Random.mesa

Produces a random sequence of INTs.
Last edited by:
MBrown on August 27, 1982 10:59 am

```
Random: CEDAR DEFINITIONS = BEGIN
```

Init: PROC [range: INT _ 0, seed: INT _ 0] RETURNS [trueSeed: INT];

The parameters range and seed determine the sequence generated by procedure Next (also Choose) below. If range<=0, Next will produce results in [0..LAST[INT]]; otherwise, Next will produce results in [0..range). If seed=0, a default seed value is used to determine the starting point of the sequence; if seed>0, seed is scaled if necessary and then used; if seed<0, a seed value is derived from the system clock. In any case, the seed value actually used (after scaling) is the INT value returned by Init.

Avoid small values of the range parameter; the intent of range is only to allow the same sequence to be produced by different machines.

Next: PROC [] RETURNS [INT];

A call to Next returns the next term in the sequence determined by the most recent call to Init.

Choose: PROC [min, max: INT] RETURNS [INT --[min..max]--];
! Error [badInterval] ([min..max] is an illegal interval).

Chooses a point in the interval [min..max] at random. (The choice is quite random, even if this interval is large.) If the interval is empty (min > max), or if the interval length (max-min+1) exceeds the range of numbers the generator was initialized to produce, raises ERROR Error[badInterval].

```
Error: ERROR [ec: ErrorType];
ErrorType: TYPE = { badInterval };

END.
```

Random number generators

Basic information

A generator is created by module instantiation, that is, it holds the state of its current pseudo-random sequence in a global frame. A generator is also a module monitor, so that two unsynchronized processes may each request the next term of the current sequence with no ill effect. (A test program that takes advantage of this feature is not likely to give repeatable results, however.)

STARTing a generator causes it to initialize itself by calling Init, with default parameters. So a simple application needing a (repeatable) random sequence need not contain an explicit Init call.

Three variants of this package are available, generating nonnegative INTs (Random), nonnegative INTEGERs (RandomInt), and CARDINALs (RandomCard). The only functional difference between the three interfaces Random, RandomCard, and RandomInt is that RandomCard is only capable of generating sequences in [0..LAST[CARDINAL]]; RandomCard.Init takes no range parameter.

Because Init is a procedure having two parameters of the same type, it is best called by using Mesa's keyword parameter syntax; Choose should also be.

Sample program

The following is the skeleton of a program that uses the Random package. The program fills an array called "rolls" with random numbers that might represent die trials:

```
DIRECTORY
  Random USING [Init, Choose],
  ...
Sample: PROGRAM IMPORTS Random, ...
  ...
  rolls: ARRAY [0..100) OF INT [1 .. 6];
  ...
  [] _ Random.Init[]; -- use default seed and full range
  FOR i: INT IN [0..100) DO
    Rolls[i] _ Random.Choose[min: 1, max: 6]
    ENDLOOP ;
  ...
 END .
```

Technical details

These generators are based on an AlgolW program distributed by Donald Knuth to his CS144b class in 1975. They use the additive random number generation algorithm that is recommended in the second edition of Seminumerical Algorithms by Knuth, and that was the subject of a Ph.D. thesis by John Reiser of Stanford ("The analysis of additive random number generators", STAN-CS-77-601, March 1977.) The additive generator has several advantages over a standard linear congruential generator (lcg):

The sequences that it generates are more "random" in several ways. With an lcg generating a sequence a(n), the related sequence a(n) mod 2 has period 2, a(n) mod 4 has period 4, and so on; thus care must be taken not to derive any results primarily from the least-significant bits of a(n). An additive generator produces numbers whose bits are more uniformly random. An lcg for a short wordlength has a short period (for example, a sequence of 16 bit cardinals must cycle after 64k terms); an additive generator may have a very long period even for short wordlengths. Many lcgs (including IBM's notorious RANDU) generate sequences that are very poorly distributed in two and three dimensions (you might see this by plotting consecutive pairs or triples of terms in euclidean space and noting the patterns); additive generators do not have this problem.

The generator relatively easy to transport from machine to machine, and can be made to generate the same sequences on each machine. This can be difficult with lcgs because the sequences they generate usually depend on the way a machine handles overflows (in particular, on the wordlength.) An additive generator can produce a random sequence without ever causing an overflow.

The generator is relatively fast because it does not use multiplication, which is a time-consuming operation on our present machines.

Change Log

Created by MBrown on July 2, 1979

Changed by MBrown on May 2, 1980 11:18 PM

Renamed module to conform to Cedar standards. Added defaulting of parameters to InitRandom, and named its result trueSeed.

Changed by MBrown on September 25, 1980 9:26 AM

Added Choose.

Changed by MBrown on March 9, 1982 10:47 am

Added Error, ErrorType (used to raise unnamed ERROR *).*

Changed by MBrown on June 8, 1982 11:04 am

Changed module name from RandomLongInt, procs Next from Random, Init from InitRandom, made module CEDAR

Changed by MBrown on August 27, 1982 11:10 am

Merged in documentation.