Random

CS135 Lecture 21

L21.0 True randomness

Our model of computation is as powerful as any other

Over lectures L01-L04 we developed a model of computation with four elements:

1. Values and expressions
2. Functions
3. Conditional expressions
4. Recursion

Two computational systems have equal computational power if each system can **simulate** any computation that the other can perform. Using our rules for substitution, these elements give a model of computation that is as powerful as any other. Since we have arbitrarily large numbers, adding lists didn't change what we can compute. We'll see in L23 how we can simulate lists with large numbers.

We can’t write a function that generates a random number

Our model of computation can’t generate random numbers.

Given a Racket expression, our computation proceeds **deterministically**, following fixed substitution rules. At each step, the next rule to apply is unambiguously defined, until no more rules can be applied.

To generate a truly random number, a computer must have access to an external random (**non-deterministic**) source, like quantum noise, radioactive decay, or even the cosmic background radiation. These sources lie outside our deterministic model.

Randomness doesn’t increase computational power

Even if we add access to a source of true randomness, it doesn’t increase what we can compute. Conceptually, any outcome we might get could still be computed deterministically by exploring all possibilities (e.g. through backtracking).

For example, if we are simulating a game involving dice or cards, we can generate every possible roll or deal. This process is slow, but a computer with a source of true randomness is theoretically no more powerful than a computer without it.

What randomness can improve is **efficiency**: With randomness, we can often compute answers faster or more simply, even though the final result is still something we could compute without it.

Special devices can generate true random numbers

Many computers provide a source of randomness that is “close” to true randomness.

For high-security applications many computers now have dedicated hardware random number generators (HRNGs), sometimes based on quantum effects.

On a Mac and computers running Linux, there is a special device called “`/dev/random`”. Even without an HRNG, the device can pull from environmental noise like keyboard timings, mouse movements, and disk I/O to create a “pool” of random numbers.

There are also public sources of randomness such as the NIST Randomness Beacon (which you should not use for secret keys): https://beacon.nist.gov/home

None of these sources is fast and convenient enough for general-purpose gaming or simulation. True randomness is mostly required for security and cryptography.

https://www.icfo.eu/news/2521/nist-and-partners-use-quantum-mechanics-to-make-a-factory-for-random-numbers/

Some early computers tried to provide true randomness

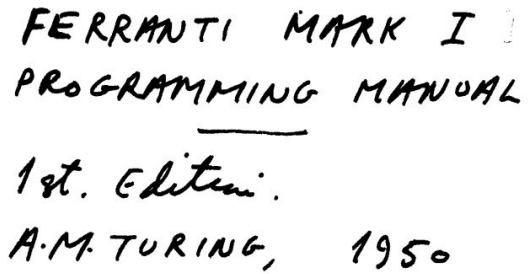
“In the Mark II machine the principle that the state of the machine at one completion signal determines the state at the next is abandoned if instructions with function symbol /W are used. The behaviour of the machine is then to be described as a ‘stochastic’ (i.e. chance-controlled) process and is suitable for calculations concerned with stochastic processes. The instruction /W actually puts random digits into the twenty least significant digits of the accumulator.

“The following problem is suitable for the use of this facility.

“A man in New York starts walking from a street intersection, and at each street intersection decides in which direction to walk by twice tossing a coin (each of the four directions is chosen equally often). It is required to find the probability that before walking twenty blocks he will have succeeded in returning to his starting point. For this purpose New York is to be assumed to be an infinite rectangular lattice of streets and avenues.”

from Alan Turing’s Manual for the Ferranti Mk. I

http://www.panix.com/~rst/turing.pdf



For most purposes “pseudorandomness” is sufficient

Pseudorandomness refers to the appearance of randomness produced by a deterministic algorithm.

For most games and simulations, it is sufficient to use deterministic functions to generate sequences of numbers that “appear” random.

In Racket, this pseudorandomness is provided by the **`random`** function.

In many games, the players themselves are a source of noise that is effectively random, since we can’t predict when a player will push a button or pull a trigger. Players can also take different actions in different circumstances.

True randomness is only a practical concern in high-security contexts.

L21.1 Pseudorandom numbers

Generating pseudorandom numbers

A “pseudorandom number generator” (PRNG) is a function that consumes a value for its current “state” (*S*) and produces a pseudorandom number R along with a new state (*S’*).

```
prng: State -> (list State Nat)
```

Starting with an initial state S_0 (called a “seed”) we can generate a sequence:

$\text{PRNG}(S_0) \Rightarrow (S_1, R_1)$
$\text{PRNG}(S_1) \Rightarrow (S_2, R_2)$
$\text{PRNG}(S_2) \Rightarrow (S_3, R_3)$
…

The sequence R_1, R_2, R_3, … is deterministic but has statistical properties similar to random numbers. Often, the pseudorandom number itself is also the state: $S_i = R_i$.

Middle-square method

An early pseudorandom number generator was proposed by the mathematician, physicist, engineer, and computer scientist John von Neumann in 1949.

Suppose we want to generate a random number between 0 and 9999 inclusive.

We square the previous number (the current state), then extract the “middle four” digits of the result to get the next number in the sequence.

```
;; middle-square-4: Nat -> Nat
(define (middle-square-4 n)
  (quotient (remainder (sqr n) 1000000) 100))
```

https://en.wikipedia.org/wiki/Middle-square_method

Middle-square method

```
;; middle-square-4: Nat -> Nat
(define (middle-square-4 n)
  (quotient (remainder (sqr n) 1000000) 100))
```

Assume `519` is our current number

`(sqr 519)` ⇒ `269361`

If we “pad” with zeros at the beginning `00269361`, the “middle four” is `2693`.

```
(middle-square-4 519) ⇒ 2693
(middle-square-4 2693) ⇒ 2522
(middle-square-4 2522) ⇒ 3604
```

Generating a list of random numbers using this method

```
;; middle-square-4: Nat -> Nat
(define (middle-square-4 n)
  (quotient (remainder (sqr n) 1000000) 100))

;; random-list: Nat Nat -> (listof Nat)
(define (random-list n seed)
  (reverse
   (foldr (lambda (x y) (cons (middle-square-4 (first y)) y))
          (list seed)
          (build-list n (lambda (n) n)))))

(random-list 8 519)
    ⇒ (list 519 2693 2522 3604 9888 7725 6756 6435 4092)
```

Seeding a random number generator

In a PRNG, the “seed” is a value that sets the initial state, which may come from a hardware source of randomness.

In the middle-square method, the seed is both the initial number in the sequence and the initial state for the PRNG.

The middle-square method can generate pseudorandom numbers with *n* digits (padded with zeros) where *n* is an even number so that extracting the middle *n* digits from a 2*n*-digit square is unambiguous.

However, the middle-square method is a very poor PRNG. For example, with a seed of 0, it will always generate 0 as the next “random number”. Many seeds quickly lead to short repeating cycles or even collapse to 0.

Linear congruential generator

One simple PRNG for straightforward applications like gaming is a “linear congruential generator”, which computes successive random numbers with the formula: $R_{i+1} = (aR_i + c) \bmod m$

The parameters *a*, *c*, and *m* need to be selected carefully, where a widely used selection gives the formula: $R_{i+1} = (1103515245 \cdot R_i + 12345) \bmod 2^{31}$

We present this formula solely for interest and insight. Just like you should (almost) never write your own sort function, you should (almost) never write your own PRNG. In particular, don’t use this formula for cryptography!

https://en.wikipedia.org/wiki/Linear_congruential_generator

L21.2 `random`

Racket has a built-in random number generator

The **`random`** function consumes a natural number **`n`** and produces a random number between `0` and **`(- n 1)`**, inclusive.

```
(random 10000) ⇒ 4075
```

It also consumes and produces state, but unfortunately this state is hidden away somewhere where we can't see it.

```
(random 10000) ⇒ 5371
```

This is called a "side effect", which is not something that normally happens in "Intermediate Student with Lambda". If we want to use **`random`**, we will just have to live with it, but remember that random is really consuming and producing state.

Seeding `random`

Sadly, there is also no way to seed the random number generator in “Intermediate Student with Lambda”.

```
(seed 3382891); ERROR 😢
```

In other programming languages (and in “Advanced Student”) it is possible to seed the random number generator and set the initial state, where the same seed will always set the initial state to the same value.

Nonetheless, with all its flaws, `random` will let us write simple games and perform simple simulations. It also makes testing harder, because we can’t repeat a sequence of random numbers without saving them somehow.

L21.3 Stochastic Simulation

Let’s try the problem posed by Alan Turing in 1950

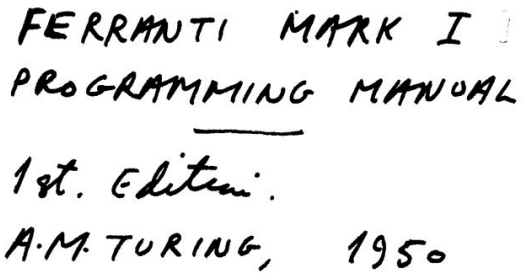
“In the Mark II machine the principle that the state of the machine at one completion signal determines the state at the next is abandoned if instructions with function symbol /W are used. The behaviour of the machine is then to be described as a ‘stochastic’ (i.e. chance-controlled) process and is suitable for calculations concerned with stochastic processes.The instruction /W actually puts random digits into the twenty least significant digits of the accumulator.

“The following problem is suitable for the use of this facility.

“A man in New York starts walking from a street intersection, and at each street intersection decides in which direction to walk by twice tossing a coin (each of the four directions is chosen equally often). It is required to find the probability that before walking twenty blocks he will have succeeded in returning to his starting point. For this purpose New York is to be assumed to be an infinite rectangular lattice of streets and avenues.”

from Alan Turing’s Manual for the Ferranti Mk. I

http://www.panix.com/~rst/turing.pdf



Let’s try the problem posed by Alan Turing in 1950

“A man in New York starts walking from a street intersection, and at each street intersection decides in which direction to walk by twice tossing a coin (each of the four directions is chosen equally often). It is required to find the probability that before walking twenty blocks he will have succeeded in returning to his starting point. For this purpose New York is to be assumed to be an infinite rectangular lattice of streets and avenues.”

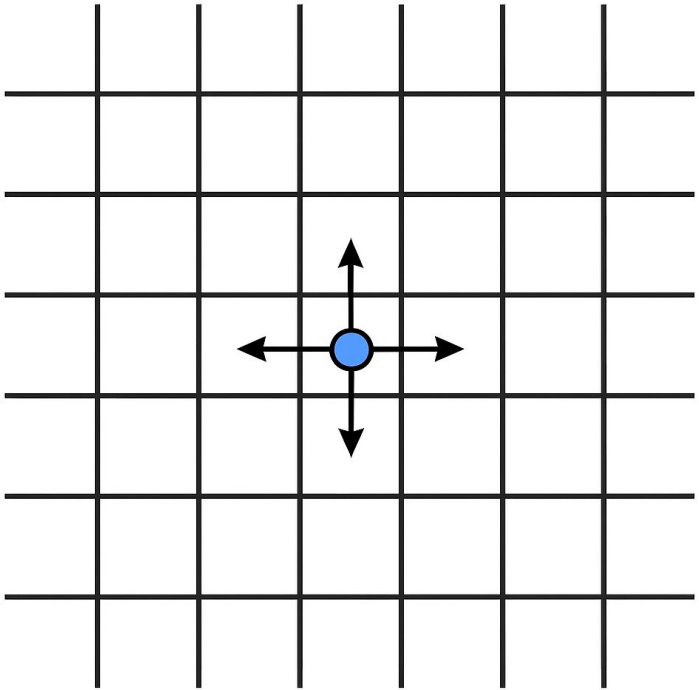
The wording reflects the time period. Today we would say “person” instead of “man”, and we wouldn’t assume that everyone knows that New York is (more or less) laid out in a grid pattern.

Let’s abstract the problem

Starting at the origin (0,0) our person will move randomly either North (0, +1), South (0, -1), East (+1, 0) or West (-1, 0).

They can make up to *n* moves. For example, *n* = 20.

We want to write a function that consumes a natural number `n`, and produces `true` or `false`, indicating if they ever returned to the origin.



A random move

```
;; Randomly move one block from a current position
;; random-move: (list Int Int) -> (list Int Int)
(define (random-move location)
  (local
    [(define move (random 4))
     (define x (first location))
     (define y (second location))]
    (cond [(zero? move) (list x (add1 y))] ; North
          [(= 1 move) (list x (sub1 y))]   ; South
          [(= 2 move) (list (add1 x) y)]   ; East
          [else (list (sub1 x) y)])))      ; West
```

Do we return to the origin in up to n moves?

```
;; Does up to n random moves return us to the origin?
;; random-moves: Nat (list Int Int) -> Bool
(define (random-moves n current)
  (cond [(zero? n) false]
        [else
         (local
           [(define new (random-move current))]
           (cond [(and (zero? (first new))
                       (zero? (second new))) true]
                 [else (random-moves (sub1 n) new)]))]))
```

10,000 simulations

```
;; Simulate m random walks of up to n moves,
;; producing the number of times we return to the origin
;; simulate: Nat Nat -> Nat
(define (simulate m n)
  (foldr (lambda (x y)
           (+ y (cond [(random-moves n (list 0 0)) 1]
                      [else 0])))
         0
         (build-list m (lambda (n) n))))

(simulate 10000 20)
```

L21 Summary

L21: You should know

- The difference between true randomness and pseudorandom numbers.
- How to use `random` to write games and simulations.
- Not to write your own PRNG or cryptographic functions.

If you are interested in this and related topics, you might want to take

CS 459: Privacy, Crypto, Network, Data Security, or

CO 487: Applied Cryptography

L21: Allowed constructs

Newly allowed constructs:

```
random
```

Previously allowed constructs:

```
( ) [ ] + - * / = < > <= >= ; '
abs acos add1 and append asin atan boolean? build-list char?
char=? char<? check-expect check-within cond cons cons? cos
define e else empty empty? even? exp expt false filter first
foldl foldr inexact? integer? lambda length list list? local
log map max min not number? odd? or pi quotient rational?
remainder rest reverse second sin sort sqr sqrt string?
string=? string<? string->list list->string string-append
sub1 symbol? symbol=? tan third true zero?
listof Any anyof Atom Bool Char Int Nat Num Rat Str Sym
```