Roll: A Language for Specifying Die-Rolls

Torben Mogensen

February 14, 2006

Abstract

Role-playing games (RPG's) use a variety of methods for rolling dice to add randomness to the game. In the simplest form, a small number of identical dice are rolled and added, but more advanced forms involve cumulative rerolling of 6's, doubling the value of doubles, removing the lowest or highest result or counting the number of dice that are below a treshold, and a host of other weird and wonderful modifications.

While die-roll programs and net-based die-roll servers exist, they can usually only handle the simplest form of die-rolls. This paper describes **Roll**, a simple functional language for defining how dice are rolled. Such definitions are then used to emulate die-rolls, gather statistics or even make exact probability calculations.

1 Overview of Roll

This first section describes the language **Roll**. If you want to run the examples while reading the text, you should first install the program as described in section 6.2 and then read the instructions for running the program in section 6.

Roll assumes all die-rolls result in either a single integer value or an unordered collection of such values. A single value is equivalent to a collection of one value, which we will call a *singleton collection*.

A die-roll definition is an expression that use numbers and operators to create simple die-rolls and combine these into more complex die-rolls or modify die-rolls according to certain conditions.

2 Simple die-rolling

Following the usual RPG convention, a single die is specified by a "`d`" or "`D`" followed by a number indicating the number of faces on the die. So, for example, a six-sided die is specified as "`d6`" or "`D6`". There is no difference between "`d`" and "`D`", both are provided only to allow different textual styles. An *n*-sided die is assumed to yield the values from 1 to *n* with equal probability. The number after a "`d`" can be any expression with an integer value (greater than 0), so, for example, "`d d6`" rolls a d6 to find which type of dice (from `d1` to `d6`) is rolled.

Rolling a certain number of dice is specified by prefixing the die-specifier by a number using the operator "`#`", so 5 six-sided dice are specified as, e.g., "`5#d6`". This deviates slightly from the RPG convention that doesn't use any operator between the number and the die. Furthermore, "`5#d6`" produces a *collection* of 5 dice instead of adding the values of 5 dice.

To get the sum of 5 six-sided dice, you write "`sum5#d6`". Note that spaces are allowed (but not required) between the numbers and operators, so we could equally well have written "`sum 5 # d 6`".

You can add, subtract, multiply and divide dice or numbers using the usual arithmetic operators (`+`, `-`, `*` and `/`). Division is integer division, so, e.g., `11/3` yields `3`. Note that `3*d6` means a single die multiplied by 3 and *not* the result of adding 3 dice. "`-`" also works as a unary operator, so you can, for example, write "`-d7`". Arithmetic only works on singleton collections.

3 Operations on collections

The collections used in Roll don't consider ordering of the values – so a collection of the values 1 and 2 is the same as a collection of the values 2 and 1. Furthermore, a single value is considered as a singleton collection and *vice versa*.

The operator "`#`" can be prefixed by any expression that evaluates to a singleton collection (with non-negative value) and followed by any die-roll expression, so you can, for example, write "`d4#d6`" to specify a collection of 1 to 4 six-sided dice or "`10#(sum 5#d6)`" to specify a collection of ten values, each obtained by adding 5 six-sided dice. If you write "`3#4#d6`", it is read as "`3#(4#d6)`" and produces a collection with a total of 12 six-sided dice, by combining 3 size-4 collections. In general, if what follows the "`#`" operator is something that produces a collection, these are combined to a single collection. In other words, you can't have a collection of collections. These will always be collapsed to a single collection.

Combination of two collectionss is done by the operator "`@`", for example in "`3#d6 @ 3#d8`" that combines a collection of 3 six-sided dice with a collection of 3 eight-sided dice. Since numbers are treated as singleton collections, they can be added to a collection in the same way, e.g., "`d6 @ 3#d8`" or "`3#d6 @ d8`".

We have already seen "`sum`", which adds the elements of a list. The operator "`count`" counts the number of dice in a collection. This, normally, isn't useful unless combined with a *filter* that removes elements that don't obey a specified condition. For example, the expression "`count =6 10#d6`" rolls 10 six-sided dice and uses the filter "`=6`" to remove the dice that aren't equal to 6 and then count these. You can also filter by less-than ("`<`"), greater-than ("`>`"), less-than-or-equal ("`<=`") or greater-than-or-equal ("`>=`"), so, for example, "`count <4 10#d6`" counts the number of dice that are less than 4. The number after the comparison operator can be any integer-valued expression (singleton collection), so you can write, e.g., "`count <d6 10#d6`" that rolls a die and counts how many of the 10 next dice that are less than this.

You can also take the *n* least or the *n* largest values from a collection using the operators "`least`" and "`largest`". For example, "`largest 1 least 2 3#d6`" finds the largest of the two smallest of three dice, *i.e.*, the middle (or median) value. Integer valued expressions that evaluate to a non-negative value can be used as first argument to `least` or `largest`. If there are

less than *n* values in the collection, all elements are returned.

4 Value definitions, conditionals, *etc.*

If you, for example, write "`d6*d6`" you get the product of two independently rolled dice. If you want to square the value of a single die, you have to store the roll of one die in a variable and use the variable twice. This can be done by a local definition of the form "`let x = d6 in x*x`" which defines `x` to be the value of a single die and then multiplies that value by itself. Any expression can be used after the equality sign and after "`in`". The name of the bound variable must consist of letters only and may not be identical to an operator name.

You can also make a conditional choice between two rolls using an `if-then-else` construction. The form of this is `if` e_1 `then` e_2 `else` e_3. If e_1 is a non-empty collection, e_2 is evaluated, otherwise (*i.e.*, if e_1 is empty), e_3 is evaluated. This is most often used in combination with a filter, *e.g.*, "`if =x y then x+y else largest 1 (x@y)`" to take the largest of two dice but let doubles count double. Strictly speaking, the `if-then-else` construction is superfluous, as you can replace it with judicious uses of filters, `count` and "#", but it makes some definitions easier to write.

Another construction that is sometimes useful is the "`foreach`" construction. It applies the same method to all values in a collection and combines the result to a new collection. For example, you can add 1 to all members of a collection `c` by writing "`foreach x in c do x+1`".

A special construction, the "dotdot" construction, produces a range of integer values. "`1..6`", for example, is the collection of all integers between 1 and 6 (inclusive). You can use any integer-producing expressions instead of the constants in the above, for example write `d4..d10` to get the range of values between the result of rolling a d4 and the result of rolling a d10. If the first value is larger than the second (*e.g.*, "`7..3`"), the empty collection is produced.

Used in combination with `foreach` you can, for example, specify that you roll 7 d10's and take the highest sum of identical dice in the result:

```
let c = 7#d10 in largest 1 (foreach x in 1..10 do sum (=x c))
```

If, for example, the 7 d10 yield the collection "1 3 3 3 5 7 7", the `foreach` construction gives "1 0 9 0 5 0 14 0 0 0", the largest of which is 14.

5 Repeated die-rolls

Sometimes, a die-roll involves repeating rolls until a certain condition occurs. A game may, for example, define that you roll and add 3 six-sided dice, but for all sixes rolled, one more die is added. If any of these dice are also sixes, yet more dice are added, and so on. Thus, there is no *a priori* bound on the number of dice rolled.

To allow such rolls, **Roll** allows conditionally repeated die-rolls. We show this first by an example. The above-described `3d6` with rerolls on sixes can be described by

```
sum (let x=3#d6 in repeat ((count =6 x)#d6))
```

We first roll 3 dice, binding the result to `x`. We then repeat an expression that rolls a number of dice equal to the number of sixes in `x`, producing in each step a new `x`, until that new `x` becomes empty. Finally, we combine all the `x`'s and add up the dice.

The semantics of the `repeat` construct is that the variable after the `let` keyword is bound to the value v_0 of the expression after the = sign. Then the expression after the keyword `repeat` is evaluated to yield a new value v_1. This is then bound to the variable and the expression is re-evaluated to yield a new value v_2, and so on, until the value of the expression becomes the empty collection. At this point, all the values are combined to a single collection $(v_0@v_1@\ldots@v_n)$. If the value of the initial expression is the empty collection, the repetition stops immediately with the empty collection as a result.

In the example above, let us assume `3#d6` evaluates to the collection `2 6 6`. This is bound to `x` and the expression in the body of the repeat construct is evaluated to a collection of 2 `#d6`, let us say `4 6`. `x` is now bould to this and the body is reevaluated to a single die, say `3`. `x` is then bound to this and, finally, the expression yields the empty collection. We now combine `2 6 6 @ 4 6 @ 3` to `2 3 4 6 6 6`.

Another example using repetition is rolling `d6`'s one at a time until a 6 is rolled and then counting the number of rolls. This can be written as

```
count (let x=d6 in repeat ((count <6 x)#d6))
```

The repeated expression uses a filter to ensure that a new die is rolled only if the previous die is less than 6.

6 A prototype implementation

The language has been implemented in Standard ML, using the Moscow ML implementaion of that language. The program, called "`roll`" is run from the command-line as described below. To run the program, you need to install Moscow ML, see section 6.2 below.

You run the program by writing "`roll`" followed by a number of arguments. Each argument can be one of

- A filename. The file should contain a definition of a die roll. If no file name is specified, the definition is read from the terminal. Input is terminated by the endo-of-file character (control-D in Unix/Linux and control-Z in DOS/Windows). If several filenames are specified, the last on the line is used.

- A number. If positive, this specifies the number of times dice are rolled. Each roll is shown on a separate line. If the number is 0 or negative, a probability distribution is calculated and printed out. If all rolls generate singleton or empty collections, an average and spread is also printed out. If the specified number is 0, the default number of iterations for the `repeat` construct is used. If the number is $-n$, at most n iterations are made. If several numbers are specified, the last on the line is used. If none are specified, 1 is used as the default value.

- A definition of the form `name=number`. This defines the name to have the specified number as value when used in the definition. This is useful if you want to do calculations of a die-roll method that uses a variable number of dice or a variable threshold, *etc*.. Any number of such definitions can be entered.

Examples

The simplest instance of use is to write just "`roll`" on the command line, then write a short definition (e.g. "`d6`") on the next line and terminate this by the end-of-file character. This will produce a single roll of a d6.

If you have a file `test.d` containing a definition like the following:

```
count >=T N#d10
```

You can call **Roll** like this:

```
roll test.d 0 N=7 T=5
```

and get the following output:

```
  Value    % =               % <
     0 :   0.16384           0.0
     1 :   1.72032           0.16384
     2 :   7.74144           1.88416
     3 :  19.3536            9.6256
     4 :  29.0304           28.9792
     5 :  26.12736          58.0096
     6 :  13.06368          84.13696
     7 :   2.79936          97.20064

Average = 4.2    Spread = 1.29614813968
```

The `-p` option switches to printing probabilities as numbers between 0 and 1, so

```
roll -p test.d 0 N=7 T=5
```

produces

```
  Value  Probability for =  Probability for <
     0 :   0.0016384         0.0
     1 :   0.0172032         0.0016384
     2 :   0.0774144         0.0188416
     3 :   0.193536          0.096256
     4 :   0.290304          0.289792
     5 :   0.2612736         0.580096
```

```
     6 :   0.1306368         0.8413696
     7 :   0.0279936         0.9720064

Average = 4.2    Spread = 1.29614813968
```

The `-g`*x* option makes the program print a graph of the probability distribution (using ASCII graphics). For example,

```
roll -g1.3 test.d 0 N=7 T=5
```

produces

```
  Value    1.3 bars per %
     0 :
     1 :  ||
     2 :  ||||||||||
     3 :  ||||||||||||||||||||||||
     4 :  ||||||||||||||||||||||||||||||||||||
     5 :  ||||||||||||||||||||||||||||||
     6 :  ||||||||||||||||
     7 :  ||||

Average = 4.2    Spread = 1.29614813968
```

Notes

- The program writes the negative sign for numbers as "`~`" so, for example, minus three is written as "`~3`". This behaviour is inherited from Standard ML. You can use both "`~`" and "`-`" to specify the sign of numbers on the command line, but in Unix/Linux you should careful about using "`~`", as it has a special meaning there.
- The program assumes that any command-line argument that starts with a digit is a number, so you can't use file names like "`7thSea.d`" (which will be read as the number 7).
- You shouldn't use dice with more than around nine hundred thousand sides, since this may cause the random number generator to loop. I can't see why you would want to use such large dice anyway.

6.1 Hints on reducing running times

If a large number of dice are used, the program can take very long to compute the probability distribution. This is because it, essentially, enumerates all possible rolls and then counts how many there are of each outcome. Newer versions of the program enumerate in an intelligent fashion (by working on unnormalized probability distributions) so, for example, the distribution

for `sum` *N*`#d10` takes quadratic time (in *N*) to compute instead of exponential, as was the case for the first versions of the program which used straight-forward enumeration. Hence, most of the hints for reducing running time that were shown in earlier versions of this manual are now redundant.

One thing to note is that binding a value in a `let` expression will force normalization of the distribution for that value. Sometimes, that may make the program take much longer, *e.g.*, in `let x= 10#d8 in sum x` compared to `sum 10#d8`, where the former takes 40 times longer. At other times, normalization can make things run faster. This is the case when there are many ways to obtain the same collection *and* these are fairly short.

It may, also, still be worthwhile to throw away unneeded information as early as possible. And if your definition uses `repeat`, you can reduce the number of iterations to less than the default. This will reduce precision, but may be necessary to obtain anything at all.

An example that shows several of these aspects is the dice system from L5R and 7th Sea. Here, you roll N openended d10's and add the M largest of these. M and N can vary. You can express this as

```
sum (largest M N#(sum let x=d10 in repeat (x/10)#d10))
```

and run it by, for example,

```
roll 0 L5R.d M=2 N=3
```

This takes 28 seconds on my computer, and if `M` and `N` are increased, it will take *much* longer. So, to get results for higher values, you can limit rerolls to, *e.g.*, 4:

```
roll -4 L5R.d M=3 N=4
```

This still takes a long time (89 seconds), but can be reduced somewhat by normalizing before picking the `M` largest dice:

```
let x = N#(sum let x=d10 in repeat (x/10)#d10)
in sum largest M x
```

This reduces the running time to 46 seconds. Note that normalization for this particular method only pays if `N` = `M`+1, and may cause much higher running times if `N` is larger than that. For example, with `M`=1 and `N`=4, the original version takes a fraction of a second, but the modified version takes 32 seconds. Hence, a conditional definition is best:

```
if =N (M+1) then
  let x = N#(sum let x=d10 in repeat (x/10)#d10)
  in sum largest M x
else
  sum (largest M N#(sum let x=d10 in repeat (x/10)#d10))
```

You should, however, only worry about optimizing if it becomes necessary. Otherwise, keep your definitions simple and readable and don't worry about speed unless you need to.

6.2 Installation

To install or use the program, you must have Moscow ML installed on your computer. You can get Moscow ML, including manuals and installation instructions from
`http://www.dina.dk/~sestoft/mosml.html`.

Then get `http://www.diku.dk/~torbenm/Dice.zip` and unpack it in an empty folder (if your are reading this, you have probably done so already). Use WinZip, Info-Zip or some other unzipper that supports long filenames.

To get a Windows/DOS executable called `roll.exe`, run `compile.bat`. This can be done by clicking on its icon in the directory where you unpacked the Zip file. Note that `roll.exe` must be called from a DOS command line.

For Linux or other Unix platforms, run `compile.csh` as a shell command in the same directory as you extracted the sources, e.g., by

```
unzip Dice.zip
bash compile.csh
```

This will compile the sources and produce an executable called `roll`. A few warnings are generated by the command, but these are harmless.

For other platforms, see the Moscow ML Owners Manual (from the above link) for how to compile the sources.

Changes from previous releases (in reverse cronological order)

(February 2005) Added option for probability graphs with the -g*x* option.

(July 2002) Added percentage based printout, which is now the default. The old behavior can be obtained with the `-p` option.

(June 2002) Fixed a bug in the `if` expression, which calculated both branches even if only one was possible.

(June 2002) Executables are no longer provided, as they caused more problems than expected. Use the compile scripts to generate an executable. A few minor bugs have been fixed. The random number generator has been improved.

(May 2002) The command-line parameters have been changed to allow definitions and input from the terminal. In the printout, the abbreviation “Std. dev.” (standard deviation) has been replaced with the equivalent and shorter term “Spread”.

(April 2002) The method for enumerating rolls for calculating distributions has chanced. This means that some examples run *a lot* faster than they used to.

(April 2002) The iteration count for `repeat` was done incorrectly in the previous release. This means that you will have to specify two more iterations when using the new version to get the same result as with the old version. The default number of iterations has been changed to 12 (where it used to be 10) to give the same result in the default case.

(Earlier) Some keywords are changed to more intuitive names: `size` is now `count` and `iterate` is now `repeat`.

(Earlier) The program `roll` has a different way of specifying roll counts. Furthermore, it no longer prints statistics if you specify a positive count, but just prints the specified number of rolls. See above for details.

7 Conclusion

Roll is a very flexible and powerful tool for defining die-rolls and should be able to specify almost any die-roll method imaginable, even methods involving unbounded rerolling. Simple methods are very easy to describe and even moderately complex methods can be described in one line of text.

Calculation of propability distributions can be slow, especialy for repeated rolls. This is because a general method applicable to all cases is used, even in cases where faster methods are known.

Appendix: Operator precedences and various notes

Precedences from highest to lowest:

```
D    d
#
sum  count   least  largest
*    /
+    -
=    <       >      <=       >=
@
..
in   repeat  do
```

Parentheses can be used to overide the precedences.

Note that spaces are optional in most places, but space is required between two adjacent numbers and between two adjacent alphabetical operators or variables.

Comments can be added to die-roll definitions. Comments begin with a backslash (“\”) and extend until the end of the line.