

Randomness and Simulation

Summary: Up to this point, all of the programs we have written are, in some sense, predictable. That is, the output depends only on the input. However, there are also programs that we want to be a bit less predictable, particularly simulations of events, games, and the like. In this reading, we consider Scheme's tools for supporting such *simulations*, particularly the `random` procedure.

Contents:

- Introduction
- The `random` Procedure
- Simulating a Die
- Other Uses of Random

Introduction

Many computing applications involve the simulation of games or events, with the hope of gaining insights and identifying underlying principles. In some cases, simulations can apply definite, well-known formulae. For example, in studying the effect of a pollution source in a lake or stream, one can keep track of pollutant concentrations in various places. Then, since the flow of water and the interactions of pollutants is reasonably well understood, one can follow the flow of the pollutants over a period time, according to known equations.

In other cases, specific outcomes involve some chance. For example, when a car begins a trip and encounters a traffic light, it may be a matter of chance as to whether the light is green or not. Similar uncertainties arise when considering specific organisms or when tabulating the outcomes involving flipping a coin, tossing a die, or dealing cards. In these cases, one may know about the probability of an event occurring (a head may occur about half the time), but the result of any one event depends on chance.

In studying events that involve some chance, one approach is to model the event or game, using a random number generator as the basis for decisions. If such models are run on computers many times, the results may give some statistical information about what outcomes are likely and how often each type of outcome might be expected to occur. This approach to problem solving is called the *Monte Carlo Method*.

The `random` Procedure

A random number generator for a typical computer language is a procedure which produces different values each time it is called. Such procedures simulate a random selection process. Scheme provides the procedure `random` for this purpose. This procedure returns integer values which depend on its parameter. In particular, `random` returns an integer value between 0 and one less than its parameter, inclusive.

```
> (random 10)
1
> (random 10)
9
> (random 10)
7
> (random 10)
0
> (random 10)
5
> (random 10)
1
> (random 10)
0
```

Simulating a Die

We can use `random` to write a program to simulate the rolling of a die, by generating integers from 1 to 6, to correspond to the faces on the die cube. The details of this simulation are shown in the following procedure:

```
;;; Procedure:
;;; roll-a-die
;;; Parameters:
;;; None
;;; Purpose:
;;; To simulate the rolling of one six-sided die.
;;; Produces:
;;; An integer between 1 and 6, inclusive.
;;; Preconditions:
;;; [None]
;;; Postconditions:
;;; Returns an integer between 1 and 6, inclusive.
;;; It should be difficult (or impossible) to predict which
;;; number is produced.
(define roll-a-die
  (lambda ()
    (+
     (random 6) ; a value in the range [0 .. 5]
     1))) ; now in the range [1 .. 6]
```

We can use that procedure to simulate the roll of two dice.

```
(define roll-dice
  (lambda ()
    (+ (roll-a-die) (roll-a-die))))
```

Other Uses of Random

We can use `random` to select between a variety of values, such as a collection of potential colors. The easiest way to do so is to use the result of `random` as the selector parameter to `list-ref`.

```

;;; Value:
;;; colors
;;; Type:
;;; List of strings
;;; Contains:
;;; A list of colors, suitable for processing by random-color
(define colors
  (list "red" "green" "yellow" "blue" "purple" "white" "black"))

;;; Procedure:
;;; random-color
;;; Parameters:
;;; [None]
;;; Purpose:
;;; Picks a "random" (unpredictable) color.
;;; Produces:
;;; color, a string
;;; Preconditions:
;;; Value colors has been defined as a list of strings that name colors.
;;; Postconditions:
;;; color names a color.
;;; It is difficult for someone to predict what color random-color
;;; will return.
(define random-color
  (lambda ()
    (list-ref colors
              (random (length colors))))))

```

We can use similar techniques to generate “different” sentences.

```

(define sentence
  (lambda ()
    (string-append
     (random-person) " "
     (random-transitive-verb) " "
     (random-object) ".")))

(define random-elt
  (lambda (lst)
    (list-ref lst (random (length lst)))))

(define random-person
  (lambda ()
    (random-elt (list "Henry" "Janet" "John" "Marge" "Sam"))))

(define random-transitive-verb
  (lambda ()
    (random-elt (list "saw" "watched" "threw" "ate" "borrowed"))))

(define random-object
  (lambda ()
    (string-append
     (random-elt (list "the" "a")) " "
     (random-elt (list "heavy" "blue" "green" "hot" "cold" "disgusting")) " "
     (random-elt (list "cup of coffee" "computer" "classroom"
                      "PBJ algorithm" "homework assignment")))))

```

Copyright © 2007 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.