Dicelab Manual

Contents

1 Introduction to Dicelab

Dicelab is a computer program to calculate the probability distributions of arbitrary dice rolls. This can be used to analyze games and design games, and perhaps it even comes in in some more serious scenarios (as if there was anything more serious than gaming!). To allow dicelab to analyze any dice rolling scheme, and not just a few predefined ones, it uses a specific language called "bones". The bones language was put together by Joel Uckelman, and both his and my work are inspired by and based on the program "roll" by Torben Mogensen. So assuming you are in the middle of an intense table-top roleplaying session, and the future of the whole federation of stars depends on you making the right decision, you might want to know what the odds are. If we assume that you have to roll 3 dice with 6 sides each, remove one of them (the lowest), sum up the remaining ones and compare with a threshold that depends on the situation and your skill, then you could check your odds to beat a threshold of 10 with the following input to dicelab:

```
count
  >10
    sum
      keep high 2
        3#d6;
```

which would yield the following output:

```
  0      0.800926
  1      0.199074
```

Almost 20%, the federation has seen worse times! Or imagine you got tangled up in a long standoff in risk, armies have built up on both sides and the decisive final battle is getting closer. An opportunity to disrupt your enemies actions has arisen, but what exactly are the odds for each round in the battle? In risk the attacker rolls 3 dice, and the defender rolls 2 (assuming has has two or more armies left). The highest attacker die is then matched with the highest defender die, the second highest attacker die is matched with the lowest defender die. For both matches the highest wins, ties go to the defender. So each round of battle, a total of two armies are destroyed. The outcome probabilities for the number of wins of the attacker could be calculated using dicelab with the following input:

```
let a = 3#d6 in
  let b = 2#d6 in
    count( (<(high 1 a) high 1 b),
           (<(high 1 low 2 a) low 1 b))
```

Which would result in the following output:

```
  0      0.292567
  1      0.335777
  2      0.371656
```

You can clearly see where the name of the game comes from! As you can hopefully see, dicelab is quite powerfull and capable to precisely calculate the probability distributions in complicated dice rolls. You will also notice that the input language is quite complicated and that the results are sometimes hard to interpret correctly. This manual hopefully helps to understand dicelab and the bones language.

2 Installation and Usage

Depending on your operating system, there are a couple of different ways to get dicelab up and running. Unless stated otherwise, the files are available from the follwing URL, where you can also check the availability of new versions: http://www.semistable.com/dicelab/.

2.1 Building from Source

Should you have the necessary tools and a liking for this, want to modify dicelab or if you cannot find an easier way to get dicelab onto your machines, you can always compile it from source. You would need to get the dicelab package from the URL mentioned above and unpack it on a reasonable modern UNIX (e.g. Linux) system with the necessary build dependencies insatlled. Dicelab uses autotools, so you can simply run `./configure` and `make` to build dicelab. You will need GNU make, gcc, flex, bison and treecc in reasonably current versions to build dicelab. Should you have problems building this manual, then that is probably because it uses some non-standard tools to create teh graphics. Just use the one from the web page for the time being.

2.2 Packages

The easiest way to install dicelab on a Linux system or something similar is to use pre-built packages provided by your OS community. In debian you could for example install dicelab directly with synaptic or your package manager of choice. Should packages for your OS not be available, pester the relevant people or the author of dicelab! In the time they try to fix it, you will have no choice but to build it from source.

2.3 Windows

Dicelab is originally a UNIX program, and the whole mode of operation will feel a bit awkward under Windows. Due to the surprising large popularity of this legacy OS, pre-built windows binaries are available in the same location as the regular sources. Please note that dicelab is not a windows GUI program that has a graphical user interface, to use it you will have to let go of your mouse and work in a command prompt!

2.4 Running

Dicelab is a commandline program that is started, reads input and produces output. While this might look awkward to some, especially those that come from a culture of graphical

user interfaces, it is in fact very powerfull and also way cooler! So you would have to start dicelab under a command prompt in windows or a shell under UNIX. It does accept a few commandline options and switches, the full syntax (which you can also get by entering *dicelab -?*) is as follows:

```
Usage: dicelab [options] [-f <file>]
Options:
  --help -h -?      print this text
  --version -v      print the program version
  --print-tree -p   print the parse tree (for debugging)
  --eval -e         evaluate the statistical distribution by
                    re-rolling
  --calc -c         calculate the distribution
  --count -n        specify the number of rolls used with --eval
                    default is 10000
  --roll -r         roll the dice as specified, will also be used
                    if no other action is requested
  --threshold -t    cutoff threshold, results with a probability less
                    than this value will be discarded by some operations
                    when using --calc
File:
  --file -f         read the dice rolling specs from the file
                    specified, use stdin if not supplied
```

The typical use case is to start dicelab with the "-c" switch, but without a file. It will then read form the standard input allowing you to use it in some kind of interactive mode. As a quick check you should try this: run `dicelab -c`, then enter "d6;" and press the enter key. You should see a list of results from 1 to 6 all having the same probability (0.166667). You could then enter another request (e.g. "d5;") and finally stop the program by pressing `Ctrl-C`. In some cases it is very time-consuming to do the maths for a dice rolling scheme, you can then resort to just re-rolling many times and summing up the results by using the "-e" instead of the "-c" switch. If you do that with the examples above, you will see roughly the same results, but with less accuracy. This can be affected by specifying a larger number to roll with "-n". The calculation mode is normally perfectly accurate, but in some cases even this mode has to make comprimises, most notably with "infinite" throws, e.g. when you reroll and add every time you roll a 6. In this case dicelab will stop once the probability has reached a given value ($10^{-}7$ by default), you can modify this with the "-t" parameter, but you should hardly ever need to use this. Instead of calculating the distribution of a dice roll, you cann also just do a single roll, which is done bu using "-r" or no mode at all.

Instead of entering the input by hand, you can also write to a file and instruct dicelab to read that file with the "-f" parameter. This is especially handy on UNIX, where you can use a she-bang line to turn a dicelab input into an executable. you could for example take the following file, make it executable with `chmod +x testfile` and execute it directly.

```
#!dicelab --calc --file
sum 3#d6;
```

3 The Dicelab Language

The most difficult part of using dicelab is to understanding the language and being able to express your problems in it. Unfortunately this is very hard to avoid, so you will have to deal with it. This part of the manual should help you get started and act as a reference. At the end of the day however, expressing something complicated in the bones language will always remain a bit of a puzzle and should be seen in a similar way: an opportunity to grind your brains against something instead of a hassle!

To save space and make reading easier, all examples will from now on be typeset next to the relevant output they produce when using them with `dicelab -c`. Where necessary some part of the output is ommited and the relevant lines are replaced with a single line containing `...`.

You are encouraged to try out the examples as we go along, to get a "feel" for the operation of the program. It is also certainly a a good idea to modify them slightly to find out what happens!

3.1 Introduction

Let us look at the most simple example, the distribution of a single die:

```
d6;                                    1     0.166667
                                       2     0.166667
                                       3     0.166667
                                       4     0.166667
                                       5     0.166667
                                       6     0.166667
```

There is already a bit more to it than meets the eye:

- The input is broken into *statements*, each statement will be processed independently. Statements are separated by semicolons and are only processed when the respective semicolon is seen (of course! otherwise it would be impossible to process a statement like `d6 + 2;`).
- The input is read linewise. This is only for efficiency reasons, a statement may be scattered over as many lines as you wish, and a single line may contain many statements (at some point the output will become a bit hard to understand though).
- You can use spaces and tabs to make your input easier to read, but it will not change the meaning of a statement: `d6;` and `d 6 ;` will return the same results.

A statement can be made up out of different syntactical elements, two of which are used in the example above: the "dice" operator `d` and a fixed number. You can of course use dice with different numbers of sides, or make up a statement that does not contain the dice operator at all (e.g. `42;`, quite boring!).

Other simple operators are `+ - * and /` which are simple arithmetic operations. To make things easier to understand you can also use braces to structure your input. Try the following examples and see whether you can understand the reasons for the output:

`d6 + 2;` and `d(6+2);`. In the first case a dice is rolled once, and then the number two is added to the result. In the second case 6 and 2 are added up, and the result is used as the number of sides for the dice.

If you are the adventurous type, you also have tried `d d6;`, a roll that is quite hard to roll with real dice, and `d6+d6;`. Adding up dice quickly gets tedious if there are many dice to roll, so there is the repetition operator `#`, using `3#d6;` does however not yield the desired result, but an error ("result is not scalar") instead, you will have to use `sum 3#d6;`, which is just ashorthand for `d6+d6+d6;`.

The reason for this is very fundamental to the understanding of dicelab, and you should take some time to make sure you understand it. A dice roll does not necessarily return a single number, but might return a set of numbers, e.g. when rolling multiple dice. Dicelab does not know what you want to do with this set, so it can't just add the dice up. You might e.g. multiply them instead or pick the highest. The result dicelab displays has to be a single value, a set with only one entry. So rolling `2#d6` returns a set with two values, `d6+d6` only a single one. The `sum` operator used above takes all values in a set and sums them up, resulting in a scalar value. In a similar way `prod` multiplies all values in a set and `count` counts the number of values. All are reducing a set to a scalar, the latter one does not make too much sense yet as all our sets had the predetermined lengths. But imagine the input `count (d6 # d6);`, this will roll one die, then use the result to determine how many dice to roll and count them. This of course leads to the same probability distribution as `d6;` alone, summing it up instead of counting might be more interesting.

There are some operators that are similar to the ones already covered, but have not been covered yet: `%` which is the modulo operator, the to-the-power-off `^`, scalar concatenation `.` whch turns `3 . 4` into "34" and the unary minus operator. Numbers can be negative, but many operators refuse to work with negative numbers, it is e.g. very hard to roll a dice with a negative number of sides!

Of course these operators are very basic, so we need a couple of slightly more complicated ones:

The already mentioned repetition operator `#` will take the set passed to it, repeat it and build a new set from it. The range operator `..` will build a set of consecutive numbers, e.g. in `sum 1..6;`. In a similar way the list concatenation operator `,` can be used to combine sets, e.g. in `sum(d6,d4);`.

Sets are in fact lists, which means they do have an order. The `perm` and `sort` operators randomize the set order or sort sets. The `rev` operator reverses the ordering in a set. This will only be usefull in combination with the `first` and `last` operators: `first 1 perm 1..6;` for example returns exaclty the same as `d6;`. This is only logical, as it builds a set of numbers from 1 to 6, shuffles them and then uses the first one as a result, effectively emulating a dice with playing cards! This operator can also be used to remove elements. e.g. in `sum drop first 2 perm 1..6;`. Instead of "drop" you can use "keep", which behaves just as if you would not have specified any of them, but is more readable. To avoid unnecessary `rev keep first 1 rev` fragments, there is also a `last` operator that behaves similar.

And before you ask: there are also `low` and `high` operators. These are used quite often in games, `sum high 3 5#d6` could e.g. be a character stat generator in a roleplaying game.

There is a set of filter operators that are used in a similar fashion: `count >= 5 3#d6` counts the dice that show a 5 or six in a set of three dice. Other filter operators are == (equals), != (not equals), < and > and of course <=.

To structure your expression in a better way and to solve some specific problems, you can use the `let` operator: it allows you to tie the result from a dice roll to a "variable" and use it in different parts of your expression. `let a = d6 in a+a` for example is the same as `d6*2` as opposed to `d6+d6`.

Similarily you can use `foreach` to apply an expression to each element of a set using a variable: `foreach v in A do B` will evaluate A, and then iterate over the set and set the variable `v` to the current element. For each element the expression `B` is evaluated. Example: `sum(foreach x in 2#d6 do x+1)` is the same as `sum(2#(d6+1))`.

The `if` operator allows you to branch depending on the result of some expression. The result of `if A then B else C` is the evaluation of `B` if the result of `A` is nonempty, and `C` otherwise. For example `if > 4 d6 then 1 else 0` is the same as `count > 4 d6`.

Finally the `while` operator allows you to use an unbounded repetition: if `x` is a variable and `E` and `F` are expressions, the `while x = e do f` is the list $v_0, v_1, \ldots, v_n$ where v_0 is the result of evaluating `E` and v_{m+1} is the result of assigning v_m to `x` and evaluating `F`, stopping at the first v_n which is empty. A good example is `sum(while x = d6 do keep == 6 d6)` which calculates a d6 that is re-rolled and added on sixes;

If you have longer files, you might find it usefull to put comments in your input that explain what a block of lines is doing. This can be done using \\ which instructs dicelab to ignore the rest of the line. It is also worth noting that braces can be used to make the operator precedence more explicit and the input easier to read.

This section is only meant as an introduction to the operators in question, please refer to section 3.2 for all the nitty-gritty details of an operator.

3.2 Reference

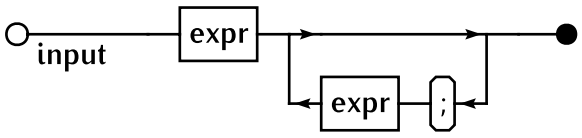
3.2.1 Scalars, Lists and Probabilities

To successfully use dicelab (and understand this reference section) it is important to understand the concepts of *scalars*, *lists* and *probabilities*. If we look at the very simple input `d6`, we will see that this dice rolling scheme will always yield exactly one result value. This is called a *scalar*. This scalar has multiple possible *values* (1-6) with a *probability* each. The output from dicelab is the list of *values* for the resulting scalar, ordered by their value, together with their respective probabilities. This means that the final result of a dicelab expression always has to be a scalar!

A *list* in contrast is a number of scalars, e.g. created by `2#d6`, the result of rolling two dice. This will not return a single number, but two, and cannot be used as a final result in dicelab. In most cases when we say "roll two d6" we actually mean "roll two d6 *and sum them up*", `d6+d6` or `sum(2#d6)` in dicelab syntax. Of course a list may contain only one value, in which case it can be used as a scalar, dicelab even uses only lists internally and does not have a seperate type for scalars.

The distinction between lists and scalars is still very important, not only because the final result must a scalar, but also because many operators only allow scalars for some of their arguments. For example `d6+2` makes sense, but `2#d6+2` does not: you can't really ad a number to a set of numbers!

3.2.2 Grammar

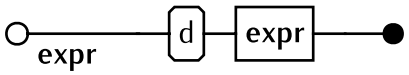


Dicelab accepts input consisting a number of *statements* seperated by semicolons, each of them will be executed on their own. In fact having multiple statements in one input is more or less the same as running dicelab multiple times with different input. Using this may be convenient in some cases, but may be confusing as well as it does clutter up the output a bit.

The last statement in the input does not need to be followed by a semicolon, but it is good practice to generally end all statements with one anyway. The reason for this is that when dicelab is running interactively by just reading from the standard input (e.g. by running `dicelab -c` and typing your statements directly into dicelab), dicelab will not know when you are finished with your current statement.

As noted above, each statement needs to result in a scalar in all cases, dicelab will complain if that is not the case.

3.2.3 Dice Operator

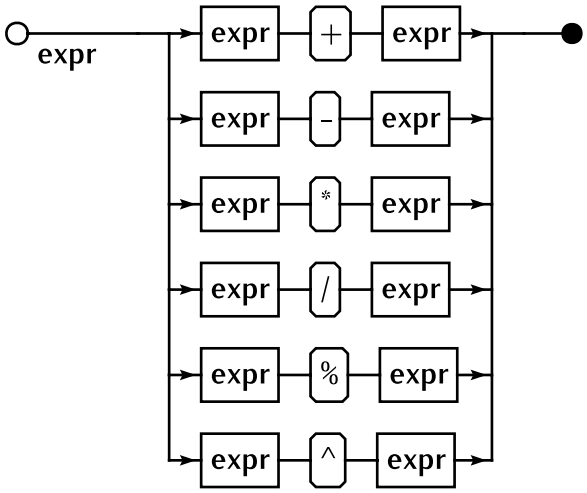


In the most simple form, when the argument is a simple number (e.g. `d5`), the dice operator will create a scalar with that many values and equal probabilities. If the argument is a scalar with multiple values (e.g. `d(d4)`) the argument is evaluated and the dice operator executed for each value. The results from all dice operator evaluations are then combined into one scalar by multiplying the probabilities of the value used and the dice operator result. Try the example above to see!

Typical examples are `d6` for a standard dice, `d20` for the 20-sided dice often used in fantasy roleplaying, `d2` or even better `d2-1` could be used to simulate a binary coin-toss. `d1` is technically possible, but a very boring dice in most cases.

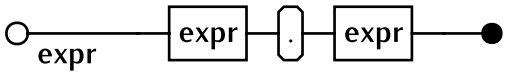
The dice operator only accepts scalars as arguments and always returns a scalar. All input values must be larger than zero.

3.2.4 Arithmetic Operators



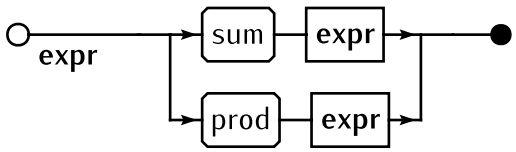
This family of operators takes two scalars as arguments and returns a scalar, performing the given mathmatical operation on them. The division operator always rounds down, so `5/2` yields 2. The modulo operator `%` returns the remainder of the division, so `5%2` yields 1. The power operator `^` return the first argument to the power of the second, so `2^8` is 256 and `10^3` is 1000.

3.2.5 Scalar Concatenation Operator



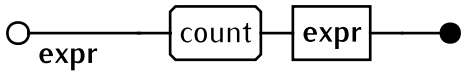
This operator takes two scalar arguments and returns a scalar that is the *text* concatenation of the two arguments. This is sometimes usefull (a `d100` could e.g. be constructed as a `d10.d10`), but in many cases it is safer to express things like that as multiplications (e.g. `d10*10+d10`). The reason for this is that the results are not padded in any way, so `12.3` is equivalent to `1.23`, both returning 123.

3.2.6 Summation and Product Operators



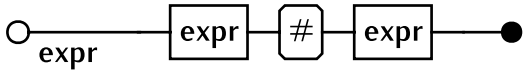
These two operators take a list as an argument, and turnthem into a scalar by either summing up or multiplying together all the scalars in it. So `sum(3#d6)` is the same as `d6+d6+d6` and `prod(3#d6)` is the same as `d6*d6*d6`. Please note that `sum(2#d6)` or `d6+d6` is *not* the same as `2*d6`, as the first case adds two *different* dice together, whereas the second case adds a dice to *itself*.

3.2.7 List Size Operator



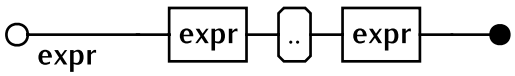
This operator takes a list as an argument and returns a scalar containing the number of scalars in that list as a value. For example `count(2#d6)` will always return two, and `count(d5#d3)` is equivalent to `d5`. This operator is normally only usefull in combination with filter opertors, e.g. when trying to determine how many dice beat a given threshold (for example `count(== 6 5#d6)`).

3.2.8 List Repetition Operator



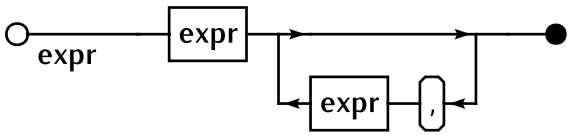
This evaluates an expression multiple times and puts all results in a list. A good example is `3#d6`, which is a list of the results of three dice rolls. Please note that this is the *list* of the results, not the sum. In essence this is similar to the list concatenation `d6,d6,d6`. Should the expression itself return a list, then these are concatenated together, so `2#3#d6` is the same as `6#d6`.

3.2.9 Range Operator



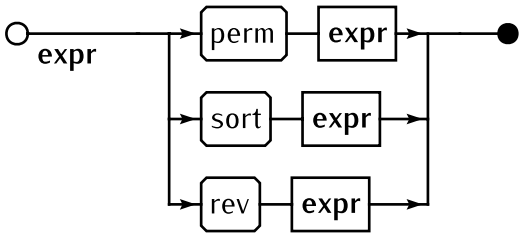
The range operator creates a list of values representing the given range, each value having a probability of 1. So e.g. `sum 1..3` returns 6. This operator is normally only usefull in combination with a loop operator like foreach or while, e.g. `sum foreach i in 1..6 do d(i)`, which is the same as `sum d1,d2,d3,d4,d5,d6`.

3.2.10 List Concatenation Operator



List concatenation is simply the combination of two lists, which of course could each be scalar. For example `sum 2#d6,d20` sums up a list of two 6-sided and one 20-sided dice, which is of course the same as `sum(2#d6)+d20`. Please note the usage of parenses in the last example, `sum 2#d6+20` is something slightly different, namely adding a d6 and a d20 to form a single result, and then summing up two of these...

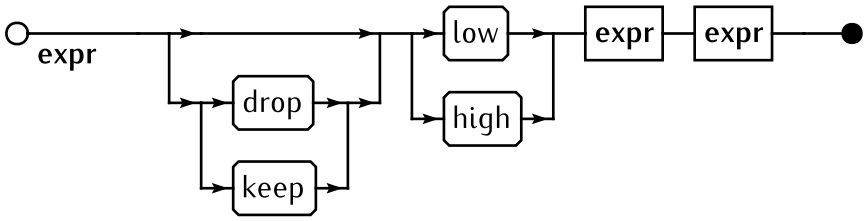
3.2.11 List Ordering Operators



These operators change the ordering of a list. *sort* obviously sorts the list by value, *rev* reverses the list and *perm* randomizes it. Please note that for many operators the ordering of a list has no consequences, e.g. `sum sort 2#d6` is the same as `sum 2#d6`, in fact

dicelab should optimize out the sort operator in this case. Also note that *perm* can be outstandingly expensive on all but the smallest lists.

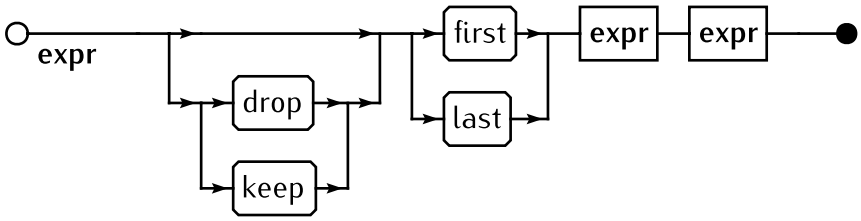
3.2.12 Low and High Operators



These filter operators remove some elements from a list. For example `sum drop low 1 3#d6` rolls 3 6-sided dice, removes the lowest roll and sums the other two up. You can change the behaviour by using of *high* instead of *low*, and by using *keep* instead of *drop*. This means that the statement `sum keep high 2 3#d6` is equivalent to the example above. You can also omit the *drop* or *keep*—, in which case the behaviour is just as if you would have specified *keep*. Please note that the first expression does not need to be constant, you could e.g. do a `sum high d6 6#d6`.

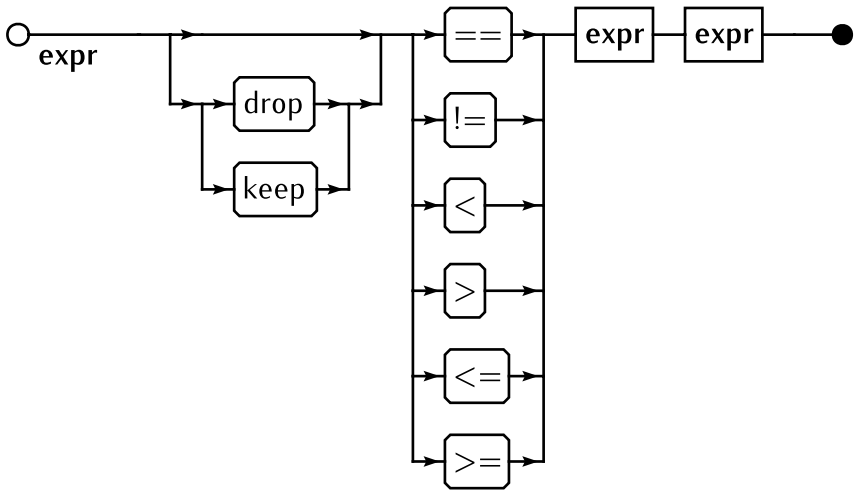
This operator does not change the ordering of the list, and might return less than you asked for, e.g. a `high 4 3#d6` of course does not differ from a `3#d6`.

3.2.13 First and Last Operators



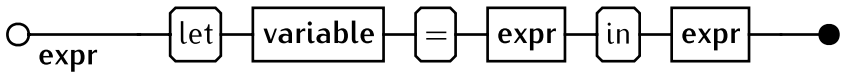
In a similar fashion, the *first* and *last* filters remove entries from a list, but not based on their value, but on their position in the list. So in essence a `sum low 1 3#d6` is the same as a `sum first 1 sort 3#d6`.

3.2.14 Filtering Operators



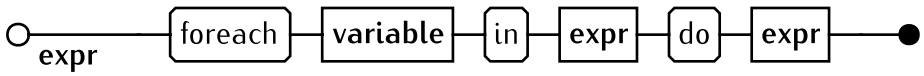
These also work in a similar way, but do not take the other values in the list into account. Instead they just filter on individual values, you could e.g. filter out all results less that a given value with `sum drop <= 2 3#d6`.

3.2.15 Let Operator



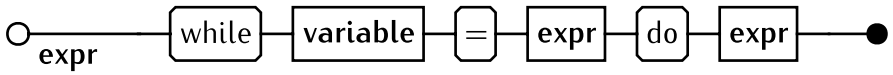
This assigns a list result to a symbolic variable which you can then use in subexpressions. For example `let A = d6 in A+A` is essentially the same as `d6*2`. Please note that this is very different from `d6+d6`, because in the example using `let` above, the two `A` refer to the same dice roll, whereas in `d6+d6` they refer to two *different* rolls, which can have different values. The result is that `let A = d6 in A+A` will never return 3, but `d6+d6` does (1+2). `d6+d6` is in effect `let A = d6 in let B = d6 in A+B`.

3.2.16 Foreach Operator



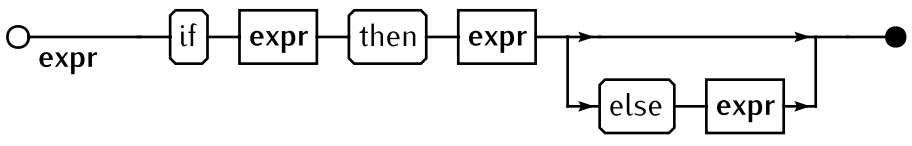
This is a bounded iteration, if `x` is a variable and `e` and `f` are expressions, then `foreach x in e do f` evaluates `e` and then assigns `x` to each element of the result in turn to evaluate f. This will result in a list which essentially is the result of `e` with `f` applied to all elements. For example `sum foreach X in 1..3 do d(X)` is the same as `d1+d2+d3`. Of course `f` can return a list for some of the evaluations, in which case the resulting list does not need to have the same length as the result of `e`. For example `sum foreach X in 1..5 do X#d3` is the same as `sum 15#d3`, and the result of the foreach expression has 15 elements, while the result of `e` in this case has only 5 (as can easily be checked with `count`).

3.2.17 While Operator



This is an unbounded iteration, if `x` is a variable and `e` and `f` are expressions, then `while x = e do f` if the list $v_0, v_1, \ldots, v_n$ where v_0 is the result or evaluating `e` and v_{i+1} is the result of assigning v_i to `x` and evaluating `f`, stopping at the first v_i which is empty. For example `sum while x = d6 do ((count == 6 x)#d6)` is rolling a d6, rerolling on a 6 and summing up the result.

3.2.18 If Operator



This is the branching operator. If `e`, `f`, and `g` are expressions, then `if e then f else g` gives `f` if `e` is nonempty, and `g` otherwise.
For example `let A = d6 in if (keep == 6 A) then d20 else A` rolls a d6 and uses the result except on a 6, in which case a d20 is rolled and used instead.

3.3 Examples

Count the number of dice greater than 7:

```
count >7 5#d10
```

Count the number of dice greater than 7 minus the number of dice equal to 1:

```
let c=5#d10 in (count >7 c)-(count ==1 c)
```

Count the number of rolls until a 6 is rolled:

```
count (while x=d6 do ((count <6 x)#d6))
```

Count the number of rolls until a 6 is rolled, more efficiently:

```
count (while x=(d6/6) do ((count <1 x)#(d6/6)))
```

Roll attributes for a new D&D character:

```
6#sum(drop low 1 4#d6)
```

Roll on the 11..66 morale check table in The Gamers' Civil War Brigade Series:

```
d6.d6
```

Find the median of 3 d20s:

```
high 1 low 2 3#d20
```

3d6 with rerolls on 6s:

```
sum(while x=3#d6 do ((count ==6 x)#d6))
```

Roll 7 d10 and find the largest sum of identical dice:

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

The Fibonacci sequence is defined by Fn = Fn-1 + Fn-2, with F1 = F2 = 1.

```
Calculate the first twenty Fibonacci numbers:
let n = 20 in
  let f = (1,1) in
    foreach i in 1..n do
      let f = (f,sum(high 2 f)) in
        if ==n i then f else ()
```

Risk has battles where the attacker rolls 3d6 and the defender rolls 2d6. The highest attacker die is matched with the highest defender die and the second highest attacker die to the second highest defender die. For both matches, the highest wins, with ties going to the defender. The number of attacker wins:

```
let a = 3#d6 in
  let b = 2#d6 in
    count( (<(high 1 a) high 1 b),
           (<(high 1 low 2 a) low 1 b))
```

Storyteller die roll with target number 8 and botches indicated at -1:

```
let c=5#d10 in
  let succs = count >7 c in
    let ones = count ==1 c in
      if >0 succs then high 1 (0,succs-ones)
      else if >0 ones then -1 else 0
```

Combat in Silent Death is rather complex. Three dice are rolled. If their sum is above a target, the roll is a hit. To calculate damage, the same dice are sorted. If all three are equal, all are summed to yield the damage. If the least two are equal, but the third is higher, the high die is the damage. If the two highest are equal, but the third is lower, the two high dice are summed to yield the damage. If all three dice are different, the middle die is the damage. This example assumes that the dice are two d8s and a d10, with a target number of 15:

```
let x = 2#d8,d10 in
  (count >15 sum x)#
    let a = low 1 x in               // low die
    let b = high 1 low 2 x in        // middle die
    let c = high 1 x in              // high die
      if ==a ==b c then a+b+c        // all equal
      else if ==a <c b then c        // two low equal
      else if >a ==c b then b+c      // two high equal
      else b
```

4 Feedback

Dicelab has gotten a bit more complicated than I originally hoped, both the implementation and the usage. Making it simpler to use, extend and maintain is therefore quite high on the list of things to do. To do this I need your help: please give feedback on what you are using it for, what you find hard to do, what you found hard to understand when using it first, what is particularily vague in the manual, what expression is especially slow and generally what you do and don't like.

Robert Lemmen <robertle@semistable.com>