Introduction

ELDARScript started as a simple dice rolling specifier for The Dicenomicon (e.g. `3d6`) and has evolved over the years to support a much richer environment for role playing mechanics. It takes inspiration from the language Fortress (especially with the use of full unicode operators) and NewtonScript (used to power the Apple Newton). It is not designed to be a general purpose language, but one tailored to rolling dice (and then figuring out what the results mean).

This document describes version 2.0 of the language, which is implemented in "The Dicenomicon +1" (aka "dN+1"). Previous versions of The Dicenomicon have slightly different support - primarily in terms of a few missing features (e.g., rational values) as well as bug fixes and minor inconsistencies that have been resolved.

ELDARScript is a dynamically typed language - what kind of data is being manipulated (numeric values, textual strings, etc...) is determined entirely at run time. It is also case insensitive - you can use upper or lower case and it doesn't matter.

Basic Values

Numeric Values

ELDARScript provides standard mathematical ways of working with numeric values. By default, numbers are integers (whole numbers), such as `5` and `-23`. Standard operations can be performed:

Expression	Value	Notes
`8 + 3`	11	Standard addition
`8 - 3`	5	Standard subtraction
`8 × 3`	24	Standard multiplication
`8 / 3`	2	Standard division, truncating the fractional result if any
`-8 / 3`	-2	While the result is actually -2 ⅔, we truncate the result which brings it towards zero (or in this case 2)
`8 ÷ 3`	3	Standard division, rounding
`8 /+ 3`	3	Standard division, rounding all fractions up (taking the "ceiling" of the result)
`8 /- 3`	2	Standard division, rounding all fractions down (taking the "floor" of the result)
`8 /= 3`	2 ⅔	Exact division, giving a rational value (see "Rational Values" for rational values)
`8 ∨ 3`	8	Higher value of the two
`8 ∧ 3`	3	Lower value of the two
`8 ≫ 3`	8	If the left value is any value other than zero (false), the right value is ignored, otherwise the left value is ignored (useful to provide a default value in case something isn't initialized)

Since dividing 8 by 3 results in 2 ⅔ (not an integer), there are multiple different versions of division to produce an integer result by either truncating (rounding towards zero), rounding

to the nearest value, rounding up, rounding down, or not rounding at all. Note that truncation may seem identical to rounding down, but it is not for negative values (truncation of negative values results in rounding up). If you want an exact fractional value, see "Rational Values" for more details about rational values.

Numbers can be compared, resulting in either a "`1`" (true) or "`0`" (false):

Expression	Value	Notes
`8 = 3`	0	Equal
`8 ≠ 3`	1	Not Equal
`8 > 3`	1	Greater Than
`8 ≥ 3`	1	Greater Than or Equal To
`8 < 3`	0	Less Than
`8 ≤ 3`	0	Less Than or Equal To

Also useful (with these comparison results) is the ability to see if both are true or one or the other is true ("and", "or" respectively). We can actually use the "lower" and "higher" operators exactly like "and" and "or". This works because the lower of two boolean values is going to only be true (1) if both are true (1). Similarly, with higher, the result will be true (1) if either are true (1)

Expression	Value	Notes
`1 ∧ 1`	1	True and True = True
`1 ∧ 0`	0	True and False = False
`0 ∧ 1`	0	False and True = False
`0 ∧ 0`	0	False and False = False
`1 ∨ 1`	1	True or True = True
`1 ∨ 0`	1	True or False = True
`0 ∨ 1`	1	False or True = True
`0 ∨ 0`	0	False or False = False

String Values

ELDARScript also includes the ability to have string (textual) values. These can be enclosed by either a single quote at the start and end, double quote at the start and end, or left and right "typographic" double quotes:

```
1.'Single Quotes'
2."Double Quotes"
3.“Typographic Quotes”
```

The ability to use different styles allows you to embed one inside another easier. Within a single our double quoted string, you can precede that quote with a backslash to make a literal quote (part of the string, not ending it) such as `"Double Quotes with \"quotes\" inside it"`. Note that strings can also have formatting in them - which also use a back slash to indicate the start of formatting (escaping quotes is just a simple version of that). This formatting can be quite rich, including the ability to change color, font, size, and even embed images and icons in it. See “Advanced String Operations” for more details.

Strings can also be operated on to combine them:

Expression	Value	Notes
`“Hello” + “World”`	“Hello World”	Concatenate the string with a space between them. A space is added automatically since this is normally what you want
`“Hello” - “World”`	“HelloWorld”	Concatenate the string without a space between them
`“Hello” × 3`	“HelloHelloHello”	Repeat and concatenate
`“a, b, c, d” / “, ”`	(“a”, “b”, “c”, “d”)	Splits a string into a list (see “Lists” below).
`(“a”, “b”, “c”, “d”) × “|”`	“a|b|c|d”	Takes a list and joins it back together, separating each element by the string
`“a=5, b=6” ÷ “a=%#, b=%#”`	(5, 6)	Parse a string and extract parts of it from a format pattern (see “Advanced String Operations”)
`“There is a % potion” % “red”`	“There is a red potion”	Formatting values into a string (see “Advanced String Operations”). Note that the right hand side can be a single value or a list
`“Hello” × “World”`	0	Currently undefined. Any expression that uses operators that doesn’t have a well defined behavior for the types involved will result in attempting to use numeric operations

Strings can also be compared using the standard comparison operators, but this will result in comparing the strings in a case insensitive form (and ignoring diacriticals) and also treating numbers as numbers. For example:

Expression	Value	Notes
`"Hello" = "hello"`	1	Equal since case is ignored
`"Aardvark" < "Zebra"`	1	Compare alphabetically
`"AC 5" < "AC 10"`	1	The 5 and 10 are treated as numbers and compared, so this is true

Mixing Strings and Numbers

In some cases, there is an explicit operation performed when mixing strings and numbers in an expression (for example, the repeated concatenation operator when you multiply a string by a number). In other cases, it isn't explicitly defined - this also applies to some cases when both operators are strings. In both of these cases, the string(s) attempts to be converted into a number, and then the numeric operation is performed (and if not, the number is converted to a string and string based operations are performed).

Expression	Value	Notes
`"3" × 5`	"33333"	Normal string repeat and concat
`3 × "5"`	15	The string 5 is converted into a numeric 5 and normal multiplication happens
`"3" × "5"`	15	Both strings values are converted to numbers and normal multiplication happens
`"3 score" × "5 years"`	15	Both strings values are converted to numbers (which ignores the first non-numeric part) and normal multiplication happens
`5 + "score"`	"5 score"	

This can be useful to convert between strings and numbers. Note that the first example we get string repeating - we may want to make sure that we use a number - we do this by multiplying the number `1` by the string. Similarly, we can force something to a string by using the format operator.

Expression	Value	Notes
`1 x "5"`	5	Force the second value to become a number
`"%" % 5`	"5"	Format the value as a string
`string(5)`	"5"	The `string` function will also convert a simple value into a string (but also performs special formatting on frames - see below)

Obviously, this isn't all that useful for constant values, but when we have a variable of unknown type, these two idioms do the trick.

List Values

Besides being able to operate on single values, ELDARScript expressions include native support for immutable lists of values. For example, you can roll six dice and get list of the each of the results (`6d6@`). You can also create a list by having two or more values separated by commas. Lists can contain other values such as numbers, strings, or fields, but can *not* contain other lists - doing so will just "flatten" the list by appending them all together.

The magic of ELDARScript is that you can then operate on all of the values in the list at the same time (in positional pairwise form between values in each list). Furthermore, if you operate on a list and a single value or a single value and a list, it treats that single value as if it were a list with as many elements as the list (there are excepts such formatting via the string `%` operator or joining via the x operator).

Expression	Value	Notes
`1, 2, 3`	(1, 2, 3)	Concatenate each value into a list
`(1, 2, 3), 4`	(1, 2, 3, 4)	Concatenate the 4 to the list
`(1, 2, 3), (4, 5, 6)`	(1, 2, 3, 4, 5, 6)	Concatenate the two lists together. Note that this does not create a "list of lists" - concatenation via the comma operator always flattens the list
`(1, 2, 3) x 5`	(5, 10, 15)	Each element of the list is multiplied by five and produces a new list

Expression	Value	Notes
`(1, 2, 3) x "/"`	"1/2/3"	One of the exceptions - a list multiplied by a string will join all elements of list separated by the string and produce a new string. This is other half of taking a string and splitting it (division) by another string which produces a list.
`10 + (1, 2, 3)`	(11, 12, 13)	Ten is added to each element
`(1, 2, 3) + (4, 5, 6)`	(5, 7, 9)	Each element of the first list is added to the corresponding element in the second
`(1, 2, 3) + (4, 5)`	(5, 7, 3)	Each element of the first list is added to the corresponding element in the second, with the second being padded out with zeros.
`(1, 2, 3) ≥ 2`	(0, 1, 1)	Returns a new list (of zeros and ones for false and true) after applying the "≥" operator to each element to find which ones are greater than or equal to 2
`1 … 10`	(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)	The … operator creates a list starting at a given value and extending to include the next
`(1, 3 … 7, 9)`	(1, 3, 4, 5, 6, 7, 9)	The … operator works inside a list
`(1, 4) … (7, 10)`	(1, 4, 5, 6, 7, 10)	The last value of the first list and the first value of the second list are used to determine the range to fill in
`5 in (1, 3, 5, 7)`	1	The *in* operator allows you to determine if a value is contained in a given list
`(1, 3, 5, 7) in 4 … 8`	(0, 0, 1, 1)	Each value of the list on the left is tested to see if it is contained in the list on the right (created via the … operator)
`× 3 : "b"`	("b", "b", "b")	The `× (count) : (value)` construct is an easy way to make a list of multiple items

Expression	Value	Notes
`("a", × 3 : "b", × 5 : "c", "d")`	("a", "b", "b", "b", "c", "c", "c", "c", "c", "d")	And combining it with other list concatenation makes an easy way to make a "weighted" list (where if you pick an item at random, you are more likely to get a "c" or "b" than either "a" or "d" in this example)

There are a couple of important things to note about list operations. First is that lists are immutable values (just like numbers and strings are - you can make a new one, but you can't change an existing one). These operations will create a new list. Second is that there is a special case difference a single value and a list with multiple values - the former will be padded out with the first value repeated, the second will add zeros to the list as needed to make it the same length as the other list. **Furthermore, a list with one element is identical to the value of that single element (and a single element is identical to a list with one element).**

Lists support a number special functions. Note that passing a list value as a parameter is the same as passing each element of the list as a separate parameter (i.e., function calls actually take a single list value which is automatically constructed via the comma operator between the parameters). A number of these functions are considered to be "numerically parameterized" functions where there is a number in the name of the parameter. This will be documented as, for example, "`high#`" which means that any integer can be used there (e.g., "`high3`", "`high1`") or even a parameter ("`high#1`"):

Expression	Value	Notes
`high2(1,2,3)`	(2,3)	**high#** - returns a list of the N largest elements in a list. Also, low#
`nth2(5,1,4,7)`	4	**nth#** - Find the Nth number when sorted from low to high
`last3(1,3,4,6,7)`	(4,6,7)	**last#** - Returns the last N items. If N is more than the number of items in the list, it just returns the entire list. Also, first#
`gt3(5,1,4,7)`	(5, 4, 7)	**gt#** - Returns the values in the list greater than the value N. Also, eq#, ne#, lt#, le#, ge#
`sum(5,1,4,7)`	17	Sum up all the values in the list
`sort(5,1,4,7)`	(1, 4, 5, 7)	Sort the values into a new list. Also rsort (for reversed sorted)

Expression	Value	Notes
`count5(5,1,5,7)`	5	**count#** - Returns the number of elements equal to the value N. This is equivalent to sum(eq#(…))
`reverse(5,1,4,7)`	(7,4,1,5)	Reverse the list
`allof(1,1,1)`	1	Returns 1 if all of the values are "true" (non-zero); Also anyof which returns 1 if any of them are.

Let's suppose you have a roll where you need to roll three six sided dice (in three colors) and only pass if the first one is 4 or more, the second is 5 or more, and the last is 6. This could be done like this:

```
1. @A ← Rd6.
2. @B ← Gd6.
3. @C ← Bd6.
4. if @A ≥ 4 ∧ @B ≥ 5 ∧ @C ≥ 6 then
5. "good"
6. else
7. "fail"
8. end
```

(The @A is a "local variable" - a place to save a result, covered in more detail in the *Variables* section. The ← is an assignment operator used to save a value there. if then else end are used to conditionally alter things, discussed in the *Flow Control* section below)

We maintain each roll, and explicitly test each value vs a different threshold and if they are all true, we are good, otherwise we fail. If we wanted this with more than three dice tests, it would start to get verbose. We can instead do this:

```
1. @A ← (Rd6,Gd6,Bd6).
2. if allof(@A ≥ (4, 5, 6)) then
3. "good"
4. else
5. "fail"
6. end
```

In this case we keep the values as a list, and compare them vs their respective thresholds and we are good if all of those comparisons are true. Lists are especially useful when you don't

know how many dice will be rolled (where it is a parameter). For example, we roll a bunch of six sided dice and we are only good if the highest die is 4 or more than the lowest:

```
1. @A ← #1d6@.
2. if high1(@A) ≥ low1(@A) + 4 then
3. “good”
4. else
5. “fail”
6. end
```

Subscription

If you have a list of values, you can access individual values via subscripting

Expression	Value	Notes
`(“a”,”b”,”c”)[1]`	“a"	First value
`(“a”,”b”,”c”)[2]`	“b"	Second value
`(“a”,”b”,”c”)[3]`	“c"	Third value
`(“a”,”b”,”c”)[4]`	0	Values beyond the end are zero
`(“a”,”b”,”c”)[0]`	0	As are values before the start
`(“a”,”b”,”c”)[“hello”]`	0	As are values that aren’t numeric subscripts
`(“a”,”b”,”c”)[“2”]`	“b”	But like other operators, if the string can be converted into a number, it will be

For convenience, strings can also be treated as if they were “records” and be subscripted. A record string is of the form `“field1:value1|field2:value2|…”` and provides a simple and easy way to keep track of multiple values (NB: Earlier versions of ELDARScript used an equal sign instead of a colon between field names and values - that syntax is still supported but not recommended):

Expression	Value	Notes
`“a:Alpha|b:Bravo|c:Charlie|1:one”[“a”]`	“Alpha"	The value of the field named “a”
`“a:Alpha|b:Bravo|c:Charlie|1:one”[1]`	“one”	Subscripts can also be numeric, and match the corresponding field with that value for a name
`“a:Alpha|b:Bravo|c:Charlie|1:one”[“d”]`	0	Missing values are zero

Expression	Value	Notes
`"a:Alpha|b:Bravo|c:Charlie|1:one|:other"["d"]`	"other"	If no field name is specified, this entry is treated as the default value for all other entries

Since a list is immutable (can't be changed, just like numbers and strings), ELDARScript has a way to manipulate individual elements which **returns a new list with changes**. It does **not** alter the original list:

Expression	Value	Notes
`("a","b","c")[2] ← "beta"`	("a","beta","c")	A new list with the second element set to the string "beta"
`("a", "b", "c")[4] ← "d"`	("a", "b", "c", "d")	Setting the fourth element adds to the end
`("a", "b", "c")[5] ← "d"`	("a", "b", "c", "d")	Since setting the fifth element is beyond the end of the list, the new value is just added to the list
`("a", "b", "c")[0] ← "d"`	("d", "a", "b", "c")	Setting a value before the start of the list (zeroth element) will prepend it to the start of the list

Note that since all simple values are also list values with a single element, attempting to set a numeric subscript to a value will create a new list with the original value and the new value.

Expression	Value	Notes
`"alpha"[2] ← "beta"`	("alpha","beta")	Treats "alpha" as a list with one element, so this appends "beta" to the end of it
`"alpha"[0] ← "beta"`	("beta","alpha")	Treats "alpha" as a list with one element, so this prepends to the start of it
`"alpha"[1] ← "beta"`	("beta")	Treats "alpha" as a list with one element, which is replaced by "beta", resulting in a list with a single element

This can also be applied to string based records:

Expression	Value	Notes
`"a:Alpha|b:Bravo|c:Charlie|1:one"["b"] ← "Beta"`	"a:Alpha|b:Beta|c:Charlie|1:one"	Replace one of the field values

Expression	Value	Notes
`"a:Alpha|b:Bravo|c:Charlie|1:one"["d"] ← "Delta"`	"a:Alpha|b:Bravo|c:Charlie|1:one|d:Delta"	Adds a new field value
`"a:Alpha|b:Bravo|c:Charlie|1:one"["2"] ← "two"`	"a:Alpha|b:Bravo|c:Charlie|1:one|2:two"	Note the use of a string as a subscript here - this will add to the record…
`"a:Alpha|b:Bravo|c:Charlie|1:one"[2] ← "two"`	("a:Alpha|b:Bravo|c:Charlie|1:one","two")	…but in this case, we get a list with two elements (since this corresponds to the case above where we use list subscripting with numeric subscripts). Record manipulation only works with string subscripts

Frame Objects

While string based records provide an easy way to manage multiple associated values in a single entity, it is limited to simple string based keys (field names) and values. Frames are created by using an open curly brace (to indicate the start of the frame) followed by a field name (which is either a string or a numeric value), a colon, and then some sort of value (which can be any sort of expression, including another frame). Each field/value pair is separated by the vertical bar character, and a closing curly brace is used at the end:

```
1. {
2.  "a":"Alpha"
3. | "b":"Beta"
4. | 1:"One"
5. | 2:"T" + "w" + "o"
6. | "Meaning of Life":6 × 7
7. | ":default:" : "???"
8. | ":before:" : "Zero or negative"
9. | ":after:" : "Lots"
10.| ":length:" : 2
11.}
```

This example is similar to the record `"a:Alpha|b:Beta|1:One"` but with several difference. First, the curly braces on lines 1 and 10 are used to show it is a frame - though the same colon is used between keys and values and vertical bar is used between pairs of keys and values. Also note that the keys are either strings or numbers - strings need to be quoted (a string record treats all keys as strings). More importantly, each value is actually an expression - in line 5 we are concatenating strings together to form the string `"Two"` while in

line 6 we value `6 x 7` to get `42`. Next, we have a "special" string `":default:"` that indicates what the default value is (for use when the subscript doesn't correspond to one of the field name values). Finally, we've got three other special strings. `":before:"` is used if the index is less than one, `":length:"` is used to say how many integer subscripts we have on this frame (so we can pretend that this is a list). If the index is greater than `":length:"` we return the value `":after:"` for anything greater than that value. So, assuming we stored the above frame into a local variable `@X`:

Expression	Value	Notes
`@X[1]`	"One"	The string "One" (line 4)
`@X[2]`	"Two"	The expression "T" + "w" + "o" (line 5)
`@X[3]`	"Lots"	Since 3 is more than the ":length:" of the frame (line 10), we return ":after:" (line 9). Note that if ":length:" were defined to be 4, this would instead use the ":default:" value and result in "???"
`@X[0]`	"Zero or negative"	Since 0 is less one, we return ":before:" (line 8)
`@X["a"]`	"Alpha"	Line 2
`@X("d"]`	"???"	The ":default:" value (line 7)

You can also use the `":first:"` key to define where the array starts (instead of 1 by default) so all values less than that are the `":before:"` or `":default:"` value. If no `":default:"` value is defined, like other subscripting, the result will be zero.

One very important difference between string based records and frames is that **frames are mutable** - as a result, the subscript setting actually *does* alter the frame (the now modified frame is also returned).

Frames as Lists

As documented above, frames can act like lists, or, more accurately, lists that can have other associated properties besides integer indexed values. This only happens if the frame includes the `":length:"` key - without it, you'd still be able to access and change the data, but the frame wouldn't look fully like a list.

Rational Values

Rational values are special forms of numbers that represent exact values when dividing an integer (or other rational values) by another integer (or rational value). For example,

dividing one by two results in ½, a rational value (i.e., being expressed as a ratio of two whole numbers). While not commonly used in game, there are a number of cases where they are. The simplest example is when dividing up treasure (and then making change as needed). 59 gold divided by 4 people leaves a bit left over, which we can then, by keeping track of the result more accurately with rational number, come up with a more even distribution (by making change in silver and copper if needed).

Flow Control

Being able to do simple expressions, we've already hinted that we can alter the flow of the program based on calculated values. This is done via an `if then else end` construction, similar to what is found in most programming languages. What is interesting is that this is also an expression (i.e., there is no difference between an *expression* and a *statement* - everything creates a value). So while something like JavaScript would have code like this:

```
1.if (a > 10) {
2.   b = 5
3.} else {
4.   b = 6
5.}
```

We can do this:

```
1.@B ← (if @A > 10 then
2.  5
3.else
4.  6
5.end)
```

Namely, the value of the `if` expression is used to assign to the variable. Granted, this is a simplistic example, but common enough that languages such as JavaScript have special operators such as `?:` which become special if then else expressions (e.g., `b = a > 10 ? 5 : 6`).

ELDARScript is specially designed to support having dice rolled, with special features to make it be aware that dice take "time" to come up with a result. This is true with expression, especially with our conditional expressions. So

```
1.if d20 > 10 then
2.  "Hit for" + d8
3.else
4.  "Missed"
5.end
```

will roll a d20, wait to see what the result is, and then roll the d8 (but only if the result was more than 10)

Chained conditionals

Instead of writing:

```
1.if @A > 10 then
2.  “A Big”
3.else
4.  if @B > 10 then
5.    “B Big”
6.  else
7.    “Small”
8.  end
9.end
```

we can combine lines 3 & 4 together and omit line 8:

```
1.if @A > 10 then
2.  “A Big”
3.elif @B > 10 then
4.   “B Big”
5.else
6.   “Small”
7.end
```

These two expressions are otherwise identical, the second is just easier to read

Iteration

ELDARScript as a special flow control operator to do things multiple times named `for`. It looks like a classic *iterator* found in most programming languages. It has a special local variable that is created during the loop, as well as list to iterate through. However, since everything is an expression, you may wonder what sort of expression it returns? It generates a list of all the expressions created by the body of the loop (this makes it work like a `map` function found in many languages). So:

```
1.for @i in 1 … 5 do
2.   @i x 3
3.end
```

will create a list (3, 6, 9, 12, 15). (If you thought “it would be easier to do `(1 … 5) x 3` to make this”, you are exactly correct).

NB: Do not try to roll dice while inside a for loop - it will not work (and even if it did, it probably wouldn't work like you expect).

You can ignore the result of the for loop and it works much more like a "traditional" loop:

```
1.@x ← 0.
2.for @i in 1 … 5 do
3.   @x ← @x + @i
4.end
```

This will take the sum of numbers 1 through 5 and store the result into the local variable `@x`. In case you are curious, the `for` loop will create a list (1, 4, 9, 14, 19), since it will take the value of the assignment operator in each pass of the list.

Comments

Once you start making complex formulas, you'll probably want to add comments to the code to help you document how it works. You can always throw in a string followed by a period (since the period will drop the previous value - the string - and use the next value), but ELDARScript provides for a specific comment capability:

```
1.❝Start by setting our total to zero❞
2.@X ← 0.
3.❝Iterate through the values❞
4.for @i in 1 … 5 do
5.   ❝Keep a running total❞
6.   @X ← @X + @I
7.end
8.❝And at this point we've got that total❞
```

The use of "heavy double comma quotation mark ornament" indicates a comment. This can also be typed by an exclamation point and double quote

```
1.!”Start by setting our total to zero”!
2.@X ← 0.
3.“!Iterate through the values!”
4.for @i in 1 … 5 do
5.   !“Keep a running total”!
6.   @X ← @X + @I
7.end
8.!”And at this point we've got that total”!
```

The editor will convert those to the “”. Unlike most programming languages, however, ELDARScript is somewhat picky about where they go - they are actually part of the language itself, rather than some lexical whitespace that can be ignored. More specifically, they can go before a statement, after a period, or after the last statement. They do not, however, have a value, nor can they, for example, go in the middle of some parameters (i.e., `foo(1 “count”, 2, “a” “name”)` is illegal. There “heavy” appearance is designed to emphasize that they go between lines (as the editor would format them).

Variables

Previous examples have already shown variables exist - the various things like `@a`, `@b`, `@x` and the like (for historical reasons, numeric local variables such as `@1`, `@2`, etc... were also valid). These are all just places to store values, using the ← operator, which takes a variable on the left and an expression on the right and stores it into the variable.

These examples are *local variables* - variables that exist within the given dice roll/formula/etc... But what if you want something like keeping track of the stats of a character? There are also *global variables*. These look just like local variables (e,g, `@STR`), but you can't change them via the ← operator (if you try, it makes a local variable instead). Instead, there are other parts of the UI that let you edit them (if you really need to change them, there is a special way to do so).

The reason for this has a lot to do with the evolution of the Dicenomicon, as well as some unique features. Since dice take time to roll on the screen, we want to bunch together as many dice as possible, and hold off dealing with the results of this until they have finished rolling. Originally, this was done by evaluating the formula to find all the dice rolls, rolling them, looking at the results, and then re-evaluating the formula with those results. This means that there were side effects from the first pass which could interfere with the second (i.e., a global variable could be set in the first pass and then the new value used in the second pass to get the "real" results). As a result, there originally were no "side-effects" allowed from evaluating a formula - you got a value, and that was it.

To set global variables, values from formulas included special "augmented" features, which included a list of side effects to perform. So when we did it the first time, we ignored these augmented side effects, and only when we did it the second did we apply them - in this case, to alter the global variable (there are other potential side effect actions as well such as playing a sound).

With the current version of the Dicenomicon, there is internal support for values such as dice rolls that are "future promises" of values, and it knows how to handle those. In a future version of ELDARScript, these explicit side effects will be handled in a much cleaner fashion.

Parameters

Suppose you have a nice formula, but want to reuse it, changing, for example, the number of dice rolled, or a target number that you are comparing it to. This is the purpose of parameters - variables that bound values when the user selects the formula. This puts them half way between a global variable and a local one. What's more, there is an explicit UI

presented that asks for what the value of these parameters should be, complete with a prompt and definition of possible valid values.

Parameters look slightly different. For example: `#1`. Two things to note - the use of a pound sign to indicate a parameter, and the use of numbers instead of letters (this is primarily historical, but due to how they are used, it is difficult to change this - since they are parsed just like normal numeric constants, the use of letters would confuse things horrendously).

Nearly any place you can have a constant numeric value, you can use a parameter. In some of those places you could use a local or global value (such as a value in an expression), but in many places you can't (such as the number of dice to roll, or how many sides they have). Furthermore, these values are a constant - you aren't allowed to change their value inside your formula. You can, however, initialize them to another value, so long as that value is based on a constant or another parameter. This is handy when you, for example, need to roll N dice and N-1 dice:

```
1.#2 ← #1 - 1.
2.if (#2Gd6) > #1Rd6 then
3.   "Success"
4.else
5.   "Failure"
6.end
```

Note how line 1 initializes the value of #2, while line 2 uses both #1 and #2 to determine how many dice to roll (the "R" and "G" indicate the color of the dice). In case you are wondering why line 2 has parenthesis around `#2Gd6`, it has to do with an unfortunate side effect of "reducers" (see *Dice* section below)

What is not possible, however, is to define a parameter based on an arbitrary expression such as a dice roll - i.e., you can not roll a die to determine the number of dice to roll using parameters (in theory, it is possible to do this for a limited set of possible dice). This is due to the feature of The Dicenomicon that allows you to re-roll dice (using "fate points") - if you rerolled that first die, what to do with already rolled dice becomes an enigma.

Roll References

Finally, it is possible to refer to dice rolls more than once. For example, if you want to roll 6 six sided dice and compare the highest 3 with the lowest 3 (this example is easily solved without using roll references via various list functions) and if the high 3 are more than 3 times the low three, we succeed. To do this, we can refer to a previous dice roll using a roll reference such as `$1`. Looking through the formula, the first dice roll is `$1`, the next is `$2`, etc... So our example would look like:

```
1.if 6d6H3 > $1L3 x 3 then
2.   “Success”
3.else
4.   “Failure”
5.end
```

In this case `$1` refers to the `6d6`. The `H3` and `L3` are “reducers” which are explained in the *Dice* section below. More specifically, the roll reference applies to the dice before any roll reducers or roll macro values are applied. This can also be written without references as:

```
1.@1 <- 6d6@.
2.if high3(@1) > low3(@1) x 3 then
3.   “Success”
4.else
5.   “Failure”
6.end
```

In general, use roll references only when absolutely required - they are subtle and prone to break (for example, editing to add an additional dice roll will screw up the reference). Furthermore, roll macros often include side effects such as re-rolling dice, which doesn’t happen with roll references

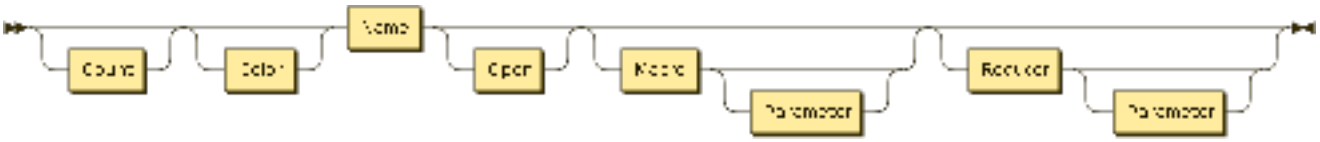
Operator Precedence

Below is a list of the operators in ELDARScript. Operators with higher precedence are evaluated before operators with lower precedence (so `1 + 2 x 3` is 7 - the multiplication of 2 and 3 is done before the addition - not 9). Operators with the same precedence are evaluated left to right

Precedence	Operator	Notes
1	.	Expression terminator
	,	List combiner
2	←	Assignment
	`if then [elif then] else end`	Conditional
3	∧	Min
	∨	Max
4	> ≥ = ≠ < ≤	Comparison
	in	Membership
5	…	Range
6	+	Addition
	-	Subtraction
7	×	Multiplication
	%	Modulo
	/ ÷ /+ /- /=	Division
8	[ ]	Subscript
9	( )	Grouping

Dice

Finally, we get to the dice rolling part, the whole reason ELDARScript and The Dicenomicon even exist (historic note: the original version of The Dicenomicon had a sort of flow chart based UI to specify how many dice to roll, what kind, and things like addition and constants, but was otherwise very limited). Specifying what dice to roll includes not only what specific kind of die to roll, but also how many, what color, special "open ended" roll, other special rolling instructions ("roll macros") and how to convert a bunch of dice to a single value ("reducer"). In a simplified form, it looks like this:



There are two kinds of dice - numeric dice and textual dice. The former results a number, the latter some sort of text. Given the flexibility of EDARScript, you wouldn't think it would matter, but while it makes sense to take the sum of rolling 5 six sided dice, it doesn't make as much sense to take the sum of five dice that pick a random character alignment.

Numeric dice are broken into standard dice that show 1 through N, a variant that go from 0 to N-1 ("zero based dice"), and a handful of special dice (such as percentile dice). There are also custom numeric dice (with whatever values you want). All textual dice are effectively custom dice - some are built in, but there is no real standard otherwise to follow. Finally, there are composite dice (percentage dice are conceptually a composite die - it doesn't really roll a d100, it rolls two d10s and combines the results), as well as step dice (which change based on the number of dice roll, used for games like Earthdawn).

All dice include the ability to specify a color (from 11 possible colors - custom dice can be tinted into just about any color).

Code	Color
R	Red
G	Green
B	Blue
C	Cyan
M	Magenta
Y	Yellow
K	Black

Code	Color
W	White
A	Gray (as in “gr_A_y”)
O	Orange
N	Brown (as in “brow_N_”)

ELDARScript also includes support for specifying the size of dice which appears before the color (but after the count), but current The Dicenomicon only support a single size (and the syntax is subject to change).

Code	Color
~F	Fine
~D	Diminutive
~T	Tiny
~S	Small
~M	Medium (usually omitted)
~L	Large
~H	Huge
~G	Gargantuan
~C	Colossal

Most numeric dice then support three extra features that textual dice do not - open rolls, roll macros and reducers. These features are what allow The Dicenomicon to support such a wide range of games.

Sides

Most dice are simple dices that go from 1 to a given value (4, 6, 8, 10, 12, 20 are the most common). These then correspond to specific geometric shapes with that many sides. Besides these values, The Dicenomicon has geometric shapes for 14, 16, 18, 24, 30, 48 and 60 sides. It can further more, by using multiple sides for the same value, create dice with 2, 3, 5, 7, and 9 sides, and knows how to combine multiple dice for 40, 60, 80, 100, 400, 600, 800, 1000, and 10000 sides. It can even handle other needed sides by treating certain sides as “blank” and rolling that die again (so it can roll a d11 by rolling a d12 and then treating the 12 as a blank and re-rolling if it shows up). This will create any value between 2 and 10000 that you may need. These are specified as “`d#`” (where # is the number, e.g. “`d6`”, “`d10`”)

Zero based Dice

Besides dice going from 1 through a given value N, dice that show values from 0 through N-1. These are useful in some places because it allows you to add a bunch of dice and not actually increase the minimum value that they will show. So rolling 10 six sided dice will result in a minimum of 10 (all ones), but 10 zero based six sided dice will still have a possible value of zero - it is just unlikely. A zero based die is specified by using a "`z`" instead of a "`d`" (e.g. "`z6`", "z10").

Special Dice

There are a handful of special dice that are used commonly enough to warrant having built in support. Note that not all have zero based forms:

Code	Color
d6a	"Averaging" dice sometimes used in war-games. The values for 1 and 6 are re-labeled as another 3 and 4 to increase the chance to get average rolls
d%, z%	Percentile dice, generating 1..100/0..99
d‰, z‰	Permille dice, generating 1..1000/0..999
d‱, z‱	Per-ten thousand dice, generating 1..10000/0..9999
d0	Single units digit (identical to z10)
d00	Tens digit (00, 10, 20 .. 90)
d000	Hundreds digit (000, 100, 200 .. 900)
d0000	Thousands digit (0000, 1000, 2000, .. 9000)
dF	FUDGE/Fate:Core specific dice - generates -1, 0, or 1 uniformly (with a minus, plus, or blank face)

Custom Numeric Dice

Additional dice can be created with arbitrary values shown on the faces (or even images or icons). These are referred to via their name. For example, there is a built in set of dice that show roman numerals instead of digits, so "`dRom6`" is a six sided die showing I .. VI. Note that these don't support a zero based version.

Composite Dice

For some game systems (e.g., In Nominae), six sided dice are treated as individual digits in a value (so you can roll from 11 to 66). Percentage dice are similarly - multiple dice are treated as a single value. These are called *composite dice* and The Dicenomicon supports custom composite dice. Up to four dice can be specified, and then a formula exists to combine the

results of those four dice into a final value. For the first example, that formula would be "`#1 x 10 + #2`".

Composite dice are otherwise treated like custom numeric dice.

Step Dice

There are a handful of games that, rather than rolling more dice, one increases the size of the dice (going from a d4 to a d6 to a d8, etc... eventually adding additional dice). To support this The Dicenomicon has support for *step dice*. Built in support includes EarthDawn (3rd edition), EarthDawn "old" (1st-2nd), and Margaret Weis Productions (i.e., Cortex based games). The number of dice to roll actually indicates the step to use. So, with EarthDawn, "4dED" will roll a d6, while "5dED" rolls a d8 and "14dED" rolls a d10 and d12 and adds them together.

Step dice are otherwise treated like custom numeric dice.

Textual Dice

Finally, there are textual dice. These are dice that have labels on their faces, for example, or other images that don't create a specific numeric value. One can easily image a die that generates the twelve signs in the zodiac (and such a thing is built in), or various dice that can generate hit locations - all based on a textual result. Because the result is textual, open rolls, roll macros, and reducers are not supported (if more than one die is rolled, the result is a list of all the values).

Open Rolls (a.k.a. "Exploding")

Many games have rules that say "roll a d6, and if the result is 6, roll an extra die" - these dice "explode" into another die. This allows a fairly linear result, but leaves the extreme "open" to generate even higher values in rare cases. What is interesting about open ended rolls like this is that theoretically any value can be roll, it is just highly unlikely that you'll roll enough 6s to get there. This is expressed by adding special characters after the "d" (or "z") but before the number of faces on the die, e.g. `3d+6` will roll three six sided dice that are open on the high end.

Besides exploding on the high end, it is not uncommon for dice to explode on the low end as well - get a one, and you start subtracting additional rolls (the actual math gets confusing), or open on both ends.

There is a quirk in most open ended dice, however. For example, with an open ended d6, there is no way to get a value of exactly a 6 - if you roll a 6 you will roll another die and get at least a 1, and 6 + 1 = 7. To deal with this, some games include the ability to make this uniform by subtracting 1 for every extra die, so that roll of 6 and 1 becomes 6 + 1 - 1 = 6.

The “high” and “low” threshold are defined as being 1 and N (for a dN dice) for all dice up to a d20. Above that, the top/bottom 5% are used (it is extremely rare for a d% open ended system to use just 100% or 1% - normally it is 96-100 and 1-5). A roll macro can be made specifying alternate values.

Code	Style
+	Open High
-	Open Low
±, +-	Open High And Low
*	Roll another (independent) die if the high threshold is hit - usually used for “dice pool” based rolls
.+	Open High (uniform)
.-	Open Low (uniform)
.±, .+-	Open High And Low (uniform)
/+	Open High, but only add subsequent dice if more than half the highest value (Unisystem)
/-	Unisystem Style Open Low
/±, /+-	Unisystem Style Open High/Low

Roll Macros

Roll macros are a way to look at a single die’s value and act upon it. A roll macro is composed of two parts - the logic used while rolling the die, and the logic used while figuring out what the value is. The first part results in an action for the die - for example, the die can automatically re-roll, or play a sound, add a “badge” to the die, or tint/highlight it.

The second half is used when figuring out the actual value of the result. Note that roll macros can simulate open dice rolls, plus do many other things.

One extra thing that roll macros can do is support a “target number”. Suppose you want to count all dice over a certain values as “1” and those under as “0” (a fairly simple “target number dice pool” roll). Roll macros can take that target number as a constant - it is specified as part of the roll macro name (and this value can, like other constant values, be a parameter).

A variety of roll macros are built in, and additional can be created via The Dicenomicon. In the table below, a # at the end of the code indicates that it takes a target number

Code	Name
TN#	Targeted
XX#	Targeted, high = +2
MIN#	Treats all rolls less than this as this value
MAX#	Treats all rolls greater than this as this value
BOD	Body damage for Champions
BRU#	Brutal dice - roll again if less than target
REO#	Re-roll once if less than the target
ARS	Ars Magica Stress Roll
MEGS	MEGS Open on doubles of a percentile die
HGR	Heavy Gear “half as zero”
HERO	Macro for DC Heroes dice
OPN#	Extra Open
OPH#	Extra High Open
CAP#	Damage Cap (treat values above the cap as zero)

Roll macros are specified after the number of faces (or name of a custom die). In some cases, if the number of faces is the same as the default die size as for the macro, the number of faces can be omitted, and a target number can also be omitted if it matches the default target for the macro (the editor in The Dicenomicon will do this automatically for you). For example, `TN` is designed for d10 dice pools in WoD, so instead of `5d10TN7`, you can write `5dTN` instead.

Reducers

Once a bunch of dice have been rolled, how do we convert that collection to a single value? The most common way is to take the sum of all the dice (for numeric dice). We’ve also see the `@` reducer which leaves the values as a list. There are a couple of other ways that we can combine dice values, again, most with a target number associated with them (unless otherwise specified, there is a target number):

Code	Name
	Default to sum of all dice value (no target)
H, HI	The N highest dice are kept
L, LO	The N lowest dice are kept

Code	Name
#	The Nth dice is used (e.g., `3d6#2` is the middle die)
>, ≥, =, ≠, <, ≤	Count how many dice are greater than, greater than or equal to, equal to, not equal to, less than, less than or equal to the value of N
X	Exalted specific success total
@	Keep results as a list (no target number)

Reducers are written at the end of the dice specification. It is important to remember that reducers act after roll macros, and based one the values that roll macros produce (which may not match what the dice show).

Also note that roll definitions are all parsed as a single value, so it is not uncommon to run into problems using the comparison operator in an if expression with a roll. For example:

```
1.if 3d6 > 10 then
2.  “Success”
3.else
4.  “Failure”
5.end
```

will always return “failure” because that if expression is actually testing “roll 3 six sided dice and count how many dice are showing a 10 or more” (which, of course, is impossible, so the result is always zero). Instead you need to do:

```
1.if (3d6) > 10 then
2.  “Success”
3.else
4.  “Failure”
5.end
```

Putting it together

Not all dice support all options, but they all use the same order of components:

- Number of dice (omitted for 1 die, can be a parameter)
- Color of the die (omitted will use a default color based on user preference or built in default based on size of dice)
- Size of Dice (omitted is normal, format subject to change)

- Sides:
  - Text Dice: `t` name of dice
    - No further specifications possible
  - Numeric Dice: `d` or `z`
    - Open/Exploding specifier (optional)
    - Sides:
      - 1 .. N numeric: `#` (can be parameter, had a d)
      - 0 .. N-1 numeric: `#` (can be parameter, had a `z`)
      - Special dice (usually with a `d`, some support `z`)
      - Custom, composite, step dice (have a name instead of a number of the sides)
    - Roll Macro (optional name, with optional target number for some macros. Target number can be parameter)
    - Reducer (optional name, target number for some reducers. Target number can be parameter)

Mixed Dice Pools

One new feature in ELDARScript 2.0 is the ability to roll different sided dice together and treat them as if they were all the same. For example, some games have you roll a bunch of different dice and take the highest N dice, such as "roll d4, 2d6, d8 and take the two highest". This can be done with a special "collection parenthesis". Individual dice are grouped together inside a `$()`. For example: `$(d4, d6, d6, d8)H2` will roll the roll described above. Only numeric dice are allowed in a mixed dice pool (though colors, open rolls and even roll macros can be specified for individual dice if desired).

Mixed dice pools are defined as the pool specifier "`$(`*dice*...`)`" followed by optional reducer. Roll macros are specified in each individual element of the dice pool as desired.

Advanced Operations

ELDARScript provides some advance string capabilities, which include the ability to make random text generation possible, as well as formatting commands.

String Formatting

When displaying text as the results of a roll, the UI support special "rich text" formatting commands. This include the ability to change the color of the text, size, font, and even embed images, icons, and play sounds. All of this is based on "escape" code, which all being with a backwards slash. Some of them are then single letters, others are "paired" - sequences that being and end with the same punctuation character (with longer information between the pairs). Formatting is cumulative - making something bold and then red results in making it bold and red, or increasing the font size and decreasing it results in the original font size. Note that some combinations don't appear - for example certain fonts don't have bold and italic variations. Also note that the editor in The Dicenomicon + 1 support easy access to these escape codes while editing a string.

Escape Code	Purpose	Example	Result
\n	Inserts return	"First Line\nSecond Line"	First Line Second Line
\p	Resets formatting	"Regular, \eBold, \RBold Red, \pPlain"	Regular, **Bold**, **Bold Red**, Plain
\e	Emphasized (bold)	"Regular, \eBold"	Regular, **Bold**
\i	Italic	"Regular, \iItalic"	Regular, *Italic*
_	Underline	"Regular, _Underline"	Regular, Underline
\r, \g, \b, \c, \m, \y, \k, \w	Set the color to red, green, blue, cyan, magenta, yellow, black, or white	"\rRed, \gGreen, \bBlue, \cCyan, \mMagenta, \yYellow, \kBlack, \wWhite"	Red, Green, Blue, Cyan, Magenta, Yellow, Black, White
\+, \-	Increase/Decrease font size	"Medium, \+Larger, \-Medium, \-Smaller"	Medium, Larger, Medium, Smaller
\=size=	Changefont size	"Medium, \=18=Larger, \=10=Smaller"	Medium, Larger, Smaller

Escape Code	Purpose	Example	Result
\!name!	Change the font name (assuming a font of the given font family name exists - these include AmericanTypewriter, AppleGothic, ArialMT, Courier, Georgia, Helvetica, MarkerFelt-Thin, TimesNewRomanPSMT, TrebuchetMS, Verdana, Zapfino)	“Regular, \!Georgia! Georgia, \!Zapfino! Zapfino”	Regular, Georgia, Zapfino
\#hex#	Change to HTML style color	“Default, \#800000#Brown, \#888#Gray”	Default, Brown, Gray
\$name$	Plays the name of a given sound the first time it is “printed” as a result. This is not applicable for things like the name of a favorite roll (where the string isn’t a result)	“Missed!$Whoosh$”	Missed *(The sound 'Whoosh' would be played’)*
\uXXXX	A special 5 character escape code, used to specify unicode characters (where XXXX is for hex characters).	“Black Pawn: \u265f Black Star: \u2605”	Black Pawn: ♟ Black Star: ★
\`inline data`	Displays an inline PNG image, where the data is base64 encoded between the back ticks	\`iVBORw0KGgoAAA ANSUhEUgAAAAUAAAA FCAYAAACNbyblAAAAH ElEQVQI12P4 //8/ w38GIAXDIBKE0DHxgljN BAAO9TXL0Y4OHwAAA ABJRU 5ErkJggg==`	
\&path&	Displays a filled shape whose path is specified by SVG path elements	\& M 10 25 L 10 75 L 60 75 L 10 25&	
\:badge:	Displays a specified badge image	\:plus:	
\~lorc~	Displays an inline LORC element, specified in the following format: name[:fg-color[+bg-color]][@size] (where the square brackets indicate optional values)		
x,y,w,h	Used only when specifying the face of a custom die, Causes the text to be placed in a grid that is w x h large at x, y	“*0,0,3,3*A*1,1,3,3*B”	

Escape Code	Purpose	Example	Result
\@global,style=,max=,min=,color=@	embeds an global variable “adjuster”, where global is the value to control; style is either adjust, circle, or check; max is the maximum value (default = current value); min is the minimum value (default = 0)	\@HP,style=adjust,min=0,max=10@	

Text Generation

Via the `%` operator with a string and a list, strings can be formatted to include other values - a process called string interpolation where each `%` inside the string corresponds to a list element:

Expression	Value	Notes
`“First % second %” % (1, 2)`	“First 1 second 2”	Each element is substituted in turn
`“Second %2 first %1” % (1, 2)`	“Second 2 first 1"	By using a number after the percent, that specific element is used
`“Start % skip %! next %” % (1, 2, 3)`	“Start 1 skip next 3”	An exclamation point causes that value to not be displayed at all
`“Percent %% first %” % (1, 2, 3)`	“Percent % first 1”	Using a double percent we get a single percent (and no substitution)
`“Plain %, Title %^ Capital %^^ Upper %^^^” % (“first word”, “second word”, “third word”)`	“Plain first word, Title Second word, Capital Third Word, Upper FOURTH WORD”	By adding a ^ after the %, the first letter of the substituted value will be converted to upper case. Two ^^ will capitalize the first letter of each word. Three ^^^ will capitalize all the letters
`“Title %~ Lower %~~” % (“FIRST WORD”, “SECOND WORD”, “THIRD WORD”, “FOURTH WORD”)`	“Title fIRST WORD Lower second word”	Adding a single ~ converts the first letter to lower case, while two ~~ converts all the letters to lower case
`“Value %#%” % (1, “pie”)` `“Value %#%” % (2, “pie”)`	“Value a pie” “Value 2 pies”	A %# indicates that this value is a count of how many of the next item there are, and it attempts to form a reasonable singular or plural of the word of the next % (i.e., this works across two substitutions)

Expression	Value	Notes
`@X ← 5.` `"X is %@X" % ""`	“X is 5”	Using a variable name with the leading @, we will display that variable’s value. Remember, this only happens with the % operator, so we have a dummy empty string on the right hand side
`"%@X← plus one is %(@X + 1)" % (5)`	“5 plus one is 6”	Besides displaying a variable, we can use a special operation to assign the value from the list to a local variable (@X←5). Note that if we wanted to suppress showing that value, we’d use the ! modifier. Then, by placing an expression inside parenthesis, we evaluate that (instead of getting the next value from the right hand side of the %).
`"%(d8)" % ""`	“3”	Note that even dice can be used inside the expression (however, no dice are actually rolled here)
`“I see%#(d4-1) %” % “sword”`	“I see no swords” “I see a sword” “I see 2 swords”	We can combine the # format (to indicate count) with an expression to make random amounts
`“I see a %(generate('Flavor:Color')) %” % “gem”`	“I see a Blue gem”	Calls to random text generators can be made, like any other ELDARScript function in an expression…
`“I see a %[Flavor:Color] %” % “gem”`	“I see a Red gem”	… but this is common enough to have a shortcut by enclosing the `generate()` parameter inside `[]`

If one of the right hand values is actually a frame instead of a simple string or number, we can create sophisticated text generation based on the fields of that frame. Again, this only happens when we use the `%` operator to create a string with formatting characters or if we use the built in function `string()`. In all of these examples, we will create a table and then display it, altering the table to show various options (and show various possible outputs)

Expression	Value	Notes

`@X ← {`
`“:table:”: "Hello"`
`}.`
`string(@X)`
	“Hello”	The magic is that the frame has an slot called “:table:” which will be used to generate the text. In this case, we just display that string

Expression	Value	Notes

`@X ← {`
`  ":table":`
`("red","green")`
`}.`
`string(@X)`
	"red"
	"green"
		If the value of ":table:" is a list, one element will randomly be picked from it

`@Y ← 5.`
`@X ← {`
`  ":table":"Y is %@Y"`
`}.`
`string(@X)`
	"Y is 5"	By having escape values inside the ":table:" string, we can evaluate that as if it were actually `"Y is %@Y" % ""` (i.e., at run time we treat it as if we were formatting it vs an empty value)

`@X ← {`
`  ":table":"X is %@X"`
`}.`
`string(@X)`
	"X is X is X is X is…"	You can refer to itself, which is normally bad (though it will eventually limit you to how deeply it will nest itself)

`@Y ← {`
`  ":table":`
`("red","green")`
`}.`
`@X ← {`
`  ":table":"X is %@Y"`
`}.`
`string(@X)`
	"X is red"
	"X is green"
		You can refer to a second frame with a ":table:" key, and that gets randomly generated, so you can make multiple random text tables and refer to each other

`@X ← {`
`  ":table":"X is %:color"`
`| "color":("red",`
`"green")`
`}.`
`string(@X)`
	"X is red"
	"X is green"
		It is usually easier to use the : shortcut and refer to another slot in the current frame, which then has random text generation applied to it.

`@X ← {`
`  ":table":"The %:animal`
`%:verb %:animal"`
`| "animal":"%:adj %:noun"`
`| "adj":("quick", "lazy",`
`"large", "small")`
`| "noun":("dog", "cat",`
`"fox", "bear")`
`| "verb":("jumped over",`
`"saw", "ignored")`
`}.`
`string(@X)`
	"The lazy dog jumped over the quick fox"	We can use multiple tables, which use other tables, etc… In this case we have `@X` whose `:table:` slot refers to "`%:animal`" and "`%:verb`". "`animal`" generates using "`%:adj`" and "`%:noun`".

`@X ← {`
`  ":table":("red",`
`"green")`
`| ":tprefix":"<"`
`| ":tsuffix":">"`
`}.`
`string(@X)`
	"<red>"
	"<green>"
		By using the special `:tprefix:` and `:tsuffix:` slots, we can specify a string for a prefix/suffix of the resulting table generation

Expression	Value	Notes

`@X ← {`
` 1…6:“Common”`
`| 7…9:“Uncommon”`
`| 10:“Rare”`
`}.`
`string(@X)`
	“Common”
	“Uncommon”
		If a frame has numeric slots, it will automatically be treated as if there was a `:table:` slot which will pick one at random. Note that the value should range from 1 upwards

`@X ← {`
` “:table:”:6`
`| 1…6:“Common”`
`| 7…9:“Uncommon”`
`| 10:“Rare”`
`}.`
`string(@X)`
	"Common” (always)	If for some reason, you want to limit the maximum index value, you can do so by specifying a numeric value for the `:table:` slot.

`@X ← {`
` “:tableroll:”:"2d5"`
`| 1…6:“Common”`
`| 7…9:“Uncommon”`
`| 10:“Rare”`
`}.`
`string(@X)`
	“Common” (15 in 25)
	“Uncommon” (9 in 25)
	“Rare” (1 in 25)
		If the index for a table with numeric slots needs to be something other than a linear random pick, we can use the `:tableroll:` slot to indicate an expression to evaluate. It is important to put the expression in quotes, or else it will get evaluated when the table is created, not when it is formatted

`@X ← {`
` “:tableroll:”:“@Y <-`
`2d5”`
`| 1…6:“Common %:color”`
`| 7…9:“Uncommon”`
`| 10:“Rare”`
`| “color”:(“red %Y”,`
`“green”)`
`}.`
`string(@X), @Y`
	“Common red 5, 0”	Variables that are set in the process of evaluating a table are available to other parts of that table (or things called from there) but not “outward”. So `@Y` has a value inside `color`, but not in the ultimate formatting call

Inheritance

Frames can “inherit” values from other frames - this is used a great deal internally, but rarely will you need to use it explicitly. There are two special slot names used for this:

Slot	Used for
:proto:	Providing an abstract frame that this is based on at creation time
:parent:	Providing an “enclosing” frame that this is contained in during execution

When asking for a value of a frame, if the frame doesn't contain one with that slot name, it will first look in the `:proto:` slot's frame (if any) for that key. If there is nothing there, it will look in the `:parent:` slot (if any).

To give an example of how this would work, consider a group of orcs. All orcs are basically the same, though they may have different HPs. So each orc would be represented by an frame that has a `"CURHP"` slot with the current health, but all would have a `:proto:` that refers to the same frame (providing all the basic information such as AC, attacks, etc..). This is classical "prototype inheritance" where instance is based on another (this is the model for JavaScript objects as well as NewtonScript).

`:parent:` based more on runtime nesting of object. An example of this would in a character sheet - a row in the "attacks" section of the sheet would have the character sheet data as a whole as its `:parent:`. Or for some sort of encounter, a room would have the current "zone" as its :`:parent:` and that zone would have the level as its `:parent:` and that would have the module as its `:parent:`.

As an example of where this is used internally, consider the following:

```
1.@W ← {
2.  ":table:"  :  "Got %:item"
3. | "item":{
4.    1…7:"%:material"
5.  | 8…10:"%:color %:material"
6. }
7. | "color":{
8.    1…6:"Red"
9.  | 7…9:"Blue"
10. }
11. | "material":{
12.    1…4:"Silk"
13.  | 6…10:"Cotton"
14. }
15.}.
16.string(@W)
```

Most of the time it will spit out "Silk" or "Cotton", but sometimes it will also apply a color to it. Note that references to the color and material table are actually made from within a table in lines 3 through 6. In order for that inner frame to find `color` and `material`, the runtime actually creates a new object whose `:proto:` is the frame on line 3 and whose `:parent:` is the whole frame on lines 1 through 15. This allows the text generation

system to be able to generate the first table (which it finds through `:proto:` inheritance) and access color and material (via `:parent:` inheritance).

See "*Libraries and Extensions*" for more details about how to define generators.

Units

ELDARScript and The Dicenomicon include support for working with items that are expressed in units (this has changed and been improved significantly in 2.0). One prime example of where this is useful is with keeping track of money - you can't just have a single value, since there will be gold, silver, copper, etc... What's more, you can convert from one form to the other (i.e., "making change"). To make matters even worse, in many campaigns, there is different currency in different locations.

To deal with this, we have the ability to mark a frame as being a scalar unit. These units are defined inside XML configuration files , which define:

- Family of the unit (money, time, distance, etc...)
- Name of unit and abbreviation ("Common gold/gp", etc...)
- Each unit has a conversion ratio to a standard unit ("20 silver to 1 gold")

When a frame has a slot named "`:unit:`", this is the name of family of the unit. Every unit that is a member of that family can have a corresponding slot. For example:

```
1.@wealth <- {
2.   ":unit:" : "money"
3.| "gold" : 3
4.| "silver" : 10
5.}
```

Much like the "`:table:`" key, having this "`:unit:`" key invokes special formatting to display this showing the abbreviated form of the units (in this example, it would format as "`3 gp, 10 sp`"). There are also functions to convert from one unit to another, and in general, make change (by converting rational values to integer where possible).

See "*Libraries and Extensions*" for more details about how to define units.

Formal Syntax

This is the formal definition for ELDARScript syntax in EBNF. Note that this also describes some semantically incorrect programs, even though they are syntactically correct, as well as some ambiguous conditions that are also resolved based on semantics. Not all of this grammar is available in all places (for example, a function definition can not currently specify rolling dice).

```
RollExpr ::= ConditionExpr ( (',' ConditionExpr) | ('.'
ConditionExpr))*
ConditionExpr ::= Assignment | 'if' AndOrExpr 'then' RollExpr
('elif' AndOrExpr 'then' RollExpr)* 'else' RollExpr 'end' | 'for'
Variable 'in' RollExpr 'do' RollExpr 'end' | AndOrExpr
Assignment ::= (Variable | Parameter) '←' ConditionExpr
AndOrExpr ::= Comparison ( ('^' | 'v' | '»') Comparison)*
Comparison ::= RollRange ( ('<' | '≤' | '=' | '≠' | '>' | '≥' | 'in'
) parseRollRange)*
RollRange ::= RollTerm ( '…' RollTerm ) *
RollTerm ::= RollMult (('+' | '-') RollMult ) *
RollMult ::= RollSubscript (('×' | '%' | '/+' | '/-' | '/=' | '÷' |
'/' ) RollSubscript)*
RollSubscript ::= FunctionCallExpr ( '[' RollExpr ']' ('←'
ConditionExpr)? | ':' Identifier  ('←' ConditionExpr)? )*
FunctionCallExpr ::=  FunctionName ( Number )? List |  ('×'
SimpleValue ':')? SimpleValue
SimpleValue ::= String | Dice
Dice ::= Frame | '(' ')' | '(' RollExpr ')' | 'true' | 'false' |
Variable ( List )? | Identifier | Parameter | Number |
(RollReference | DieCount? DieAppearance Macro? | '$(' DieCount?
DieAppearance Macro? (',' DieCount? DieAppearance Macro?)* ')' )
Reducer? | DieCount? TextDieAppearance
DieCount ::= Parameter | Number ('½')?
DieColor ::= [rgbcmykwaon]
DieAppearance ::= DieColor? [dz] Open? (Identifier (Number) | Number
| Parameter | '%' | '‰' | '‱' | '6a' )?
TextDieAppearance ::= DieColor? [t] Identifier (Number)
Open ::= '+' | '-' | '*' | '±' | '.+' |'.-' |'.±'
Macro ::= Identifier (Number | Parameter)?
```

```
RollReference ::= [$][1-9][0-9]?
Variable::= [@] [A-Z][A-Za-z0-9]?
Parameter ::= [#] [1-9][0-9]?
Frame ::= '{' ( List )? ( Slot ('|' Slot )* )? '}'
Slot ::= SlotName ':' ConditionExpr
SlotName ::= ( String | Identifier | Number ('…' Number )? ) (','
SlotName )*
List ::= '(' ( ConditionExpr ( ',' ConditionExpr )* )? ')'
```