DiceCode Programming Language

Pre-build ISO
2nd Edition

Jan 17, 2020

Table of Contents

Introduction....4
  Conventions....4
Supported Characters....5
  Symbols....5
  Alphanumeric....5
  Special Characters....5
Syntax....6
  Line....6
  Variable Types....6
    void....6
    int....6
    frac....6
    roll....6
      Dies....6
      Special Rules for Rolls....7
    Arrays....8
    Declaration and Initialization....8
  Built-in Functionalities....8
    Unary Operations....8
      Casting....8
      Logical Negation....9
      Negation, Increment and Decrement....9
    Binary Operations....9
      Assignment....9
      Addition....10
      Subtraction....10
      Multiplication....11
      Division....11
      Remainder....12
      Logical....12
    Ternary Operations....14
      Ternary Conditional....14
    Sections....14
      Scopes....14
      Comments and Documentation....14
    Constants....15
    Functions....15
      Conditional....15

Loops....15
Functions....15
Built-in Functions....16
Order of Operations....18
Preprocessing....19
Functions....19
Debugging....20
Run-time Exceptions....20
Interpreter Exceptions....21
Sample Program....22

Introduction

DiceCode is a programming language made to facilitate analysis of probability distributions, specially when related to dice rolls and games, using linguistic elements common to many tabletop games.

Conventions

In this document, text in bold between lesser-than, greater-than signs (<>) is read as an expression.

<expression>

Reference to names in text is explicit by italics.

Type, expression or variable reference

Supported Characters

Symbols

+ - * / % ( ) [ ] { } < > ? ! = . , ; : # & | *(space character)*

Alphanumeric

[a, z], [A, Z], [0, 9]

Special Characters

\n \t

Syntax

Line

A line contains one or more expressions and is ended by a semi-colon (;). The interpreter ignores almost all spaces ( ), all tab (\t) and all line-break (\n) characters, so it requires semi-colons to locate code lines.

<expressions>;

Variable Types

void

Represents nothing, or *null*. Used for valueless expressions or functions that do not return anything.

int

Represents integer numbers in the range [-(2^31), 2^31-1].

frac

Represents rational numbers in the range [-(2^31), 2^31-1] with 2^(-32) precision. Each *frac* is composed of two *ints*, a nominator and an unsigned denominator. Fractions are always displayed in their simplified form. If denominator is 0, *frac* evaluates to *infinity* or *neg_infinity*.

roll

Roll variables. A roll is an expression that represents a random event, more specifically a dice roll.

Dies

Dies are written in d-notation, which consists of two values separated by the letter 'd'. The value of the first expression is the amount of dies, and the value of the second expression is the amount of faces of those dies.

Rolls can also have modifiers. These modifiers can be composite of any expression that evaluates to a *roll* or *int*.

<quantity>d<value> <modifiers>

Example:

roll simple = 1d4; //simple roll of a 4-sided dice.

roll mult = 2d6; //simple roll of 2 6-sided dies.

roll mod = 1d20 + 1; //roll with simple modifier.

roll mult_mod = 1d4 + 1d6 + 1d8; //roll with two modifiers, each being a roll.

roll complex_mod = 1d(2d8) + (1d8 > 4? 4: 1d4);

/ Roll with complicated modifiers.*

When evaluated the first dice needs its faces evaluated into an int to determine how many faces such dice has.

The roll's modifier is a ternary conditional, so it is evaluated to one of its expressions.

**/*

<amount>d<faces>l<filter> – This roll is composed of *amount* dies of *faces*-sided dies, but only the top *filter* are considered. For example, let a roll of 4d6 have the result {2, 3, 4, 4}, or 13 in total. If that roll had been 4d6l3 instead, the result would only consider the top 3 results, so it would have been {3, 4, 4} instead, having a total value of 11.

<amount>d<faces>s<filter> – This roll is composed of *amount* dies of *faces*-sided dies, but only the bottom *filter* are considered. In the example above, if the roll had been 4d6s3 then the result would have been {2, 3, 4}, or 9 in total.

Special Rules for Rolls

Rolls are a unique type because they are not resolved to values. The entire expression that the roll is initialized or assigned to is considered the value of the roll. In this manner, compared to other programming languages, a roll is somewhere between a string and a function, albeit with a few limitations.

Example:

int deeFourDeterm() { return 1d4; }

roll deeFourRoll() { return 1d4; }

roll roll_one = 2d4;

roll roll_two = 1d4 + deeFourDeterm();

roll roll_three = 1d4 + deeFourRoll();

// Even though all three expressions look the same, when rolls roll_one and roll_three are analyzed by one of the built-in functions all dies are considered and cycled through. In the case of roll_two, the modifier is an int, so it is evaluated before analysis, which means that the analysis will have up to four different results depending on the specific result the function returned in that run.

*int squareRoll(roll input) { int eval = input; return eval*eval; }*

*int squareInt(int input) { return input*input; }*

roll roll_0 = squareRoll(1d6);

roll roll_1 = squareInt(1d6);

// In this particular case, even though both rolls are written differently, they behave the same way. When analyzed, the argument will be cycled through in both cases, even though the roll looks like a set int. When the roll is analyzed, the analysis tool is looking for its dies, wherever they might be, however they might appear.

Arrays

<type>[] - Array variable of type *type*. Arrays have variable size and can have multiple dimensions.

Declaration and Initialization

<type> <name>; – Variable declaration.

<type>[] <name>; – Array declaration. Declares a new empty array of type *type*.

Built-in Functionalities

Unary Operations

Casting

(frac) <int> – The resulting *frac* expression has the original *int* as its nominator and 1 as its denominator.

(int) <frac> – The resulting *int* expression is equal to the *frac*'s nominator divided by its denominator.

(int) <roll> – The resulting *int* expression is a possible result of the roll. The result is generated following the logic of the roll expression. Casting from *roll* to *int* can be implicit.

(roll) <int> – The resulting *roll* expression is composed of only a modifier.

(<target_type>[]) <origin_type>[] – The resulting array is an element-wise cast of the first array into *target_type*.

Logical Negation

!<int> – The resulting expression is equal to 1 if the *int* expression is 0 and 0 otherwise. Applied element-wise for arrays.

Negation, Increment and Decrement

-<expression> – The resulting expression is equal to the negative of *expression*. If *expression* is an array, then the negation is applied element-wise.

<variable>++ – Adds 1 to the value of *variable* in the order *variable*+1. Applied element-wise for arrays. The resulting expression is equal to the new value of *variable.*

++<variable> – Adds 1 to the value of *variable* in the order 1+*variable*. Applied element-wise for arrays. The resulting expression is equal to the new value of *variable.*

<variable>-- – Subtracts 1 to the value of *variable* in the order *variable*-1. Applied element-wise for arrays. The resulting expression is equal to the new value of *variable.*

--<variable> – Subtracts 1 to the value of *variable* in the order -1+*variable*. Applied element-wise for arrays. The resulting expression is equal to the new value of *variable.*

Binary Operations

Assignment

<type> <var> = <expression>; – Declares the variable *var* of type *type* and initializes it to it the value of *expression*.

<var> = <expression> – Assigns the value of *expression* to the variable *var*. The resulting expression is of the same type and value as *var*.

<var> <op>= <expression> – Assigns the value of the operation *op* between *var* and *expression* to *var.* The resulting expression is equal to the new value of *var*.

Addition

<int>+<int> – The resulting expression is an *int* expression whose value is equal to the sum of the two *int* expressions.

<int>+<frac> *or* <frac>+<int> - The resulting expression is a *frac* whose value is equal to the sum of the cast of the *int* expression into a *frac* expression and the original *frac* expression.

<frac>+<frac> - The resulting expression is a *frac* whose value is the sum of the two *frac* expressions.

<int>+<roll> - The resulting expression is an *int* expression whose value is equal to the sum of the original *int* expression and a cast from the *roll* expression into *int*.

<roll>+<int> *or* <roll>+<roll> - The resulting expression is a *roll*.

<type[]>+<type[]> – The resulting expression is an element-wise sum of the two arrays. If either array is larger, the resulting expression will have the same size as the largest array.

Subtraction

<int>-<int> – The resulting expression is an *int* expression whose value is equal to the difference of the two *int* expressions.

<int>-<frac> *or* <frac>-<int> – The resulting expression is a *frac* whose value is equal to the difference of the cast of the *int* expression into a *frac* expression and the original *frac* expression.

<frac>-<frac> – The resulting expression is a *frac* whose value is the difference of the two *frac* expressions.

<int>-<roll> – The resulting expression is an *int* expression whose value is equal to the difference of the original *int* expression and a cast from the *roll* expression into *int*.

<roll>-<int> *or* <roll>-<roll> – The resulting expression is a *roll*.

<type[]>-<type[]> – The resulting expression is an element-wise subtraction of the two arrays. If either array is larger, the resulting expression will have the same size as the smaller array.

Multiplication

<int>*<int> – The resulting expression is an *int* expression whose value is equal to the product of the two *int* expressions.

<int>*<frac> or <frac>*<int> – The resulting expression is a frac whose value is equal to the product of the cast of the int expression into a frac expression and the original frac expression.

<frac>*<frac> – The resulting expression is a *frac* whose value is the product of the two *frac* expressions.

<int>*<roll> – The resulting expression is an *int* expression whose value is equal to the product of the original *int* expression and a cast from the *roll* expression into *int*.

<roll>*<int> *or* <roll>*<roll> – The resulting expression is a *roll*.

<type[]>*<type[]> – The resulting expression is an element-wise product of the two arrays. If either array is larger, the resulting expression will have the same size as the smaller array.

Division

<int>/<int> – The resulting expression is a *frac* expression whose nominator is the first *int* expression and whose denominator is the unsigned version of the second *int* expression. If the denominator is negative, the nominator is multiplied by -1.

<int>/<frac> – The resulting expression is a *frac* whose value is equal to the fraction of the cast of the *int* expression into a *frac* expression over the original *frac* expression.

<frac>/<int> – The resulting expression is a *frac* whose value is equal to the fraction of the original *frac* expression over the cast of the *int* expression into a *frac* expression.

<frac>/<frac> – The resulting expression is a *frac* whose value is the division of the two *frac* expressions.

<int>/<roll> – The resulting expression is a *frac* expression whose value is equal to the fraction of the original *int* expression and a cast from the *roll* expression into *int*.

<roll>/<int> or <roll>/<roll> – Division. The resulting expression is a *frac* expression whose nominator and denominators are implicit casts from *roll* into *int* expressions.

<int[]>/<int[]> – The resulting expression is an array of *frac* whose nominators are each element of the first array and whose denominators are each element of the second array. If either array is larger, then the resulting array will have the same size as the smaller array.

Remainder

<int>%<int> – The resulting expression is an *int* expression whose value is equal to the remainder of the division of the first expression by the second expression.

Logical

<expression_one>==<expression_two> – The resulting expression is an *int* expression whose value is equal to 1 if the expressions have the same value or 0 if the expressions have different values.

If the type of any of the expressions is *roll*, it is evaluated to an *int* expression before the comparison.

When used on two array variables of the same type, the resulting expression is equal to 1 if every element of the two arrays are also equal in value.

<expression_one>===<expression_two> – The resulting expression is an *int* expression whose value is equal to 1 if the expressions are identical or 0 if the expressions are not. *roll* expressions are compared bit-wise, so even if they are functionally the same they will be considered different if they are worded differently.

Example:

```
int equals = (1d4 + 1d6) === (1d6 + 1d4); //equals is 0.

equals = 1d6 === 1d6; //equals is 1.
```

When used in array variables, the resulting expression will be equal to 1 if and only if the two array variable are two different references to the same array.

Example:

```
int[] a = {1, 2, 3};

int[] b = a;

int[] c = {1, 2, 3};

int equals = a === b; //equals is 1

equals = a === c; //equals is 0

int compare = a == c; //compare is 1
```

<expression_one> != <expression_two> – The resulting expression is an *int* expression whose value is equal to 1 if the expressions have the same value or 0 if the expressions have different values.

If the type of any of the expressions is *roll*, it is evaluated to an *int* expression before the comparison.

When used on two array variables of the same type, the resulting expression is equal to 1 if every element of the two arrays are also equal in value.

<expression_one> > <expression_two> – The resulting expression is equal to 1 if the value of *expression_one* is greater than the value of *expression_two* and equal to 0 otherwise.

If either expression is of type *roll* then it will be evaluated to *int* before the comparison.

If the expressions are both array expressions and the first array is larger than the second, then the resulting expression is equal to 1.

<expression_one> < <expression_two> – The resulting expression is equal to 1 if the value of *expression_one* is lesser than the value of *expression_two* and equal to 0 otherwise.

If either expression is of type *roll* then it will be evaluated to *int* before the comparison.

If the expressions are both array expressions and the first array is smaller than the second, then the resulting expression is equal to 1.

<expression_one> >= <expression_two> – The resulting expression is equal to 1 if the value of *expression_one* is greater than or equal to the value of *expression_two* and equal to 0 otherwise.

If either expression is of type *roll* then it will be evaluated to *int* before the comparison.

If the expressions are both array expressions and the first array is larger than or the same size as the second, then the resulting expression is equal to 1.

<expression_one> <= <expression_two> – The resulting expression is equal to 1 if the value of *expression_one* is lesser than or equal to the value of *expression_two* and equal to 0 otherwise.

If either expression is of type *roll* then it will be evaluated to *int* before the comparison.

If the expressions are both array expressions and the first array is smaller than or the same size as the second, then the resulting expression is equal to 1.

<expression_one> && <expression_two> – The resulting expression is equal to 1 if the value of neither expression is 0 and 0 otherwise.

<expression_one> || <expression_two> – The resulting expression is equal to 0 if the value of both expressions is 0 and 1 otherwise.

Ternary Operations

Ternary Conditional

<condition>?<if_expression>:<else_expression> – The resulting expression is equal to *else_expression* if *condition* is equal to 0, and *if_expression* otherwise.

Sections

Scopes

(<expression>) – Evaluates the contents of *expression* before the rest of the expression it is inserted in.

<function>(<arguments>) – An argument is an expression whose type must be acceptable by the function, and all arguments are separated by commas within *arguments*.

{<expression>} – Every variable inside *expression* is invisible to the rest of the code. When in front of functions, defines the scope of those functions.

#<expression> – Preprocessing. Code that is processed before the interpreter

Comments and Documentation

//<comment> – Everything written in *comment* up to the end of the line is ignored by the interpreter.

/*<multi-line comment>*/ – Multi-line comment. Everything written in *multi-line comment* is ignored by the interpreter. The beginning (/*) and end (*/) of the comment may be in different lines.

/**<documentation>*/ – Multi-line documentation. Markdown syntax. Every line in *documentation* must begin with an asterisk (*) to be properly interpreted by the Markdown parser. Ignored by the interpreter.

Constants

null – Declared variables that are not initialized, empty arrays and undefined expressions have their value equal to *null*.

infinity – A *frac* whose denominator is equal to 0 and whose nominator is positive is displayed as *infinity*.

neg_infinity – A *frac* whose denominator is equal to 0 and whose nominator is displayed *neg_infinity*.

nan – A *frac* whose both nominator and denominator is equal to 0.

Functions

Conditional

if(<condition>) <if_scope> else <else_scope> – Evaluates *if_scope* if *condition* is not equal to 0, otherwise evaluates *else_scope*.

Loops

while(<condition>) <loop_scope> – Loops through *loop_scope* if *condition* is not equal to 0.

for(<initial_expression>;<condition>;<final_expression>) <loop_scope> – Evaluates *initial_expression*, then if *condition* is not equal to 0, loops through *loop_scope*, and finally evaluates *final_expression*.

break; – Immediately breaks out of the loop scope, if in one.

continue; – Immediately finishes the current loop evaluation, skipping to the next.

Functions

<type> <name>(<arguments>) {<function_scope>} – Defines a scope that may be called by *name* within the program. The function must resolve to a variable of type *type*. Every argument inside *arguments* is a <type> <arg_name> double that defines the type of each argument and the name for the argument that the function will use internally.

return <expression>; – Immediately breaks out of the function's scope, evaluating the function's expression to *expression*.

except <int>; – Immediately breaks out of the program, printing an error code to the console.

Built-in Functions

int mcm(int a, int b) – Returns an *int* equal to the minimum common multiplier of *a* and *b*.

int mcd(int a, int b) – Returns an *int* equal to the maximum common divider of *a* and *b*.

<int/frac> abs(<int/frac> input) – Returns the absolute value of *input*.

<type> min(<expression_one>, <expression_two>) – Returns whichever of the given expressions is smaller.

int min(roll input) – Returns an *int* equal to the minimum possible value of *input*.

<type> min(<type>[] input) – Returns whichever element of *input* is the smallest.

<type> max(<expression_one>, <expression_two>) – Returns whichever of the given expressions is larger.

int max(roll input) – Returns an *int* equal to the maximum possible value of *input*.

<type> max(<type>[] input) – Returns whichever element of *input* is the largest.

int length(<type>[] input) – Returns the size of *input*.

<type>[] sort(<type>[] input) – Sorts and returns *input*.

<type>[] reduce(<type>[] input) – Returns *input* without repeated elements.

frac avg(<int/frac> n_0, <int/frac> n_1) – Returns a *frac* equal to the average of two numbers, (*n_0*+*n_1*)/2.

frac mean(<type>[] input) – Returns a *frac* equal to the mean value of *input*.

frac mean(roll input) – Returns a *frac* equal to the mean value of *input*, the average between its max and min values.

frac median(<type>[] input) – Returns a *frac* equal to the median value of *input*.

frac median(roll input) – Returns a *frac* equal to the median value of *input*.

frac expected(roll input) – Returns a *frac* equal to the expected value of *input*. The expected value of a roll is calculated by the sum of all results weighted by their likelihood.

int[] all(roll input) – Returns an *int* array containing all values *input* takes, sorted.

int[] possible(roll input) – Returns an *int* array containing all values *input* can take. It is functionally equal to *reduce(all(input))*.

int[] each(<roll>) – Returns an array with the result of each dice of an arbitrary generation of *roll*, in order of appearance. Modifiers are applied only to the last dice. The size of this array is not deterministic and depends on the modifiers. *Example: (1d4)d6 can have anywhere between 2 and 5 dies, one d4 and at least 1d6 up to 4d6.*

int[][] every(<roll>) – Returns a 2-dimensional array with every possible result of the dies of *roll*. WARNING: The array has the potential to get very large very quickly. Make sure *roll* does not contain many dies.

frac[] odds(roll input) – Returns a *frac* array containing the odds of *input*, in the order of *possible(input)*.

frac chance(int val, roll gen) – Returns a *frac* equal to the chance that *gen* will generate the value *val*. If *gen* cannot generate *val*, returns 0/1.

<type> read(<type> input) – Returns input from the user via console. Waits for input before continuing.

void print(<expression>) – Prints the value of *expression* to the console, followed by a new line.

<type>[] push(<type>[] array, <type> element) – Adds *element* to the end of *array* and returns the modified array.

<type>[] insert(<type>[] array, <type> element, int index) – Adds *element* to *array* at *index*, pushing elements of higher index forward. Returns the modified array.

<type>[] empty(int n) – Creates an empty array containing *n* null elements.

Order of Operations

Order	Operation	Syntax
1	Expression scope	(<expression>)
2	Casting	(<type>) <expression>
3	Dice	<exp>d<exp><op><exp>
4	Negation	-<expression>
5	Division	<expression>/<expression>
6	Multiplication	<expression>*<expression>
7	Remainder	<expression>%<expression>
8	Subtraction	<expression>-<expression>
9	Addition	<expression>+<expression>
10	Logical negation	!<expression>
11	Logical binary operations	<expression><op><expression>
12	Ternary conditional	<cond>? <exp>: <exp>
13	Assignment	<expression>=<expression>

Preprocessing

Functions

import(<path>) – Imports the source code of the file in *path* into this file. The interpreter will first parse every imported file in order before parsing this file.

define <constant> = <value> – Defines *constant* to a value, such that every occurrence of *constant* in this file is replaced by *valued* before parsing. *constant* must be written entirely in uppercase.

Debugging

Run-time Exceptions

Code	Name	Description
0x000000	Arbitrary Stop	This exception code is an arbitrary stop to be thrown by the user.
0x000001	Array Index out of Bounds	This exception code is thrown if an array is accessed with an index that is either larger or equal to its size, or if the given index is negative.
0x000002	Null Pointer	This exception is thrown if arithmetic is attempted with the null constant.
0x000003	Invalid Cast	This exception is thrown if an expression is cast to a type which is not described. A few examples are 2-dimensional array to a 1-dimensional array and *roll* to *frac*.
0x000004	Type Mismatch	This exception is thrown if an expression is of a different type than it should be. Usually caught by the interpreter.
0x000005	Invalid Dice	This exception is thrown if any of the expressions in a dice is invalid. A dice cannot have a negative number of faces or be filtered through a negative number.

Interpreter Exceptions

Every exception thrown by the interpreter points exactly to where the error was caught, for ease of correction.

Code	Name	Description
0x00000000	Syntax Error	Thrown by the interpreter if any syntactical rule is broken. These rules can be invalid symbol placement, not finishing a line with semi-colons, improper scope closing, etc.
0x000001	Type Mismatch	Thrown by the interpreter if a variable is initialized to a value of improper type, if a function returns a value of improper type or if an expression expected to be of a specific type is of another type.
0x000002	Invalid Path	Thrown by the interpreter in the preprocessing phase. If the path is invalid or does not point to a DiceCode source code file then it is considered invalid.
0x000003	Preprocessing Error	Thrown by the interpreter if there is any syntactical error inside preprocessor code.

Sample Program

```
// This is a program that automates and analyses D&D 5e stats
int[6] stats;
int[6] modifiers;
roll stat_roll = 4d6l3;
roll stat_mod = (int) (4d6l3/2) - 5;
for(int t = 0; t < 6; t++)
{
    stats[t] = stat_roll;
    modifiers[t] = (int) (stats[t]/2) - 5;
}
frac stat_exp = expected(stat_roll);
frac mod_exp = expected(stat_mod);
print(stat_exp);
print(stats);
print(mod_exp);
print(modifiers);
```

Program Output:

```
306/25 (12.24)
[ 10, 12, 13, 13, 14, 13 ]
109/125 (0.87)
[ 0, 1, 1, 1, 2, 1 ]
```

Thanks to Jasper Flick at https://anydice.com/ for providing the analysis for the program output!