其进行规约并最终得到一个包含规约后表达式的新的赋值语句。例如，在一个变量 `x` 值为 «`2`» 的环境里对 «`x = x + 1`» 进行规约，我们会得到语句 «`x = 2 + 1`»，然后再把它规约就得到 «`x = 3`»。

可是然后呢？如果表达式已经是 «`3`» 这样的值了，那么我们就应该执行赋值，也就意味着对环境进行更新，即把这个值与适当的变量名关联起来。因此规约一个语句不单需要生成一个规约了的新语句，还要产生一个新的环境，这个环境有时候会与执行规约时的环境不同。



我们的实现将使用 `Hash#merge` 创建一个新的散列来更新环境，不会改变旧值：

```
>> old_environment = { y: Number.new(5) }
=> {:y=>«5»}
>> new_environment = old_environment.merge({ x: Number.new(3) })
=> {:y=>«5», :x=>«3»}
>> old_environment
=> {:y=>«5»}
```

可以选择破坏性地改变当前环境，而不是创建一个新的，但是避免破坏性的修改可以促使我们把 `#reduce` 的结果完全明确出来。如果 `#reduce` 想要改变当前的环境，它就得给调用者返回一个改变后的环境进行通知；反之，如果它不返回一个环境，那么就可以肯定没有造成任何变化。

这个约束帮助我们强化了表达式和语句的区别。对于表达式，把一个环境传递给 `#reduce`，然后得到一个规约了的表达式；因为没有返回一个新的环境，所以很明显规约一个表达式不会改变环境。对于语句，我们将用当前的环境调用 `#reduce`，然后得到一个新的环境，这表明规约一个语句会对环境有影响。（换句话说，Simple 小步语义的结构告诉我们：Simple 的表达式是纯净无害的，而它的语句不是这样。）

因此从一个空的环境规约 «`x = 3`» 应该会产生一个新的环境 { `x: Number.new(3)` }，但是我们还期望这个语句以某种方式得到规约；不然的话，抽象机器将会不断地把 «`3`» 赋值给 `x`。这时候 «`do-nothing`» 就派上用场了：一个完整的赋值语句规约成 «`do-nothing`»，就表明语句的规约已经结束，并且可以认为新环境中的东西就是执行结果。

总结起来，赋值的规约规则是：

- 如果赋值表达式能规约，那么就对其规约，得到的结果就是一个规约了的赋值语句和一个没有改变的环境；
- 如果赋值表达式不能规约，那么就更新环境把这个表达式与赋值的变量关联起来，得到的结果是一个 «do-nothing» 语句和一个新的环境。

这样，我们就有了实现一个赋值类 `Assign` 的足够信息。唯一的困难就是 `Assign#reduce` 需要既返回一个语句又返回一个环境——而 Ruby 的方法只能返回一个对象——但我们可以把它们放到由两个元素组成的数组中返回，这就模拟了这种情况。

```
class Assign < Struct.new(:name, :expression)
  def to_s
    "#{name} = #{expression}"
  end

  def inspect
    "<#{self}>"
  end

  def reducible?
    true
  end

  def reduce(environment)
    if expression.reducible?
      [Assign.new(name, expression.reduce(environment)), environment]
    else
      [DoNothing.new, environment.merge({ name => expression })]
    end
  end
end
```



正如我们承诺的那样，`Assign` 的规约规则保证了如果一个表达式不可规约（如一个值），它就只会增加到环境上。

可以像表达式一样对一个赋值语句反复规约，直到其不能再规约为止。通过这个方法就可以对一个赋值表达式求值。

```
>> statement = Assign.new(:x, Add.new(Variable.new(:x), Number.new(1)))
=> «x = x + 1»
>> environment = { x: Number.new(2) }
=> { :x=>«2»}
>> statement.reducible?
=> true
>> statement, environment = statement.reduce(environment)
=> [«x = 2 + 1», { :x=>«2»}]
>> statement, environment = statement.reduce(environment)
=> [«x = 3», { :x=>«2»}]
>> statement, environment = statement.reduce(environment)
=> [«do-nothing», { :x=>«3»}]
```

```
>> statement.reducerible?  
=> false
```

这个过程甚至比手工规约表达式更难，因此为了处理语句，需要重新实现虚拟机，让它能在每一步规约时显示当前的语句和环境：

```
Object.send(:remove_const, :Machine)  
  
class Machine < Struct.new(:statement, :environment)  
  def step  
    self.statement, self.environment = statement.reduce(environment)  
  end  
  
  def run  
    while statement.reducerible?  
      puts "#{statement}, #{environment}"  
      step  
    end  
  
    puts "#{statement}, #{environment}"  
  end  
end
```

现在这台机器又可以为我们工作啦：

```
>> Machine.new(  
  Assign.new(:x, Add.new(Variable.new(:x), Number.new(1))),  
  { x: Number.new(2) }  
)  
x = x + 1, {:x=>«2»}  
x = 2 + 1, {:x=>«2»}  
x = 3, {:x=>«2»}  
do-nothing, {:x=>«3»}  
=> nil
```

可以看到，这台机器仍然在执行表达式的规约步骤（«`x + 1`» 规约成 «`2 + 1`»，再规约成 «`3`»），但是这个规约过程现在不是发生在语法树的顶层，而是在一个语句里。

既然知道语句规约是如何工作的了，那么我们就可以对其进行扩展，以支持其他类型的语句。让我们从 «`if (x) { y = 1 } else { y = 2 }`» 这样的语句开始，这个语句包含了一个叫作条件（«`x`»）的表达式，还有两个语句，一个称为结果（«`y = 1`»），另一个是替代语句（«`y = 2`»）⁷。对条件进行规约的规则很简单：

- 如果条件能规约，那就对其进行规约，得到的结果是一个规约了的条件语句和一个没有改变的环境；
- 如果条件是表达式 «`true`» 了，就规约成结果语句和一个没有变化的环境；

注 7：此条件语句与 Ruby 的 `if` 不同，Ruby 中的 `if` 是返回一个值的表达式，但是在 Simple 中，这是一个语句，它从其他两个语句中选择一个求值，并且它唯一的结果就是对当前环境的影响。

- 如果条件是表达式 «false»，就规约成替代语句和一个没有变化的环境。

在这种情况下，所有规则都不会改变环境——第一条规则中对条件表达式的规约只会生成一个新的表达式，而不会产生新的环境。

下面是翻译成 If 类的规则：

```
class If < Struct.new(:condition, :consequence, :alternative)
  def to_s
    "if (#{condition}) { #{consequence} } else { #{alternative} }"
  end

  def inspect
    "#{self}"
  end

  def reducible?
    true
  end

  def reduce(environment)
    if condition.reducible?
      [If.new(condition.reduce(environment), consequence, alternative), environment]
    else
      case condition
      when Boolean.new(true)
        [consequence, environment]
      when Boolean.new(false)
        [alternative, environment]
      end
    end
  end
end
```

下面是规约操作：

```
>> Machine.new(
  If.new(
    Variable.new(:x),
    Assign.new(:y, Number.new(1)),
    Assign.new(:y, Number.new(2))
  ),
  { x: Boolean.new(true) }
).run
if (x) { y = 1 } else { y = 2 }, {x=>«true»}
if (true) { y = 1 } else { y = 2 }, {x=>«true»}
y = 1, {x=>«true»}
do-nothing, {x=>«true», :y=>«1»}
=> nil
```

这些都与预期一致，但如果能支持不带 «else» 从句的条件语句就好了，比如 «if (x) {y = 1}»。幸运的是，把语句写成 «if (x) { y = 1 } else { do-nothing }» 就可以做到，这和

没有 «else» 从句的效果是一样的：

```
>> Machine.new(
  If.new(Variable.new(:x), Assign.new(:y, Number.new(1)), DoNothing.new),
  { x: Boolean.new(false) }
).run
if (x) { y = 1 } else { do-nothing }, {:x=>«false»}
if (false) { y = 1 } else { do-nothing }, {:x=>«false»}
do-nothing, {:x=>«false»}
=> nil
```

既然不仅实现了表达式，还实现了赋值语句和条件语句，我们就有了组成程序所需要的基础材料，这样的程序可以执行计算和进行决策，做实际的工作。主要的限制是我们还不能把这些基础材料“连接”到一起：没有办法给多个变量赋值或者执行多个条件运算，这大幅度地限制了语言的可用性。

为摆脱这个限制我们可以再定义一种语句——序列（sequence），它把两个语句（如 « $x = 1 + 1$ » 和 « $y = x + 3$ »）连接到一起，组成一个更大的语句（如 « $x = 1 + 1; y = x + 3$ »）。一旦有了序列语句，我们就可以反复使用它们构建更大的语句；例如，序列 « $x = 1 + 1; y = x + 3$ » 和赋值语句 « $z = y + 5$ » 能连到一起组成序列 « $x = 1 + 1; y = x + 3; z = y + 5$ »⁸。

对序列进行规约的规则有点微妙：

- 如果第一条语句是 «do-nothing»，就规约成第二条语句和原始的环境；
- 如果第一条语句不是 «do-nothing»，就对其进行规约，得到的结果是一个新的序列（规约之后的第一条语句，后边跟着第二条语句）和一个规约了的环境。

看了代码你会更清楚这些规则：

```
class Sequence < Struct.new(:first, :second)
  def to_s
    "#{first}; #{second}"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    case first
    when DoNothing.new
```

注 8：为了达到我们的目的，这个语句构造出 « $(x = 1 + 1; y = x + 3); z = y + 5$ » 还是 « $x = 1 + 1; (y = x + 3; z = y + 5)$ » 都没有关系。在执行规约时，这个选择会影响规约的顺序，但是两种方式最终的结果是一样的。

```

        [second, environment]
    else
        reduced_first, reduced_environment = first.reduce(environment)
        [Sequence.new(reduced_first, second), reduced_environment]
    end
end
end

```

这些规则的总体效果就是：不断规约一个序列时，一直都在规约它的第一个语句，直到成为『do-nothing』，然后再去规约第二个语句。在虚拟机里运行一个序列，我们可以看到这种效果：

```

>> Machine.new(
  Sequence.new(
    Assign.new(:x, Add.new(Number.new(1), Number.new(1))),
    Assign.new(:y, Add.new(Variable.new(:x), Number.new(3)))
  ),
  {}
).run
x = 1 + 1; y = x + 3, {}
x = 2; y = x + 3, {}
do-nothing; y = x + 3, {:x=>«2»}
y = x + 3, {:x=>«2»}
y = 2 + 3, {:x=>«2»}
y = 5, {:x=>?2?}
do-nothing, {:x=>«2», :y=>«5»}
=> nil

```

Simple 里重要但仍缺失的只有某种无限制的循环结构了，所以为了完成任务，我们引入一个『while』语句，以便程序可以执行任意次数的重复计算⁹。像『while($x < 5$) { $x = x * 3$ }』这样的语句，包含了一个叫作条件（『 $x < 5$ 』）的表达式和一个叫作语句主体（body）的语句（『 $x = x * 3$ 』）。

为一个『while』语句写出正确的规约规则需要一点技巧。我们尝试着像『if』语句那样对其处理：如果能规约就对条件进行规约；不能的话，就根据条件是『true』还是『false』相应地规约语句主体或者执行『do-nothing』，那下一步会怎么样呢？条件已经被规约成一个值或者丢弃了，并且语句主体已经被规约成『do-nothing』，那么我们如何执行下一周期的循环呢？每一步规约要想与将来的规约步骤交流，只能通过产生一个新的语句和环境来实现，而使用这种方法，我们就没有地方记录最初条件和语句主体供下一个循环使用。

小步的解决方式¹⁰是使用序列语句把『while』的一个级别展开，把它规约成一个只执行一次循环的『if』语句，然后再重复原始的『while』。这意味着我们只需要一个规约规则：

注 9：使用序列语句，我们已经能够硬编码固定数量的重复操作了，但还是无法控制运行时的重复行为。

注 10：我们总试图把『while』的迭代行为直接构建成规约规则，而不是找到一种途径让抽象机器去处理它，但这不是小步语义的工作方式。参考 2.3.2 节，其中介绍的大步语义是一种让规则完成工作的语义。

- 把 «while (条件) { 语句主体 }» 规约成 «if (条件) { 语句主体 ; while (条件) { 语句主体 } } else { do-nothing }» 和一个没有改变的环境。

在 Ruby 中实现这个规则很容易：

```
class While < Struct.new(:condition, :body)
  def to_s
    "while (#{condition}) { #{body} }"
  end

  def inspect
    "#{self}"
  end

  def reducible?
    true
  end

  def reduce(environment)
    [If.new(condition, Sequence.new(body, self), DoNothing.new), environment]
  end
end
```

这给了虚拟机根据需要对条件和语句主体进行求值的机会：

```
>> Machine.new(
  While.new(
    LessThan.new(Variable.new(:x), Number.new(5)),
    Assign.new(:x, Multiply.new(Variable.new(:x), Number.new(3)))
  ),
  { x: Number.new(1) }
).run
while (x < 5) { x = x * 3 }, {:x=><<1>>}
if (x < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=><<1>>}
if (1 < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=><<1>>}
if (true) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=><<1>>}
x = x * 3; while (x < 5) { x = x * 3 }, {:x=><<1>>}
x = 1 * 3; while (x < 5) { x = x * 3 }, {:x=><<1>>}
x = 3; while (x < 5) { x = x * 3 }, {:x=><<1>>}
do-nothing; while (x < 5) { x = x * 3 }, {:x=><<3>>}
while (x < 5) { x = x * 3 }, {:x=><<3>>}
if (x < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=><<3>>}
if (3 < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=><<3>>}
if (true) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=><<3>>}
x = x * 3; while (x < 5) { x = x * 3 }, {:x=><<3>>}
x = 3 * 3; while (x < 5) { x = x * 3 }, {:x=><<3>>}
x = 9; while (x < 5) { x = x * 3 }, {:x=><<3>>}
do-nothing; while (x < 5) { x = x * 3 }, {:x=><<9>>}
while (x < 5) { x = x * 3 }, {:x=><<9>>}
if (x < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=><<9>>}
if (9 < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=><<9>>}
if (false) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=><<9>>}
do-nothing, {:x=><<9>>}
=> nil
```

或许这个规约规则看起来有点像是在逃避——好像我们总是在往后推迟对 «while» 的规约，一直没有实际进展——但它确实很好地解释了一个 «while» 语句真正的意思：检查条件，对语句主体求值，然后重新开始。奇怪的是，对 «while» 进行规约，会把它转换成一个语法上更庞大的程序，其中包括条件语句和序列语句，而不是直接对它的条件和语句主体进行规约，但有一个能定义一种语言形式语义的技术方案是非常好的，因为我们会更容易理解这种语言中的不同部分彼此之间是如何关联的。

3. 正确性

如果程序只是语法有效但实际上却是错误的，这时按照我们给出的语义执行会发生什么呢？我们之前完全忽视了这一点。语句 «`x = true; x = x + 1`» 是一段语法有效的 Simple 代码，我们确实可以构建一个抽象语法树来表示它，但试图反复对其规约的时候，它将会崩溃，因为在尝试往 «`true`» 上加 «`1`» 的时候抽象机器会终止。

```
>> Machine.new(
  Sequence.new(
    Assign.new(:x, Boolean.new(true)),
    Assign.new(:x, Add.new(Variable.new(:x), Number.new(1)))
  ),
  {}
).run
x = true; x = x + 1, {}
do-nothing; x = x + 1, {:x=>«true»}
x = x + 1, {:x=>«true»}
x = true + 1, {:x=>«true»}
NoMethodError: undefined method `+' for true:TrueClass
```

处理这个问题的一个方法就是在表达式能被规约的时候增加更多的约束，加入对求值失败可能性的考虑，这时求值过程有可能会中止，而不是总要试图规约成一个值（然后就可能在处理过程中崩溃）。我们本来可以把 `Add#reducible?` 实现成这样：«`+`» 的两个参数要么都是可规约的，要么都是数字类型 (`Number`) 实例，这时它才返回 `true`，这种情况下，表达式 «`true + 1`» 将会中止处理而永远不会变成一个值。

最终，我们需要一个比语法更强大的工具，它要能“看到未来”并让我们避免执行任何可能崩潰或者中止处理的程序。这一章是关于动态语义 (dynamic semantic) 的——程序执行时具体在做什么——但那并不是一个程序所拥有的唯一一种含义；在第 9 章，我们将研究静态语义 (static semantic)，看看如何根据语言的动态语义来判断一个语法上有效的程序是否具有有用的含义。

4. 应用

我们定义的程序设计语言非常基本，但在写下所有规约规则的时候，仍然不得不做了一些设计上的决策并明确地表述它们。例如，与 Ruby 不同的是，Simple 这种语言会区分表达式和语句，前者返回一个值，后者不会返回值；与 Ruby 相同的是，Simple 的环境只与已

经完全规约成值的变量关联，而不与仍然有待执行的更大表达式关联¹¹。我们可以通过给出不同的小步语义来改变上面任何的策略，这将描述一种新的语言，这种语言拥有同样的语法，但有着不同的运行时行为。如果向语言中增加更多精心设置的特性——数据结构、过程调用、异常和一个对象系统——我们需要做出更多的设计决策并在定义语义时无歧义地表达它们。

小步语义的细节化、面向执行的风格能让它无歧义地定义真实世界的编程语言。例如，Scheme 编程语言最新的 R6RS 标准使用了小步语义 (<http://www.r6rs.org/final/html/r6rs/r6rs-Z-H-15.html>) 描述其执行，并提供了 PLT Redex 语言 (<http://redex.racket-lang.org/>) (设计用来定义和调试操作语义的一门特定领域的语言) 对那些语义的参考实现 (<http://www.r6rs.org/refimpl>)。OCaml 编程语言，在一个更简单的 Core ML 语言基础之上构建了一系列的分层，也有对于基础语言运行时行为的小步语义定义 (<http://caml.inria.fr/pub/docs/u3-ocaml/ocaml-ml.html#htoc5>)。

参考 6.2.2 节，那里还有一个小步操作语义的例子，它用了一个甚至更简单的叫作 lambda 演算的编程语言定义了表达式的含义。

2.3.2 大步语义

我们已经看到了小步操作语义是什么样子的：设计一台抽象机器维护一些执行状态，然后定义一些规约规则，这些规则详细说明了如何才能对每种程序结构循序渐进地求值。特别地，小步语义大部分都带有迭代的味道，它要求抽象机器反复执行规约步骤 (Machine#run 中的 while 循环)，这些步骤以及与它们同样类型的信息可以作为自身的输入和输出，这让它们适合这种反复进行的应用程序。¹²

这种小步的方法有一个优势，就是能把执行程序的复杂过程分成更小的片段解释和分析，但它确实有点不够直接：我们没有解释整个程序结构是如何工作的，而只是展示了它是如何慢慢规约的。为什么不能更直接地解释一个语句，完整地说明它的执行过程呢？好吧，我们可以，而这正是大步语义 (big-step semantic) 的依据。

大步语义的思想是，定义如何从一个表达式或者语句直接得到它的结果。这必然需要把程序的执行当成一个递归的而不是迭代的过程：大步语义说的是，为了对一个更大的表达式求值，我们要对所有比它小的子表达式求值，然后把结果结合起来得到最终答案。

在很多方面，这都比小步的方法更自然，但确实失去了一些对细节的关注。例如，小步语义明确定义了操作应该发生的顺序，因为在每一步都明确了下一步规约应该是什么。但是

注 11：Ruby 的 proc 在某种意义上允许把复合表达式复制给变量，但是一个 proc 仍然是一个值：它本身不能再执行任何求值操作了，但是能和其他值一起作为一个更大表达式的一部分进行规约。

注 12：对一个表达式和一个环境进行规约将得到一个新的表达式，而且下一次还可以重用旧的环境；对一个语句和一个环境进行规约将得到一个新的语句和一个新的环境。

大步语义经常会写成更为松散的形式，只会说哪些子计算会执行，而不会指明它们按什么顺序执行。¹³ 小步语义还提供一种轻松的方式用以监视计算的中间阶段，而大步语义只是返回一个结果，不会产生任何关于如何计算的证据。

为了理解做出的这种权衡，让我们回顾一些常见的语言结构，并看如何在 Ruby 中实现它们的大步语义。我们的小步语义要求有一个 `Machine` 类跟踪状态并反复执行规约，但是这里不需要这个类了；大步规约的规则描述了如何只对程序的抽象语法树访问一次就计算出整个程序的结果，因此不需要处理状态和重复。我们将只对表达式和语句类定义一个 `#evaluate` 方法，然后直接调用它。

1. 表达式

处理小步语义时，我们不得不区分像 «`1 + 2`» 这样可规约的表达式和像 «`3`» 这样不可规约的表达式，这样规约规则才能识别一个子表达式什么时候可以用来组成更大的程序。但是在大步语义中，每个表达式都能求值。唯一的区别，如果我们想要有个区别的话，就是对一些表达式求值会直接得到它们自身，而对另一些表达式求值会执行一些计算并得到一个不同的表达式。

大步语义的目标是像小步语义那样对一些运行时行为进行建模，这意味着我们期望对于每一种程序结构，大步语义规则都要与小步语义规则程序最终生成的东西保持一致。（把操作语义写成数学形式之后，这是能被准确证明的。）小步语义规则规定，像数值（`Number`）和布尔值（`Boolean`）这样的值不能再规约了，因此它们的大步规约非常简单：求值的结果直接就是它们本身。

```
class Number
  def evaluate(environment)
    self
  end
end

class Boolean
  def evaluate(environment)
    self
  end
end
```

变量（`Variable`）表达式是唯一的，这样它们的小步语义允许它们在成为一个值之前只规约一次，所以它们的大步语义规则与小步规则一样：在环境中查找变量名然后返回它的值。

```
class Variable
  def evaluate(environment)
    environment[name]
```

注 13：我们用这种方法实现的大步语义不会有二义性，因为 Ruby 本身已经进行了排序决策，但是在数学化地定义大步语义时，就不可避免地要讲清楚准确的求值策略了。

```
    end  
end
```

二元表达式 Add、Multiply 和 LessThan 更有意思，它们要求先对左右子表达式递归求值，然后再用恰当的 Ruby 运算合并两边的结果值：

```
class Add  
  def evaluate(environment)  
    Number.new(left.evaluate(environment).value + right.evaluate(environment).value)  
  end  
end  
  
class Multiply  
  def evaluate(environment)  
    Number.new(left.evaluate(environment).value * right.evaluate(environment).value)  
  end  
end  
  
class LessThan  
  def evaluate(environment)  
    Boolean.new(left.evaluate(environment).value < right.evaluate(environment).value)  
  end  
end
```

为了检查这些大步的表达式语义是否正确，下面将在 Ruby 的控制台验证一下：

```
>> Number.new(23).evaluate({})  
=> «23»  
>> Variable.new(:x).evaluate({ x: Number.new(23) })  
=> «23»  
>> LessThan.new(  
  Add.new(Variable.new(:x), Number.new(2)),  
  Variable.new(:y))  
) .evaluate({ x: Number.new(2), y: Number.new(5) })  
=> «true»
```

2. 语句

在我们要定义语句的行为时，这种类型的语义就能发挥作用了。在小步语义下表达式会规约成其他表达式，但语句会规约成 «do-nothing» 并且得到一个经过修改的环境。我们可以把大步语义的语句求值看成一个过程，这个过程总是把一个语句和一个初始环境转成一个最终的环境，这避免了小步语义不得不对 #reduce 产生的中间语句进行处理的复杂性。例如，对一个赋值语句按照大步的方法求值应该完整地对其表达式求值，并返回一个包含结果值的更新了的环境：

```
class Assign  
  def evaluate(environment)  
    environment.merge({ name => expression.evaluate(environment) })  
  end  
end
```

类似地，DoNothing#evaluate 无疑将把未更改的环境返回，而 If#evaluate 的工作相当地直

接：对条件求值，然后把环境返回，这个环境来自于对序列或者替代语句求值得到的结果。

```
class DoNothing
  def evaluate(environment)
    environment
  end
end

class If
  def evaluate(environment)
    case condition.evaluate(environment)
    when Boolean.new(true)
      consequence.evaluate(environment)
    when Boolean.new(false)
      alternative.evaluate(environment)
    end
  end
end
```

有两种有趣的情况就是序列语句和『while』循环表达式。对于序列，我们只需要对两个语句求值，但是初始环境需要“穿过”这两个求值过程，这样第一个语句求值的结果就能成为第二个语句求值的环境。这可以写成 Ruby 代码：用第一次求值的结果作为第二次求值的参数：

```
class Sequence
  def evaluate(environment)
    second.evaluate(first.evaluate(environment))
  end
end
```

为了让先前的语句为后边的做准备，“穿过”环境是至关重要的：

```
>> statement =
  Sequence.new(
    Assign.new(:x, Add.new(Number.new(1), Number.new(1))),
    Assign.new(:y, Add.new(Variable.new(:x), Number.new(3)))
  )
=> «x = 1 + 1; y = x + 3»
>> statement.evaluate({})
=> { :x=>«2», :y=>«5»}
```

对于『while』语句，我们需要彻底想清楚对一个循环完整求值的各个阶段：

- 对条件求值，得到『true』或者『false』；
- 如果条件求值结果是『true』，就对语句主体求值得到一个新的环境，然后在那个新的环境下重复循环（也就是说对整个『while』语句再次求值），最后返回作为结果的环境；
- 如果条件求值结果是『false』，就返回未修改的环境。

这是对一个『while』语句行为的递归解释。就像序列语句，循环体生成的更新了的环境被

下一个迭代使用这一点非常重要；不然的话，条件一直都是 «true»，那么循环就永远也没有机会停下来了。¹⁴

知道了大步 «while» 语义的行为表现之后，就可以实现 While#evaluate 了：

```
class While
  def evaluate(environment)
    case condition.evaluate(environment)
    when Boolean.new(true)
      evaluate(body.evaluate(environment)) ❶
    when Boolean.new(false)
      environment
    end
  end
end
```

- ❶ 循环在这里发生：body.evaluate(environment) 对循环求值得到一个新的环境，然后我们把那个环境传回当前方法中开始下一次迭代。这意味着可能会堆积很多对 While#evaluate 的嵌套调用，直到条件最后成为 «false» 然后返回最后的环境。



就像任何递归代码一样，如果调用嵌套得太深可能会导致 Ruby 调用栈溢出。一些 Ruby 的实现会实验性地支持对尾调用的优化，这个技术能通过尽可能重用同样的栈帧来减少溢出风险。在 Ruby 的官方实现（MRI）里，我们可以这样打开尾调用优化：

```
RubyVM::InstructionSequence.compile_option = {
  tailcall_optimization: true,
  trace_instruction: false
}
```

为了确认生效，可以尝试对同样的 «while» 语句求值，这是之前用来检查小步语义的：

```
>> statement =
  While.new(
    LessThan.new(Variable.new(:x), Number.new(5)),
    Assign.new(:x, Multiply.new(Variable.new(:x), Number.new(3)))
  )
=> «while (x < 5) { x = x * 3 }»
>> statement.evaluate({ x: Number.new(1) })
=> {:x=>«9»}
```

这与小步语义给出的结果一致，所以看起来 While#evaluate 做的事情没错。

3. 应用

我们稍早时候对小步语义的实现只是适度使用了 Ruby 调用栈：在对一个大型程序调用

注 14：当然，没有什么能够阻止 Simple 程序员写出条件永远也不会为 «false» 的 «while» 语句，但如果那就是他们想要的，那也是可行的。

`#reduce` 时，消息会遍历抽象树直到其到达一段准备好规约的代码，这会引起一系列对 `#reduce` 的嵌套调用。¹⁵ 但是伴随着反复执行小步规约，虚拟机通过维护当前程序和环境完成了对整个计算过程的跟踪；值得一提的是，嵌套调用只是用来遍历语法树查找下一步的规约对象，而不是执行规约本身，因此调用栈的深度受到程序语法树深度的限制。

相比之下，大步方式的实现会执行较小规模的计算，并将其作为更大规模计算的一部分。为了跟踪还有多少求值工作要做，它使用了更多的栈，并完全依赖栈来记住当前处理在整个计算中的位置。看上去像是对 `#evaluate` 的一次调用，实际上转换成了一系列递归调用，每一次调用都对一个子程序求值，这都让其在语法树中更进一步。

这个差别突出了每一种方法的目的。小步语义设定了一台能执行小操作的简单抽象机器，因此它包含了关于如何产生有用中间结果的详尽细节；大步语义把汇编整个计算的重担交给了机器或者执行它的人，在仅通过一步操作就把整个程序转换成一个最终结果的过程中，要求它跟踪许多中间子目标。根据我们想用一个语言的操作语义干什么——或是构建一个高效的实现，证明程序的某些属性，或是设计某个最佳变换——可能采用其中一种方法或者另一种方法会更合适。

大步语义在定义真正程序设计语言上最有影响的应用是第 6 章提到的标准 ML 编程语言 (<http://www.lfcs.inf.ed.ac.uk/reports/87/ECS-LFCS-87-36/>) 的原始定义，它用大步方式定义了 ML 的所有运行时行为。在这个例子之后，OCaml 的核心语言用大步语义 (<http://caml.inria.fr/pub/docs/u3-ocaml/ocaml-ml.html#htoc7>) 补足了它更细节的小步定义。

W3C 也用到了大步操作语义：XQuery 1.0 和 XPath 2.0 规范 (<http://www.w3.org/TR/xquery-semantics/>) 使用数学化的推理规则描述它的语言应该如何求值，并且 XQuery 和 XPath 规范全文的 3.0 版本 (<http://www.w3.org/TR>xpath-full-text-30/>) 包括了一个使用 XQuery 写成的大步语义。

你可能注意到了，通过使用 Ruby 语言而不是数学语言写下 Simple 的小步和大步语义，我们已经为它实现了两个不同的 Ruby 解释器。操作语义实质上是这样的：通过描述一个解析器来说明一种语言的含义。正常情况下，这个描述应该用简单的数学符号来写，只要我们能理解，这将使一切都清晰而且无歧义，但是这样过于抽象而且离现实中的计算机有一定距离。把一种真实世界编程语言的额外复杂性（类、对象、方法调用……）引入到本该简约的说明当中，这是 Ruby 语言的缺点，但是如果我们就理解 Ruby，那么就更容易理解整个过程，并且能够执行的描述可以当作一个解释器，这是个很好的红利。

注 15：有一种操作语义的替换形式，叫作规约语义，它通过引入所谓的规约上下文，把“下一步规约什么”和“如何对其进行规约”分离开来。这些上下文只是一些简明描述了规约在程序中何处发生的模式。这意味着我们只需要写真正执行计算的规约规则，从而把一些样板文件（boilerplate）从更大型的语言中去掉。

2.4 指称语义

到目前为止，我们已经从操作性方面观察了程序设计语言的含义，它通过展示程序执行之后发生的事情解释了程序的含义。而指称语义（denotational semantic）转而关心从程序本来的语言到其他表示的转换。

这种类型的语义没有直接处理程序的执行，而是关注如何借助另一种语言的已有含义——一种低级的、更形式化的或者至少比正在描述的语言更好理解的语言——解释一个新的语言。

指称语义确实是一种比操作语义更抽象的方法，因为它只是用一种语言替换另一种语言，而不是把一种语言转换成真实的行为。例如，如果我们需要向一个人解释英语动词“walk”的含义，但和他没有共同的口头语言，可以通过来回走的动作来沟通。另一方面，如果我们需要向一个说法语的人解释“walk”，可以跟他讲“marcher”——不可否认这是一种更高层次的沟通方式，不需要麻烦地运动了。

指称语义通常用来把程序转成数学化的对象，所以不出意料，可以用数学工具研究和控制它们，但是我们可以看看如何用另一种方式表示 Simple 程序，借此大致了解指称语义。

把 Simple 转成 Ruby 从而得到 Simple 语言的指称语义，¹⁶事实上，这意味着把一个抽象语法树转成一个 Ruby 代码的字符串。不管怎样，我们得到了那种语法本来的含义。

但“本来的含义”是什么呢？我们表达式和语句的 Ruby 指称（denotation）是什么样的呢？从操作上我们已经看到一个表达式使用一个环境（environment）然后把它转成一个值；在 Ruby 中表达这个过程的一种方式是用一些参数表示环境参数，然后返回一些表示值的 Ruby 对象。对于像 «5» 和 «false» 这样简单的常量表达式，我们根本无需使用环境，而只需要关心它们最终的结果如何能表示成一个 Ruby 对象。幸运的是，Ruby 已经设计了专门的对象表示这些值：我们可以使用 Ruby 值 5 作为 Simple 表达式 «5» 的结果，同样地，把 Ruby 的值 false 作为 «false» 的结果。

2.4.1 表达式

我们可以用这个思想为 Number 类和 Boolean 类写一个 #to_ruby 的实现：

```
class Number
  def to_ruby
    "-> e { #{value.inspect} }"
  end
end
```

注 16：这意味着我们将用 Ruby 代码生成 Ruby 代码，但是选择用同样的指称语言和实现元语言只是为了让事情简单。例如我们很容易用 Ruby 写出能生成包含 JavaScript 字符串的代码来。

```
class Boolean
  def to_ruby
    "-> e { #{value.inspect} }"
  end
end
```

下面在控制台运行它们：

```
>> Number.new(5).to_ruby
=> "-> e { 5 }"
>> Boolean.new(false).to_ruby
=> "-> e { false }"
```

这些方法每个都产生一个刚好包含 Ruby 代码的字符串，并且因为 Ruby 是一种我们已经理解其含义的语言，所以可以看到这些字符串都是构造 proc 的程序。每一个 proc 都带有一个叫 e 的环境参数，它们完全忽略这个参数而直接返回一个 Ruby 值。

因为这些符号都是 Ruby 代码组成的字符串，所以可以使用 Kernel#eval 转换成可调用的 Proc 对象实际执行，然后在 IRB 中检查它们的行为¹⁷：

```
>> proc = eval(Number.new(5).to_ruby)
=> #<Proc (Lambda)>
>> proc.call({})
=> 5
>> proc = eval(Boolean.new(false).to_ruby)
=> #<Proc (lambda)>
>> proc.call({})
=> false
```



现阶段，完全避免 proc，而使用更简单的 #to_ruby 实现是很诱人的，这只需要把 Number.new(5) 转换成字符串 '5' 而不是 '-> e { 5 }' 等，但是从源语言结构中获得其本质语义是指称语义这一方法的一部分，那么我们需要知道，即便某些特定的表达式不会用到环境，通常的表达式也还是需要一个环境的。

为了表示确实使用环境的表达式，我们需要决定如何用 Ruby 表示环境（environment）。在研究操作语义时我们已经了解了环境，那么既然它们已经用 Ruby 实现了，现在可以重用早期的思想——把一个环境表示成一个散列表。不过细节需要做一些改动，因此要注意其中微妙的差别：在我们的操作语义中，环境是生存在虚拟机中的，并且把变量名与 Number.new(5) 这样的 Simple 抽象语法树联系起来；但在我们的指称语义中，环境存在于我们要把程序转换得到的语言中，因此要在那个世界而不是在一个虚拟机的“外部世界”起作用。

注 17：只有 Ruby 既做实现语言又作为指称语言的时候我们才能这么做。如果指称是 JavaScript 源代码，我们就得到 JavaScript 的控制台去实验它们了。

注意，这意味着指称环境（denotational environment）应该把变量名与 5 这样的原生 Ruby 值，而不是与表示 Simple 语法的对象关联起来。我们把 { x: Number.new(5) } 这样的操作环境（operational environment）看成在要转换成的语言中拥有指称 '{ x: 5 }'，并且因为实现的元语言和指称语言正好都是 Ruby，所以不必有什么顾忌。

既然知道环境将是一个散列，那么就可以实现 Variable#to_ruby 了：

```
class Variable
  def to_ruby
    "-> e { e[#{name.inspect}] }"
  end
end
```

这段代码，把一个变量表达式转换成一个在环境散列中查找合适值的 Ruby proc：

```
>> expression = Variable.new(:x)
=> «X»
>> expression.to_ruby
=> "-> e { e[:x] }"
>> proc = eval(expression.to_ruby)
=> #<Proc (lambda)>
>> proc.call({ x: 7 })
=> 7
```

关于指称语义重要的一点是它是组合式的：一个程序的指称由组成它的各部分的指示构成。在开始指称（denotating）Add、Multiply 和 LessThan 这样的更大表达式时，我们就能理解这种合成性了：

```
class Add
  def to_ruby
    "-> e { (#{left.to_ruby}).call(e) + (#{right.to_ruby}).call(e) }"
  end
end

class Multiply
  def to_ruby
    "-> e { (#{left.to_ruby}).call(e) * (#{right.to_ruby}).call(e) }"
  end
end

class LessThan
  def to_ruby
    "-> e { (#{left.to_ruby}).call(e) < (#{right.to_ruby}).call(e) }"
  end
end
```

这里使用字符串串联操作把子表达式的指称组成一个大表达式的指称。我们知道每一个子表达式都将在 Ruby 源码中用一个 proc 表示，因此可以将它们作为更大段 Ruby 代码的一部分，那些更大段的代码使用提供的环境调用这些 proc，并使用它们返回的值进行一些计

算。下面是得到结果：

```
>> Add.new(Variable.new(:x), Number.new(1)).to_ruby
=> "-> e { (-> e { e[:x] }).call(e) + (-> e { 1 }).call(e) }"
>> LessThan.new(Add.new(Variable.new(:x), Number.new(1)), Number.new(3)).to_ruby
=> "-> e { (-> e { (-> e { e[:x] }).call(e) + (-> e { 1 }).call(e) }).call(e) < (-> e {
3 }).call(e) }"
```

这些指称已经够复杂的了，很难了解它们做的事情是否正确。让我们运行它们确认一下：

```
>> environment = { x: 3 }
=> {:x=>3}
>> proc = eval(Add.new(Variable.new(:x), Number.new(1)).to_ruby)
=> #<Proc (lambda)>
>> proc.call(environment)
=> 4
>> proc = eval(
    LessThan.new(Add.new(Variable.new(:x), Number.new(1)), Number.new(3)).to_ruby
)
=> #<Proc (lambda)>
>> proc.call(environment)
=> false
```

2.4.2 语句

我们可以用类似的方式定义语句的指称语义，但是要记住操作语义中提到的：对一个语句求值产生的是一个新的环境而不是一个值。这意味着 Assign#to_ruby 需要为 proc 构造一些代码，以使结果是一个更新了的环境散列：

```
class Assign
  def to_ruby
    "-> e { e.merge({ #{name.inspect} => (#{expression.to_ruby}).call(e) }) }"
  end
end
```

还是可以在控制台对其进行检查：

```
>> statement = Assign.new(:y, Add.new(Variable.new(:x), Number.new(1)))
=> «y = x + 1»
>> statement.to_ruby
=> "-> e { e.merge({ :y => (-> e { (-> e { e[:x] }).call(e) + (-> e { 1 }).call(e) })
.call(e) }) }"
>> proc = eval(statement.to_ruby)
=> #<Proc (lambda)>
>> proc.call({ x: 3 })
=> {:x=>3, :y=>4}
```

和之前一样，DoNothing 的语义非常简单：

```
class DoNothing
  def to_ruby
```

```
'-> e { e }'  
end  
end
```

对于条件语句，我们可以把 Simple 的 «if (...) { ... } else { ... }» 转换成一个 Ruby 的 if ... then ... else ... end，确保环境传到了需要它的地方：

```
class If  
  def to_ruby  
    "-> e { if (#{}{condition.to_ruby}).call(e)" +  
    " then (#{}{consequence.to_ruby}).call(e)" +  
    " else (#{}{alternative.to_ruby}).call(e)" +  
    " end }"  
  end  
end
```

就像在大步操作语义中一样，我们需要小心地定义序列语句：对第一个语句求值的结果作为对第二个语句求值时的环境。

```
class Sequence  
  def to_ruby  
    "-> e { (#{}{second.to_ruby}).call((#{}{first.to_ruby}).call(e)) }"  
  end  
end
```

最后，就像处理条件语句那样，我们可以把 «while» 语句转成 proc，在返回最终环境之前，它使用 Ruby 的 while 重复执行语句主体：

```
class While  
  def to_ruby  
    "-> e {" +  
    " while (#{}{condition.to_ruby}).call(e); e = (#{}{body.to_ruby}).call(e); end;" +  
    " e" +  
    " }"  
  end  
end
```

哪怕是一个简单的 «while» 都具有一个冗长的表示，所以有必要用 Ruby 解释器检查一下它的含义正确与否：

```
>> statement =  
  While.new(  
    LessThan.new(Variable.new(:x), Number.new(5)),  
    Assign.new(:x, Multiply.new(Variable.new(:x), Number.new(3)))  
  )  
>> «while (x < 5) { x = x * 3 }»  
>> statement.to_ruby  
>> "-> e { while (-> e { (-> e { e[:x] }).call(e) < (-> e { 5 }).call(e) }).call(e);  
e = (-> e { e.merge({ :x => (-> e { (-> e { e[:x] }).call(e) * (-> e { 3 }).call(e)  
}).call(e) }) }).call(e); end; e }"  
>> proc = eval(statement.to_ruby)
```

```
=> #<Proc (lambda)>
>> proc.call({ x: 1 })
=> {x=>9}
```

语义类型比较

«while» 是一个区分小步语义、大步语义和指称语义的好例子。

«while» 的小步操作语义是以一台抽象机器的归约规则形式写成的。整个循环并不是规约行为的一部分——规约只是把一个 «while» 语句转成一个 «if» 语句——但是它会作为将来由机器执行的规约序列的一部分。为了理解 «while» 做了什么，我们需要考虑所有的小步规则，并弄懂随着一个 Simple 程序的执行它们之间是如何互相作用的。

«while» 的大步操作语义是以一个求值规则的形式写成的，这个规则说明如何把最终的环境直接计算出来。这个规则包含了对其本身的递归调用，因此明显表明 «while» 在求值过程中会引发一个循环，但不是 Simple 程序员熟悉的那种循环。大步的规则是递归的形式，描述了如何根据对其他语法结构的求值对一个表达式或者语句完整地求值，因此这个规则告诉我们，对一个 «while» 语句求值的结果可能会依赖于一个不同环境下同样语句的求值结果，但把这种思想与 «while» 应该展现的迭代方式联系起来需要跳跃性思维。幸运的是这种跳跃并不太大：一点点的数学推理可以表明两种类型的循环在本质上是等价的，并且在元语言支持尾调用优化的时候，它们事实上也是等价的。

«while» 的指称语义展示了如何用 Ruby 对其重写，也就是如何通过 Ruby 的 while 关键字对其进行重写。这是一个简单直接得多的转换：Ruby 提供对迭代循环的原生支持，而指称规则也表明 «while» 能用 Ruby 的这个特性实现。要理解这两种类型的循环没有什么困难，所以如果我们理解了 Ruby 中 while 循环的工作方式，也能理解 Simple 的 «while» 循环。当然，这意味着我们已经把理解 Simple 的问题转换成了理解指称语言的问题，而如果指称语言像 Ruby 一样庞大而且定义不良，这就是一个严重的缺点；但在有一个能用来写指称的小型数学语言时，这就成了一个优点。

2.4.3 应用

做完所有这些工作之后，指称语义完成了什么目标呢？它的主要目的是展示如何把 Simple 翻译成 Ruby，它将后者作为工具来解释不同的语言结构是什么意思。这恰巧给了我们执行 Simple 程序的一种途径——因为已经用可执行的 Ruby 写下了指称语义的规则，而且这些规则的输出本身就是可执行的 Ruby——但这只是偶然事件，因为我们之前有可能用普通的英语写规则并用一些数学语言写下指称。真正重要的是我们自己随意设计了一种语言，并把它转换成一种其他人或者其他东西能理解的语言。

为了赋予这种转换一些解释能力，把一部分语言含义放到表面而不再只是隐含在背后会非常有帮助。例如，这种语义把环境表示成具体的 Ruby 对象——在 proc 中传入和返回的散列，而不是把 Simple 中的变量表示成真正的 Ruby 变量，然后依赖 Ruby 自己微妙的变量作

用域规则去定义 Simple 的变量访问机制；这样表示环境更为明确直接。在这方面这种语义除了把解释性的工作交给 Ruby，还多做了一些事情；它把 Ruby 作为一个简单的基础，但是在表面做了一些额外的工作，从而准确地展示了不同程序结构是如何使用和改变环境的。

这之前我们看到过，操作语义通过为一种语言设计一个解释器来解释这种语言的含义。与此对比，语言到语言的指称语义更像是一个编译器：在这种情况下，我们的 `#to_ruby` 实现高效地把 Simple 编译成 Ruby。这些类型的语义虽然都对如何为一种语言高效地实现一个解释器或者编译器只字不提，但确实提供了一个基础标准可以检验任何生效了的实现。

这些指称的定义还在一些语言的原始状态中出现过。早期版本的 Scheme 标准使用指称语义 (http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-10.html#%25_sec_7.2) 定义核心语言，而不像现在的标准使用小步操作语义来定义，并且 XSLT 文本转换语言的开发是由 Philip Wadler 对 XSLT 模式 (<http://homepages.inf.ed.ac.uk/wadler/topics/xml.html#xsl-semantics>) 和 XPath 表达式 (<http://homepages.inf.ed.ac.uk/wadler/topics/xml.html#xpath-semantics>) 的指称定义来引导的。

3.3.2 节有一个实际使用指称语义定义正则表达式的例子。

2.5 形式化语义实践

对于为计算机程序赋予含义的问题，本章已经展示了几种不同的方法。在每种情况下，我们都已经避免了数学化的方法并使用 Ruby 了解了它们的策略，但是形式化的语义通常都是由数学化的工具完成的。

2.5.1 形式化

我们对形式语义的研究并不是特别正式。一直没有认真关注过数学符号，而使用 Ruby 作为元语言意味着比起理解程序的各种方式，我们更关注执行程序的不同方式。合适的指称语义关注的是通过把程序转换成定义良好的数学对象以获得程序的核心含义，关心的是把一个 Simple 的 «while» 语句无歧义的完整表示成一个 Ruby 的 `while` 循环。



为了提供对指称语义有用的规定和对象，专门发展了称为域理论的数学分支，它采用基于单调函数上不动点的一种计算模型，并且这个单调函数定义在偏序集合上。我们可以通过把程序“编译”成数学函数来理解这个程序，并且域理论的技巧还能用来证明这些函数一些有趣的特性。

另一方面，尽管我们只是用 Ruby 含糊地概括了一下指称语义，但关于操作语义，我们已经在精神上接近它的形式化表示了：我们对方法 `#reduce` 和 `#evaluate` 的定义实际上只是用 Ruby 翻译的数学化推理规则。

2.5.2 找到含义

形式化语义的一个重要应用是为一种编程语言的含义给出一个无歧义的定义，而不是让其依赖于像自然语言规范文档和“由实现规范”这样更加随意的方法。形式化的定义还有其他用途，例如证明某种语言通常情况下的特性，以及特定程序在特定情况下的特性，证明语言中程序之间的等价性，研究如何在不改变程序行为的情况下安全地变换程序而使其效率更高。

例如，既然操作语义与解释器的实现极为接近，那么计算机科学家就可以把一个适当的解释器看成一种语言的操作语义，然后证明它在那种语言的指称语义方面的正确性——这意味着证明了由解释器给出的含义和由指称语义给出的含义之间存在着明显的联系。

指称语义的一个优点是比操作语义抽象层次更高，它忽略了程序如何执行的细节，而只关心如何把它转换成一个不同的表示。例如，如果存在一种指称语义可以把两种语言翻译成某种共通的表示，就使对不同语言写成的两个程序进行比较成为可能。

抽象程度会使指称语义看起来有点兜圈子。如果问题是如何解释一种程序设计语言的含义，那么把一种语言翻译成另一种语言是如何让我们更接近问题答案的呢？一个指称只不过与它的含义一样好；尤其是，如果指称的语言有某种操作性的含义，那么一个指称语义只是让我们更接近于能实际执行一个程序，这个语言的语义本身展示了它是如何执行的，而不是如何翻译成另一种语言的。

形式化的指称语义使用抽象的数学对象（通常是函数）来表示表达式和语句这样的编程语言结构，并且因为数学上的约定会规定如何对函数求值这样的事情，这就有了一种直接在操作意义上思考指称的方式。我们已经使用了不太正式的方式，把指称语义看成是一种语言到另一种语言的编译器，而事实上这是多数编程语言最终得以执行的方式：一个 Java 程序将会由 javac 编译成字节码，字节码将被 java 的虚拟机即时编译成 x86 的指令，然后一个 CPU 会把每一条 x86 指令解码成类 RISC（精简指令集）的微指令放到一个核上去执行……它会在什么地方结束呢？是编译器，还是虚拟机，还是一直重复下去？

当然程序最终会执行，因为语义这个高机会到达底部暴露出实际的机器：半导体中的电子，它们遵守的是物理法则。¹⁸一台计算机是维护这个不确定结构的装置，大量复杂的解释层在彼此之上保持稳定平衡，这就允许多点触控手势这样人体尺度的想法和 while 循环这样的想法，都能被逐渐地向下翻译给硅和电的物理世界。

2.5.3 备选方案

本章你已经看到了许多不同名称的语义类型。小步语义还叫结构化操作语义（structural

注 18：或者，在 Charles Babbage 设计的分析机这种机械计算机的场景下，是齿轮和纸遵守物理规律。

operational semantic) 和转换语义 (transition semantic)；大步语义更普遍的叫法是自然语义 (natural semantic) 或者关联语义 (relational semantic)；而指称语义还可以称为不动点语义 (fixed-point semantic) 或者数学语义 (mathematical semantic)。

还有其他类型的形式语义可用。其中一个就是公理化语义 (axiomatic semantic)，它通过在语句执行前后分别给出抽象机器状态的断言来描述一个语句的含义：如果一个断言 (前置条件) 在语句执行前初始是 true，那么随后的其他断言 (后置条件) 将是 true。公理化语义在验证程序的正确性方面很有用：随着语句合到一起组成更大的程序，它们对应的断言也能合到一起组成更大的断言，其目标就是表明对一个程序总体的断言与它的预期定义匹配。

虽然细节有所不同，但是公理化语义是描述 RubySpec project 最好的语义类型，RubySpec project (<http://www.rubyspec.org>) 是“Ruby 程序设计语言的可执行规范”，它使用 RSpec 类型的断言既描述 Ruby 的核心以及标准库，又描述 Ruby 内置语言结构的行为。例如，下面是 RubySpec 描述 Array#<< 方法的片段：

```
describe "Array#<<" do
  it "correctly resizes the Array" do
    a = []
    a.size.should == 0
    a << :foo
    a.size.should == 1
    a << :bar << :baz
    a.size.should == 3

    a = [1, 2, 3]
    a.shift
    a.shift
    a.shift
    a << :foo
    a.should == [:foo]
  end
end
```

2.6 实现语法解析器

本章，我们已经手工构建了 Simple 程序的抽象语法树——通过手写 Assign.new(:x, Add.new(Variable.new(:x), Number.new(1))) 这样的普通 Ruby 表达式，而不是先写 'x = x + 1' 这样原始的 Simple 源代码，然后使用一个语法解析器自动地把它转成语法树。

从头开始完整地实现一个 Simple 的语法解析器过于复杂，会分散我们讨论形式语义的注意力。尽管破解一个小程序语言很有趣，但是感谢解析工具和解析库的存在，在他人工作的基础上构造一个语法解析器并不是特别困难，因此下面将对其简单介绍一下。

Treetop (<http://treetop.rubyforge.org/>) 是 Ruby 可用的语法解析工具中最好的一个，它是一

一种特定领域的语言，能让语法解析器自动生成。一种语言的 Treetop 描述会写成解析表达式语法 (parsing expression grammar)，这是一个简单的类正则表达式 (regular-expression-like) 的规则集合，既易写又易理解。最好的是，这些规则能够使用方法定义作为注释，这样的话，就可以为语法解析过程中生成的 Ruby 对象定义行为。Treetop 既能定义语法结构，又能定义基于这些结构进行运算的 Ruby 代码集合，这使 Treetop 很适合描述一种语言的语法并赋予它可执行的语义。

为了让我们体验一下这是如何工作的，下面给出关于 Simple 的 Treetop 语法简装版，它只包含解析字符串 “while (x < 5) { x =x * 3 }” 所需要的规则：

```
grammar Simple
rule statement
  while / assign
end

rule while
  'while (' condition:expression ')' { ' body:statement ' } {
    def to_ast
      While.new(condition.to_ast, body.to_ast)
    end
  }
end

rule assign
  name:[a-z]+ '=' expression {
    def to_ast
      Assign.new(name.text_value.to_sym, expression.to_ast)
    end
  }
end

rule expression
  less_than
end

rule less_than
  left:multiply '<' right:less_than {
    def to_ast
      LessThan.new(left.to_ast, right.to_ast)
    end
  }
  /
  multiply
end

rule multiply
  left:term '*' right:multiply {
    def to_ast
      Multiply.new(left.to_ast, right.to_ast)
    end
  }
  /
  term
```

```

end

rule term
  number / variable
end

rule number
  [0-9]+ {
    def to_ast
      Number.new(text_value.to_i)
    end
  }
end
rule variable
  [a-z]+ {
    def to_ast
      Variable.new(text_value.to_sym)
    end
  }
end
end

```

这种语言看起来有点像 Ruby，但这种相似性只是表面的；语法是用特别的 Treetop 语言写出来的。关键字 `rule` 为分析一种特定种类的语法引入一个新的规则，并且每个规则里的表达式描述了它将要识别的字符串结构。规则可以递归地调用其他规则——例如 `while` 规则调用表达式（expression）规则和语句（statement）规则——而且分析从第一条规则开始，这是这种语法中的语句。

这些表达式语法规则彼此调用的顺序反应了 Simple 运算符的优先级。表达式语法调用 `less_than`，然后 `less_than` 立即调用 `multiply`，在 `less_than` 对优先级更低的运算符 `<` 进行匹配之前，`multiply` 能在字符串中匹配到 `*` 运算符。这确保表达式 '`1 * 2 < 3`' 被解析成 «`(1 * 2) < 3`» 而不是 «`1 * (2 < 3)`»。



为了让事情简单，这个语法没有试图限制可以在一种表达式中出现的另一种表达式种类，这意味着这个表达式将会接受一些明显错误的程序。

例如，对于二元表达式 `less_than` 和 `multiply`，我们设定了两个规则——但是分别设立两个规则的唯一原因是为了强调运算符的优先级，这样每一个规则只要求一个更高优先级的规则匹配其左侧运算对象，然后同样或者更高优先级的规则匹配其右侧运算对象。这将使像 '`1 < 2 < 3`' 这样的字符串能成功通过解析，即便 Simple 的语义无法赋予这个表达式结果一个含义。

这些问题中有一些可以通过对语法稍作调整得以解决，但是总会有其他一些不正确的情况语法解析器不能识别。这一问题我们将分成两个关注点，首先保持语法解析器尽可能的自由，其次将在第 9 章使用一个不同的技术来检测无效的程序。

语法中大多数的规则都使用外边带上括号的 Ruby 代码标注。在每一个括号里，代码都定义一个叫 `#to_ast` 的方法，在解析一个 Simple 程序的时候，它能用在由 Treetop 构建的对应语法对象上。

如果把这个语法保存到叫作 simple.treetop 的文件里，我们可以使用 Treetop 加载它来生成一个 SimpleParser 类。这个解析器可以把一个由 Simple 源代码组成的字符串转换成由 Treetop 的 SyntaxNode 对象构建出来的一个表示：

```
>> require 'treetop'
=> true
>> Treetop.load('simple')
=> SimpleParser
>> parse_tree = SimpleParser.new.parse('while (x < 5) { x = x * 3 }')
=> SyntaxNode+While1+While0 offset=0, "...5) { x = x * 3 }" (to_ast,condition,body):
    SyntaxNode offset=0, "while (" 
    SyntaxNode+LessThan1+LessThan0 offset=7, "x < 5" (to_ast,left,right):
        SyntaxNode+Variable0 offset=7, "x" (to_ast):
            SyntaxNode offset=7, "x"
            SyntaxNode offset=8, " < "
            SyntaxNode+Number0 offset=11, "5" (to_ast):
                SyntaxNode offset=11, "5"
                SyntaxNode offset=12, ")" { }
                SyntaxNode+Assign1+Assign0 offset=16, "x = x * 3" (to_ast,name,expression):
                    SyntaxNode offset=16, "x":
                        SyntaxNode offset=16, "x"
                        SyntaxNode offset=17, " = "
                        SyntaxNode+Multiply1+Multiply0 offset=20, "x * 3" (to_ast,left,right):
                            SyntaxNode+Variable0 offset=20, "x" (to_ast):
                                SyntaxNode offset=20, "x"
                                SyntaxNode offset=21, " * "
                                SyntaxNode+Number0 offset=24, "3" (to_ast):
                                    SyntaxNode offset=24, "3"
                                    SyntaxNode offset=25, " }"
```

这个 SyntaxNode 结构是一个具体语法树：它专门为了 Treetop 的处理而设计，并且含有关于这个具体语法树的节点是如何与生成它们的原始代码关联起来的大量信息。下面是 Treetop 文档 (http://treetop.rubyforge.org/using_in_ruby_html) 不得不说的一些话：

请不要尝试自己向下遍历语法树，并且不要把这棵树的结构作为你自己常用的数据结构。它包含的节点比你应用程序所需要的要多得多，甚至为输入的每个字符都分配一个还绰绰有余。

但是，你可以为根规则增加方法，根规则以一种合理的格式返回你需要的信息。每个规则可以调用它的子规则，并且从外面尝试遍历树时，利用这些遍历语法树的方法是一个非常好的选择。

这就是我们已经做到的。我们没有直接操纵这棵乱糟糟的树，而是使用语法中的标记在每个节点上定义一个 `#to_ast` 方法。如果在根节点上调用这个方法，它会根据 Simple 的语法

对象构建一棵抽象语法树。

```
>> statement = parse_tree.to_ast  
=> «while (x < 5) { x = x * 3 }»
```

这样我们已经自动地把源代码转换成了一棵抽象语法树，并且现在可以使用这棵树以通常的方式查看程序的含义了：

```
>> statement.evaluate({ x: Number.new(1) })  
=> { :x=>«9»}  
>> statement.to_ruby  
=> "-> e { while (-> e { (-> e { e[:x] }).call(e) < (-> e { 5 }).call(e) }).call(e);  
e = (-> e { e.merge({ :x => (-> e { (-> e { e[:x] }).call(e) * (-> e { 3 }).call(e)  
}).call(e) }) }.call(e); end; e }"
```



这个解析器和 Treetop 通常还有一个缺点，就是生成一个右结合的具体语法树。这意味着字符串 '1 * 2 * 3 * 4' 被解析时会被当成：'1 * (2 * (3 * 4))'：

```
>> expression = SimpleParser.new.parse('1 * 2 * 3 * 4', root: :expression).to_ast  
=> «1 * 2 * 3 * 4»  
>> expression.left  
=> «1»  
>> expression.right  
=> «2 * 3 * 4»
```

但是乘法通常是左结合的：写 '1 * 2 * 3 * 4' 的时候，我们实际的意思是 '((1 * 2) * 3) * 4'，这里数字是从表达式的左边（而非右边）开始分组结合的。对乘法来说这没什么关系——求值的时候两种方式会产生同样的结果——但对像减法和除法这样的运算就有问题了，因为对 «((1 - 2) - 3) - 4» 求值的结果与对 «1 - (2 - (3 - 4))» 求值的结果并不相同。

为了修正这个缺点，我们不得不让这些规则和 #to_ast 实现得更加复杂一些。参考 6.2.3 节，那里有构建左结合 AST 的 Treetop 语法。

能够像这样解析 Simple 程序很方便，但是因为困难的工作都由 Treetop 做了，所以我们对一个语法解析器实际如何工作并没有了解多少。在 4.3 节，你将会看到如何直接地实现一个解析器。