# **Troll**: A Language for Specifying Die-Rolls

Torben Mogensen
`http://www.diku.dk/~torbenm/Troll`

February 14, 2013

**Abstract**

Role-playing games (RPGs) 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 tens, doubling the value of doubles, removing the lowest or highest result, counting the number of dice that are below a treshold, or a multitude of other variations and combinations of these.

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 **Troll**, a simple, yet powerful, declarative language for defining how dice are rolled. Such definitions are then used to emulate die-rolls or make probability calculations.

## 1 Overview of Troll

**Troll** is derived from an earlier language called **Roll**, in some sense you could call **Troll** a second version of **Roll**, but since **Troll** isn't backwards compatible with **Roll**, I have decided to give the new language a different name.

This first section describes the language **Troll**. If you want to run the examples while reading the text, the easiest approach is to open a browser window to the **Troll** online interface (`http://topps.diku.dk/torbenm/troll.msp`), enter the examples in the text window, select "Roll" or "Calculate probabilities" and then press "Troll the dice!".

**Troll** 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.

**Troll** can make random rolls of the defined dice-roll method or it can calculate a probability distribution. The probability distribution is calculated as exactly as the numeric precision and the bound on rerolls allows, so it is much more precise than statistics obtained by sampling a large number of rolls.

## 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.

"d" or "D" can be prefixed by a number so, for example, "5d6" means five six-sided dice. Note that these are not added, to do so you will have to write "sum 5d6". The reason is that we often want do do something other to the dice than adding them, for example we may want to count how many are above a threshold. The number preceding "d" need not be a constant, so you can, for example, write "(4+1)d6" or even "(d4)d6".

You can add, subtract, multiply and divide 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* rolling 3 individual dice. "-" also works as a unary (prefix) operator, so you can, for example, write "-d7". Arithmetic only works on singleton collections, so you can't write "3d6+4". You must instead write "sum 3d6+4"

A variant of the d operator is z, where z$n$ generates a roll of a die numbered from 0 to $n$. A standard ten-sided dice is normally numbered from 0 to 9, so this can be written as z9. z$n$ is equivalent to d$(n+1)$-1. Like d, z can be prefixed by the number of dice and it can be written using a capital letter, e.g., 3z9 or Z4.

## 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*.

We saw that we could roll a number of dice by prefixing "d" with a number. We can, similarly, repeat any dice specification $n$ times with 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 5d6)" to specify a collection of ten values, each obtained by adding 5 six-sided dice. 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 (union) of two collections is done by the operator "U", for example in "3d6 U 3d8" 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 U 3d8" or "3d6 U d8". For backwards compatibility, you can also use "@" as an union operator.

An alternative way to build collections is by listing the elements in curly braces, for example

"`{1, 2, 1}`" for the collection consisting of two ones and a two. Note that the order is irrelevant, but the number of times an element occurs is significant. A special case is "`{}`", which describes the empty collection. The elements listed in the braces need not be constants or even singletons, you can, for example, write "`{d6, 3d8}`", which is equivalent to "`d6 U 3d8`". In fact, "{$a_1$, . . . ,$a_n$}" is equivalent to "$a_1$ `U`. . .`U` $a_n$", so use either according to taste.

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.

You can randomly (with equal probability) choose an element from a collection with the operator "`choose`". For example, "`choose {-1,0,1}`" returns either -1, 0 or 1 with equal probability. "d*X*" is equivalent to "`choose (1..`*X*`)`". If the argument to "`choose`" is the empty collection, an error is reported. Related to "`choose`" is "`pick`", which randomly picks a specified number of elements from a collection without replacement. For example, "`{1..10} pick 3`" picks three different numbers between 1 and 10. If there are fewer elements in the first argument than specified in the second argument, the net result will be equal to the first argument. Note that, while "`pick`" selects different *elements*, these need not have different *values*. For example, "`{1,2,2}  pick 2`" can return the collections `{1,2}` or `{2,2}` (but not `{1,1}`). "`pick`" can be used to emulate drawing of cards.

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 = 10d6`" rolls 10 six-sided dice and uses the filter "`6 =`" to keep only the dice that are equal to 6 and then count these. You can also filter by less-than ("`<`"), greater-than ("`>`"), less-than-or-equal ("`<=`"), greater-than-or-equal ("`>=`") or different-from ("`=/=`"), so, for example, "`count 4 < 10d6`" counts the number of dice that are greater than 4. The number before the comparison operator can be any integer-valued expression (i.e., singleton collection), so you can write, e.g., "`count d6 > 10d6`" that rolls a die and counts how many of the 10 next dice that are less than this. Note that "*x* `>` *y*" is not the same as "*y* `<` *x*". The first will return the elements of *y* that are less than the singleton *x* and the second will return the elements of *x* that are greater than the singleton *y*. Even if both *x* and *y* are singletons, there will be a difference. You can add several filters, so for example "`3<= 5>= d6`" returns only values between 3 and 5, otherwise returning the empty collection.

Another way to remove elements from a collection is with the "`drop`" operation. "*x* `drop` *y*" removes from *x* *all* elements that are also found in *y*. For example, "`2d6 drop 3`" will remove all threes from the two d6s. Another example is "`d6 drop d6`", which as a net effect will return the empty collection if the two dice are equal and return the first die if not.

Dual to "`drop`" is "`keep`": "*x* `keep` *y*" keeps from *x* all elements that are also found in *y*. For example, "`(d6+d6) keep {5,7}`" adds 2 d6 and keeps the result if it is in the collection {5,7}. So the result is either the empty collection, 5 or 7. Another example is "`d6 keep d6`", which as a net effect will return the empty collection if the two dice are different and return the first die if they are equal. Note that "`keep`" is not multiset intersection, as "*x* `keep 6`" keeps *all* sixes in *x*, where multiset intersection would keep only one.

Also similar to "`drop`" is "`--`", which is multiset subtraction. Unlike "`drop`", "`--`" removes only as many occurrences of elements as there are in the second argument. So "`{2,2,3} -- {2,4}`" returns "`{2,3}`", where "`{2,2,3} drop {2,4}`" returns "`{3}`".

You can remove duplicates from a collection with the "`different`" operator. For example, "`different {2, 1, 2}`" returns the collection consisting of 1 and 2 once each. To count the number of different values in a collection *x*, you can write "`count different` *x*".

The smallest value in a collection can be found using the operator "`min`", for example "`min 3d6`" finds the smallest of three dice. Similarly, "`max`" finds the largest value in a collection. If the argument to "`min`" or "`max`" is the empty collection, the empty collection will be retruned.

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 3d6`" 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. "`min`" is equivalent to "`least 1`" and "`max`" to "`largest 1`". Similar operators are "`minimal`" and "`maximal`". Where "`min`" returns a single copy of the minimal value in its argument and is undefined on empty collections, "`minimal`" returns all copies of the minimal value and is defined on the empty collection (where it would return the empty collection). Similarly, "`maximal`" returns all maximal values in its argument.

The *median* (middle) of a collection is the least value *m* in the collection that has the property that (at least) half the values in the collection are less than or equal to *m*. For example, the median of the collection $\{2,6,23\}$ is 6, as two values (which is half of three values, rounded up) in $\{2,6,23\}$ are less than or equal to 6. If there is an even number of values in the collection, the median is the smallest value in the largest half of the collection. Note that some definitions of median say that the median of a collection is the average of the two middle elements. But since that is not always an integer, we have chosen the above definition. You can find the median of a collection using the operator `median`, so, for example, `median 3d20` finds the middle value of three rolled d20s.

## 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 "`x := d6; 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 in the assignment and after the semicolon. The name of the bound variable must consist of letters only and may not be identical to an operator name. You can make multiple assignments, such as "`x := d6; y := d6; x*x*y*y`". A defined value is visible only in the following expression, so for example "`(x := d6; x) U x`" leaves the second "`x`" undefined, since the definition isn't visible outside the parentesis. Local definitions do not owerwrite earlier definitions (they only temporarily hide them), so "`x := 2; (x := 1; x) U x`" returns the collection “`{1, 2}`”. Note that you can’t use variables that are named the same as operators, including `d` and `z`.

You can 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 2*x else max (x U y)`” to take the largest of two dice but let doubles count double.

You can make a probabilistic choice with arbitrary probability by using the `?` construct. `?`$p$, where $p$ is a decimal fraction less than 1 (written as $0.d_1 \cdots d_n$, where $d_1 \cdots d_n$ are decimal digits), will return `1` with probability $p$ and the empty collection with probability $1-p$. By writing `if ?`$p$ `then` $x$ `else` $y$, you can choose $x$ with probability $p$ and $y$ with probability $1-p$.

If you want to do something equivalent to a logical **or**, you can use `U`, as if any of the arguments are non-empty, so is the result. For example, “`if x = 2 U y = 3 then 42 else 24`” returns 42 if either `x = 2` or `y = 3` and otherwise returns 24. Intersection on collections wouldn’t work as logical **and**, so a special operator “`&`” is added for this. $e_1$ `&` $e_2$ is equivalent to `if` $e_1$ `then` $e_2$ `else {}`, *i.e.*, if $e_1$ is empty, the empty collection is returned, otherwise $e_2$ is returned. So, only if both collections are non-empty will a non-empty collection be returned. As an example, writing “`if x = 2 & y = 3 then 42 else 24`” will get you 42 back if both `x = 2` and `y = 3` and 24 if not.

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`”.

You can, for example, specify that you roll 7 d10s and take the highest sum of identical dice in the result:

```
c := 7d10; max (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

A die-roll can involve repeating rolls until a certain condition occurs. For example, a d7 can be simulated by rolling a d8 until the result is less than 8. Ignoring for a moment that **Troll** supports d7s directly, we can write this as

```
repeat x:=d8 until x<8
```

We can also repeat while a certain condition holds, so we can equivalently write the above as

```
repeat x:=d8 while x=8
```

A slightly more useful example is rerolling two d6s until they are different:

```
repeat x:=2d6 until (min x)=/=(max x)
```

The `repeat` constructs simply repeats the assignment until the condition is true (for `until`) or false (for `while`), returning the last value. Note that `x` is visible only in the condition, so it can’t be used after the loop, nor in later iterations of the same loop.

Sometimes certain conditions adds more dice to the roll. For example, World of Darkness adds an extra dice for every 10 rolled, repeating this for 10s in the new dice. For such mechanisms, we use the `accumulate` construct. This works like the `repeat` loop, except that it returns a collection of the values from all iterations instead of only the last. For example,

```
accumulate x:=d10 while x=10
```

might return the collection "10 10 4" (though the numbers will be shown in a different order). To be precise, a World of Darkness roll counts how many dice from a pool of `N` dice have a value over 7, adding dice to the pool for every 10 rolled. This can be written as

```
count 7< N#(accumulate x:=d10 while x=10)
```

When calculating distributions, **Troll** puts a limit to the number of iterations in `accumulate` (but not on `repeat`, which it calculates exactly). The default is 12, but this can be changed by a command-line parameter, see section 8. If you are unsure if the limit is high enough, try changing it by 1 or 2 and see how much the result changes. If you find the change too high for comfort, increase the limit.

## 6 Functions

You can define functions that you can call later. Function definitions are placed either before or after the expression that calculates the actual roll. For example, you can write

```
function mul(v) =
  if v then (min v)*call mul(largest ((count v)-1) v)
  else 1

call mul(5d10)
```

Here, `mul` is defined as multiplying the elements of a collection. `mul` is then called with `5d10` as argument. Hence, we get the product of five d10. Note that the function can call itself, i.e., it is recursive. You can have several mutually recursive functions such as:

```
function even(n) =
  if n=0 then 1 else call odd(n-1)

call even(d9)

function odd(n) =
  if n=0 then 0 else call even(n-1)
```

In the above, `even` returns 1 if its argument is even and 0 if it is odd, so calling it with a d9 as argument would return 1 slightly more often than 0 (on average 5 out of 9 times). Note that function declarations appear both before and after the main expression.

Functions can use all of the features of **Troll**, including dice rolls:

```
function down(n) =
  x := d n;
  if x=1 then 1 else x + call down(x)

call down(10)
```

which rolls a d10 followed by a die with as many sides as that roll and so on until a 1 is rolled and adds the results.

Note that while functions give great flexibility and make a lot of the other features of **Troll** (such as loops) redundant, it is often much faster to calculate probabilities when dice are defined by the standard operators and loops than by using functions. Hence, you should only use functions if all else fails.

When calculating distributions, there is a limit to call depth. The call-depth limit is the same as the limit on iterations of `accumulate`. If a call is made when the limit is exceeded, the non-recursive branch of the body is used (regardless of its probability). If there is no conditional or no non-recursive branch, 0 is returned. This is a rather arbitrary value, so you should design your functions so the probability of exceeding the limit is small (or increase the limit until this is the case). The `mul` function above will have a call depth equal to the number of elements in the collection that is used as argument to `mul`, so the limit should be at least this high. The `down` function can have unbounded call depth, but the chance that it is deeper than the default limit (12) is small. Hence, the calculated probability distribution will be close to the "real" distribution. As with `accumulate`, you can try changing the limit to get an idea of how much it affects the result.

Note: Due to the way scopes of variables are handled in **Troll**, variables defined on the command-line are not available inside function definitions.

## 6.1 Compositional functions

Functions like the above can be very slow to calculate probabilities for, as the calculator has no information about properties that might be used to speed up calculation. In order to allow faster calculation of user-defined functions, it is possible to define *compositional* functions.[1]. A compositional function is defined by three cases:

1. The function's result when applied to the empty collection. This is specified as a value.
2. The function's result when applied to a singleton collection. This is specified as a the name of a function to apply to singletons.
3. The function's result when applied to a union of two collections. This is specified as the name of a function of two arguments, which will be applied to the results of applying the composite function to the two sub-collections.

As an example, we can define a compositional function `product` that multiplies all the elements of a collection (so it is equivalent to `mul` as defined above).

[1]homomorphims, really

```
compositional product(1,id,times)

function id(x) = x

function times(x,y) = x*y
```

The product of the empty collection is 1, the product of a singleton is the value itself and the product of a union is obtained by multiplying the products of the components of the union. The composite `product` is much faster than `mul` and it doesn't suffer from the limit on call depth.

The declarations of `product`, `id` and `times` can be in any order. Compositional functions are called like other functions (with `call`).

The last two elements of the triple that defines a compositional function can be user-defined functions or predefined operators (with some restrictions described below), so you can define `product` as

```
compositional product(1,sum,*)
```

as an alternative to the above.

If a compositional function defined by the triple $(v_0, f, g)$, $v_0$ and $g$ must obey the following laws for all values $x, y, z$ in the range of $f$:

$$\begin{array}{lcl} g(v_0,x) & = & x \\ g(x,y) & = & g(y,x) \\ g(x,g(y,z)) & = & g(g(x,y),z) \end{array}$$

In other words, $v_0$ is a neutral element of $g$ and $g$ is commutative and associative. If this is not the case, the defined function is not compositional and the calculation may get wrong results. The laws are not checked by **Troll**, so it is up to the user to ensure that they are true. If you use predefined operators for $g$, **Troll**restricts these to commutative and associative operators, i.e, `+`, `*`, `@` and `U`. Predefined operators allowed for $f$ are `-`, `d`, `D`, `z`, `Z`, `sum`, `count`, `min`, `max`, `minimal`, `maximal`, `choose` and `different`.

As can be seen, compositional functions must be used with care, but they can greatly speed up probability calculation, and some functions (such as `product`) are most easily defined compositionally.

## 7 Text

You can do simple text formatting in **Troll**. Text in **Troll** are rectangular boxes of text that you can combine horisontally and vertically to form larger boxes.

If you combine two boxes of different width or height, blank lines or columns are added to the smaller box to make them equally wide or high before combining, so the combined box will be rectangular. How this is done depends on which operator is used to combine the boxes:

$b_1$ `||` $b_2$**:** The boxes $b_1$ and $b_2$ are combined so $b_1$ is to the left of $b_2$. Blank lines are added to the bottom of the shorter box to make them of equal height before they are combined.

$b_1$ `|>` $b_2$**:** The boxes $b_1$ and $b_2$ are combined so $b_1$ is on the top of $b_2$. Blank spaces are added to the right of the narrower box to make them of equal width before they are combined.

$b_1$ `<|` $b_2$**:** The boxes $b_1$ and $b_2$ are combined so $b_1$ is on the top of $b_2$. Blank spaces are added to the left of the narrower box to make them of equal width before they are combined.

$b_1$ `<>` $b_2$**:** The boxes $b_1$ and $b_2$ are combined so $b_1$ is on the top of $b_2$. Blank spaces are added on both sides of the narrower box to make them of equal width before they are combined.

The three different vertical combination operators allow columns to be left-aligned, right-aligned or centered. For example, `"1" |> "two" |> "three"` produces the box

```
1
two
three
```

while `"1" <| "two" <| "three"` produces

```
    1
  two
three
```

and `"1" <> "two" <> "three"` produces

```
  1
 two
three
```

The combining operators group to the right, so `"1" || "two" <> "three"` corresponds to `"1" || ("two" <> "three")` and produces

```
1 two
 three
```

by putting the box containing `1` before the box with the centered text of `two` and `three`.

You can put the combining operators inside strings, so `"1" <> "two" <> "three"` can be abbreviated to `"1<>two<>three"`. Note that parentheses inside strings are just characters and don't group the operators, so `"(1<>two)||three"` is the same as `"(1" <> "two)" || "three"`.

## 7.1 Making rolls into strings

Adding a single quote (`'`) in front of an expression that produces a collection of numbers (e.g., `3d6`) converts the collection into a string. Adding a number *n* in front of the quote produces *n* of the following expression and puts each sample on a separate line in a right-aligned box. For example, `6 ' sum largest 3 4d6` might produce

```
 8
13
 3
10
11
 6
```

You can combine this with a text column:
`"Str |>Dex|>Con|>Int|>Wis|>Chr" || 6'sum largest 3 4d6`
to produce, e.g.,

```
Str 18
Dex  8
Con  6
Int 15
Wis 16
Chr 11
```

# 8 An implementation

**Troll** has been implemented in Standard ML, using the Moscow ML implementaion of that language.

You can run **Troll** either as a stand-alone program on your PC or through the **Troll** web interface located at `http://topps.diku.dk/torbenm/troll.msp`.

The web version of **Troll** is fairly easy to use, but has some restrictions compared to the full version (but presents the results in a nicer way by using HTML). In particular, when computing probability distributions, long-running calculations will be aborted. However, the web interface should suffice for most uses.

The stand-alone program, called "`troll`" is run from a command-line as described below[2]. To run the program, you need to install Moscow ML, see section 8.3 below.

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

[2]So you need to open a command-line window before using it. I recommend using a text editor that supports command-line windows (such as XEmacs).

- A filename. The file should contain a definition of a die roll. If several filenames are specified, the last on the line is used. If no file name is specified, the definition is read from the terminal. Input is terminated by the end-of-file character (control-D in Unix/Linux and control-Z in DOS/Windows).
- 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, the average, spread and mean deviation is also printed out. When calculating these, the empty collection is treated as having the value 0. If the specified number is 0, the default number of iterations for the `accumulate` construct is used.[3] 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. Note that strings doesn't make sense in probability calculations, so you get an error message if you try to use strings or string operators when calculating probabilities.
- 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 "`troll`" 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.t` containing a definition like the following:

```
count T <= N d10
```

You can call **Troll** like this:

```
troll test.t 0 N=7 T=5
```

and get the following output:

```
  Value     % =                 % >=
     0 :     0.16384          100.0
     1 :     1.72032           99.83616
     2 :     7.74144           98.11584
     3 :    19.3536            90.3744
     4 :    29.0304            71.0208
     5 :    26.12736           41.9904
```

[3]The default number is currently set to 12

```
     6 :   13.06368            15.86304
     7 :    2.79936             2.79936

Average = 4.2    Spread = 1.29614813968  Mean deviation = 1.0450944
```

The -p option switches to printing probabilities as numbers between 0 and 1, so the command `troll -p test.t 0 N=7 T=5` produces

```
  Value  Probability for =   Probability for >=
     0 :    0.0016384           1.0
     1 :    0.0172032           0.9983616
     2 :    0.0774144           0.9811584
     3 :    0.193536            0.903744
     4 :    0.290304            0.710208
     5 :    0.2612736           0.419904
     6 :    0.1306368           0.1586304
     7 :    0.0279936           0.0279936

Average = 4.2    Spread = 1.29614813968  Mean deviation = 1.0450944
```

The -g*x* option makes the program print a graph of the probability distribution (using ASCII graphics). For example, `troll -g1.3 test.t 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  Mean deviation = 1.0450944
```

Note that there is no space between the "g" and the following number, which specifies the number of bars printed per percent of probability.

## 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, which is used to implement **Troll**. Definitions in **Troll** use the traditional "-", though. You can use both "~" and "-" to specify negative numbers on the command line, but in Unix/Linux you should avoid using "~", as it has a special meaning when used in the command line.

- The program thinks that any command-line argument that starts with a digit is a number, so you can't use file names like "`7thSea.t`" (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.
- Definitions of spread and mean deviation can be found in the article `RPGdice.pdf`, that is distributed as part of the **Troll** package. The article is a general discussion about dice mechanisms for role-playing games.
- In both Linux/Unix and Windows, you can redirect the output to a file using the > character. For example,

  ```
  troll test.t 0 N=7 T=5 > output.txt
  ```

  puts the output of the command into the file `output.txt`.

## 8.1 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. 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, which would be the case for straightforward enumeration. But it can not always do so.

In a few cases, normalization can make things run faster. This is the case when the possible collections are fairly small *and* there are many ways to obtain the same collection.

Binding a value to a variable will force normalization of the distribution for that value. Often, that will make the program take longer, *e.g.*, in `x := 10d8;(min x)+(sum x)` compared to `(min 10d8)+(sum 10d8)`, where the former takes *much* longer. Note that the two rolls do not give the same distribution. If a bound variable is used only once, it is automatically substituted by its bound expression, so no extra cost is incurred.

Note that you will have to use a definition (and, hence, force normalisation) if the same value is used twice, it is a good idea to throw away unneeded information before normalization. For example, if you have

```
x := 10d8; count 1=x + count 8=x
```

you don't really need values between 2 and 7, so these can be filtered out before binding to `x`:

```
x := 10d8 drop (2..7); count 1=x + count 8=x
```

which reduces running time dramatically. Alternatively, we can process each die individually and add the results:

```
sum 10#(x := d8; count 1=x + count 8=x)
```

which is even faster.

If your definition uses `accumulate`, 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 is the dice system from L5R and 7th Sea. Here, you roll N open-ended d10s and add the M largest of these. M and N can vary. You can express this as

```
sum (largest M N#(sum accumulate x:=d10 while x=10))
```

and run it by, for example,

```
troll 0 L5R.t M=2 N=3
```

This takes about 50 seconds on my (fairly old) 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.*, four:

```
troll -4 L5R.t M=3 N=4
```

While the smaller limit reduces precision, the chance of having more than 4 rerolls on a die is sufficiently small that it matters little in the grand picture.

## 8.2 Sampling rolls

Some probability distributions may take so much time to calculate exactly that you might want to sample and analyse a large number of rolls instead of waiting for the exact result. You can use **Troll** to do the sampling and analysis: You make one **Troll** definition that generates the samples and combines them to a simple **Troll** definition that chooses one of the samples randomly. Running the generated definition through **Troll** (using the probability distribution option) will calculate the distribution of the samples.

As an example, consider the **Troll** definition below:

```
"choose {" || (S'
   (sum (largest M N#(sum accumulate x := d10 while x=10))))
|| (((S-1)'",") <| "}")
```

The middle line is the **Troll** definition of the roll (the L5R roll defined above) and the other lines turn `S` samples of this roll into a string that forms a **Troll** definition of the form `choose {`$N_1,\ldots N_S$`}`, where each $N_i$ is a sample. Assuming this definition is in a file `sample.t`, you could run it by a command

```
troll sample.t S=10000 M=5 N=8 > samples.t
```

to generate 10000 samples in the file `samples.t`. You would then run

```
troll 0 samples.t
```

to analyse the samples to get a probability distribution. More samples will make the sampled distribution closer to the exact distribution, but the probability of rare results in particular will be inexact even at high sample numbers. Running several times and comparing the different estimates for the same probability should give you some idea of the precision.

For example, in three different runs with M=2 and N=4 and S=100000 samples, I got estimates for a result 50 of 0.015%, 0.018% and 0.011%, so there is considerable doubt about the last digit. The exact (up to rounding errors) probability is 0.0178846929%.

In conclusion, you should only use the sampling method to get rough estimates when you can't wait for the exact result (or run out of memory when calculating it).

## 8.3 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 for Linux/Unix and Windows from
`http://www.itu.dk/~sestoft/mosml.html`.

Get the **Troll** package from `http://www.diku.dk/~torbenm/Troll` and unpack `Troll.zip` in an empty folder. Use WinZip, Info-Zip or some other unzipper that supports long filenames.

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

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

```
unzip Troll.zip
bash compile.csh
```

This will compile the sources and produce an executable called `troll`. A few warnings are shown during compilation, 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 chronological order:

**(February 2013)** Added the `median` operator. This has been optimised for common cases such as `median` *x*d*y*, but works (albeit sometimes slowly) in all cases.

**(October 31 2012)** Fixed an error in the parser that made it give a parse error on `d5 d6` and similar combinations.

**(June 2012)** Added `--` operator and allow predefined operators in addition to functions in `compositional` declarations.

**(November 2011)** Added the `keep` operator.

**(June 2011)** Added `minimal` and `maximal` operators and allows text values in probability calculations.

**(November 2010)** Fixed bug in `foreach` construct.

**(October 2010)** Added `?` construct.

**(September 2010)** Allow function declarations both before and after main expression.

**(September 2010)** Add memoisation to user-defined functions. This dramatically speed up some calculations.

**(May 2010)** Inline affine local bindings when calculating probabilities.

**(February 2010)** Added `U` as synonym for `@`.

**(October 2009)** Added web interface (`http://topps.diku.dk/torbenm/troll.msp`).

**(October 2009)** Fixed bug in implementation of `choose`.

**(May 2009)** Added compositional function definitions.

**(March 2009)** Added `z` operator.

**(July 2008)** Added `pick` operator.

**(June 2008)** Added `&` operator.

**(June 2007)** Various minor optimizations and clean-up of code.

**(May 2007)** Optimized `choose` for very long lists.

**(May 2007)** Added functions.

**(April 2007)** Optimized probability calculation for `accumulate`.

**(March 2007)** Added text boxes (section 7).

**(January 2007)** Added `different` operator.

**(October 2006)** Added mean deviation to printout.

**(October 2006)** Added `drop` operator, reduced rounding errors in accumulated probability.

**(September 2006)** Updated from **Roll** to **Troll**.

## Differences between Roll and Troll

- `d` and `D` can now be prefixed with a number, which saves a lot of uses of `#` and brings the notation closer to traditional RPG notation.
- Collection-building using curly braces has been added. In particular, this allows easy specification of the empty collection (`{}`).
- The `let` $x$=$e_1$ `in` $e_2$ construct has been replaced by $x$`:=`$e_1$`;` $e_2$. The difference is syntactic only.
- Filters are now infix instead of prefix, so you write `3< x` instead of `>3 x`.
- A number of new operators have been added: `min, max, choose, drop, different` etc..
- The `let-in-repeat` construct from **Roll** has been replaced with two simpler constructs, `repeat` and `accumulate`. The old construct is closer to `accumulate` than it is to the new `repeat`, so most uses of the old construct should be translated to use `accumulate`. The old construct was more powerful than the new constructs, but considerably harder to use, and I never found any real-world example where the extra power was useful.
- Where **Roll** in addition to the probability for being equal to a value printed probabilities for a result being less than the value, **Troll** prints the more useful (for games) probability of a result being greater than or equal to the value.

## 9 Summary

**Troll** is a language for defining die-rolls. Simple methods are very easy to describe and even moderately complex methods can be described in one line of text. Even unbounded rerolls and complex comparisons and calculations can be described.

Unlike most other tools, **Troll** makes probability calculations whose accuracy is limited only by the available precision of floating-point calculations, with the exception of `accumulate`, where the maximal number of iterations can limit precision further. Even then, the accuracy is much higher than what can be obtained even with a ridiculous number of random samples.

Calculation of probability distributions can in some cases be slow, since the program may have to fall back on enumerating all possible outcomes.

# Appendix: Language summary

| | |
|---|---|
| d$n$ or D$n$ | roll one d$n$ (a die labelled $1-n$) |
| $m$d$n$ or $m$D$n$ | roll $m$ d$n$ |
| z$n$ or Z$n$ | roll one z$n$ (a die labelled $0-n$) |
| $m$z$n$ or $m$Z$n$ | roll $m$ z$n$ |
| `+, -, *, /` | arithmetic on single values |
| `sum` | add up values in collection |
| `count` | count values in collection |
| `U` or `@` | union of collections |
| {$e_1,\ldots,e_n$} | union of $e_1,\ldots,e_n$ |
| `min, max` | minimum or maximum value in collection |
| `minimal, maximal` | all minimum or maximum values in collection |
| `least` $n$, `largest` $n$ | $n$ least or $n$ largest values in collection |
| `median` | median of a collection |
| $m$#$e$ | $m$ samples of $e$ |
| $m$..$n$ | range of numbers from $m$ to $n$ (both inclusive) |
| `choose` | choose value from collection |
| $e$ `pick` $n$ | pick (without replacement) $n$ values from collection $e$ |
| `<, <=, >, >= , =, =/=` | filters: Keep values from 2nd argument that compare to 1st argument |
| `drop` | elements found in 1st argument and not in 2nd |
| `keep` | elements found in 1st argument that are also in 2nd |
| `--` | multiset difference |
| `different` | remove duplicates |
| `if-then-else` | conditional. Any non-empty is considered true |
| ?$p$ | Returns 1 with probability $p$ and {} otherwise |
| `&` | substitute for logical **and** |
| $x$ `:=` $e_1$`;` $e_2$ | bind $x$ to value of $e_1$ in $e_2$. |
| `foreach` $x$ `in` $e_1$ `do` $e_2$ | evaluate $e_2$ for each value in $e_1$ and union the results. |
| `repeat` $x$ `:=` $e_1$ `while/until` $e_2$ | repeatedly evaluate $e_1$ while or until $e_2$ becomes true (non-empty). Return last value |
| `accumulate` $x$ `:=` $e_1$ `while/until` $e_2$ | repeatedly evaluate $e_1$ while or until $e_2$ becomes true (non-empty). Return union of all values |
| `function` | define function |
| `compositional` | define compositional function |
| `call` | call function |

| | |
|---|---|
| `'` | make text box of single sample |
| $n$ `'` | make text box of $n$ samples (right-aligned) |
| `\|\|` | Combine text boxes horisontally |
| `\|>` | Combine text boxes vertically, left-aligned |
| `<\|` | Combine text boxes vertically, right-aligned |
| `<>` | Combine text boxes vertically, centre-aligned |

## Variables, spaces and comments

Variable names are sequences of letters (both upper and lower case), so they can not contain digits. Nor can variable names be identical to operators (e.g., `d`, `z` or `sum`). Names are case-sensitive. Note that, unlike most programming languages, **Troll** doesn't allow digits in variable names.

Spaces are optional in most places, but space is required between two adjacent numbers and between two adjacent alphabetical operators or variables. For example, "`dX`" is a variable called "`dX`", while "`d X`" is a die with X sides. Similarly, "`Nd6`" is a variable called "`Nd`" followed by the number 6, while "`N d6`" is a collection of N d6s.

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

## Operator precedences

Operator precedences from highest to lowest are:

| Grouping | Operators | | | | | | |
|---|---|---|---|---|---|---|---|
| prefix | prefix `D` | prefix `d` | prefix `Z` | prefix `z` | | | |
| right | infix `D` | infix `d` | infix `Z` | infix `z` | `#` | | |
| right | `=` | `<` | `>` | `<=` | `>=` | `=/=` | |
| prefix | `choose` | `count` | `sum` | `min` | `max` | `least` | `largest` |
| | `different` | `minimal` | `maximal` | `median` | `'` | | |
| prefix | prefix `-` | | | | | | |
| left | `*` | `/` | | | | | |
| left | `+` | infix `-` | | | | | |
| right | `U` | `@` | `&` | | | | |
| left | `drop` | `keep` | `pick` | `--` | | | |
| none | `..` | | | | | | |
| right | `\|>` | `<\|` | `<>` | `\|\|` | | | |
| n/a | `else` | `while` | `until` | `do` | | | |
| right | `;` | | | | | | |

Note that the list of operators with same precedence as `choose` is split over two lines.

When two operators from the same group can be combined, they are grouped left or right according to the left column of the table above, so for example `3 < 6 > x` is equivalent to `3 < (6 > x)` while `d6 + d8 - 3` is equivalent to `(d6 + d8) - 3`. Parentheses can be used to overide the precedences.

There are two cases of minus: A prefix minus binds more tightly than `*`, but infix minus binds like `+`. Similarly, prefix `d` binds tighter than infix `d`.

Note that you can not group several occurences of `..`, i.e., `2..3..4` is not valid syntax.