**Goo scripting language description:**

1. Adding a script to an object does nothing. In order to use it, first you need to prompt the Goo Interpreter to translate your scripts into events with “build” instruction.
2. All instruction in the first level of the script are translated into events that will be executed every iteration – so called default events. Each if statement in the script creates a conditional/triggerable event – instructions placed inside the scope of these statements are executed only if all conditions in the if statement are fulfilled. Nested if and else statements, while also being translated into different events, are connected with each other and will be executed in the order given in the script – using interrupts if needed.

Using “**event [name]()”** statement you can create dormant events with a name set to a given string. Dormant events won’t be executed in every loop nor in a reaction to normal triggers. They are an equivalent of functions from normal programming languages and you can give them parameters by putting them in the parenthesis. There’s a major difference between these events and functions – events cannot return values, you must use variables belonging to the owner of the script if you want to save results of the event. Dormant events can be called manually in the terminal or in a different event using run() method. You can improve readability of your script by combining conditional events with dormant events. Examples:

event(“sleepy”){ print(“Zzz…\n”) }; //A line from a script of the object with “bob” id.

objects.bob.events.sleepy.run();

1. Most methods or accessors need a context based on one of these templates:

[subject].[method/variable] – accessing methods and variables of the subjects: layers, cameras, mouse and env(environment);

cameras.[camera\_id].[method/variable];

layers.[layer\_id].[method/variable];

layers.[layer\_id].[object\_id].[method/variable];

layers.[layer\_id].[object\_id].[module].[method/variable];

layers.[layer\_id].[object\_id].[module].[element\_id].[method/variable];

For example:

layers.first.patrick.var.apples = 10; //In the “first” layer find an object with an id “patrick” and set its variable named “apples” to 10.

You can shorten “layers.[layer\_id].[object\_id]” using built-in variable **objects** like this:

objects.[object\_id].[method/variable];

objects.[object\_id].[module].[method/variable];

objects.[object\_id].[module].[element\_id].[method/variable];

When accessing the owner of the script, you can use built-in variable **me** like this:

me.var.money += 50;

Under the hood, accessing the context by id in this way uses aggregation command **first**:

cameras.[camera\_id] 🡸🡺 first(cameras.id == [camera\_id]);

layers.[layer\_id] 🡸🡺 first(layers.id == [layer\_id]);

objects.[object\_id] 🡸🡺 first(objects.id == [object\_id]).

1. Values or contexts can be assigned to a variable using **let** statement:

let number = 10;

let string = “hello Goo”;

let a = layers.background;

let bob = objects.bob;

let money = objects.bob.var.money;

1. In order to simultaneously access the context of a group of objects use aggregation methods: first, last, all, random. Aggregation templates:

[context].[aggregation]([…].[method/variable] [comparison] [value]);

[context]. [aggregation]([…].[method/variable] [comparison]… […].[method/variable]);

You can assign aggregated context to a variable using **let** statement.

Examples of aggregation:

objects.first(id == “bob”).var.money = 10; //This line does exactly the same thing as this line: “objects.bob.var.money = 10”;

objects.all(group == “tree”).events.spawn\_apples.run(); //For all objects in a group “tree” spawn apples.

let first\_bob = objects.first(group == “bob”);

let last\_bob = objects.last(group == “bob”);

let all\_bobs = objects.all(group == “bob”);

let all\_rich\_kids = objects.all(var.money >= 1000).

1. In order to repeat instructions or whole events use **while** loop. While loop repeats everything in its scope as long as its condition returns true. Examples:

while(objects.x.var.bool==”true”){ objects.x.events.doSomething(); }

let i = 0;

while(i < 10){ x.var.int+=2; i++; }

let x = all(group == “x”);

while(x.size > 0){ me.events.destroyRandomX(); }

Operands in the conditions of the while loop must return single values – groups of objects cannot be used.

1. Each if statement creates new conditional event. Nested statements, after being translated into separate events, are still connected to their parents. Examples of if statements:

if(objects.bob.var.int > 10){ objects.bob.events.doSomething(); }

let group\_a = all(group == “group\_a”);

if(group\_a.var.apples > 5){ group\_a.events.giveOrange(); }

Just like in while loops, operands in the conditions must return single values – groups of objects cannot be used without a special directive like **atomic.** Without directives every operation on groups is treated like a SIMD instruction (Single Instruction Multiple Data).

1. After being triggered, conditional events execute all instructions placed inside them. These are sources of all triggers:
   1. Terminal – “run()” command;
   2. Events – “run()” command;
   3. Time – based on main loop iterations. Passing of seconds and minutes can be detected by using built-in variables **second\_passed** and **minute\_passed** – they are set to true if the stated time has passed in the current iteration;
   4. Camera – based on its state, position and **collision\_with** method.
   5. Keyboard – with variables: **key\_down**, **key\_pressed** and **key\_released**;
   6. Mouse – with variables: **mouse\_moving**, **mouse\_down, mouse\_pressed**, **mouse\_released**. Mouse can also select a group of objects based on their position and approximated size – selected group is saved in a built-in context variable **mouse\_selection**;
   7. Objects – current state;
   8. Variables – current state;
   9. Collision – collisions of objects from the same layer detected in the current main loop iteration;
   10. Editable Text Fields – with **content** variable;
   11. Movement – based on current state and **is\_moving** variable.
2. To create a new object you can use clone() method after choosing an existing object or function new() to create a blank object. Remember that you can use objects from another layers as templates. Examples:

Objects.FirstStar.clone(“SecondStar”);

Objects.SecondStar.var.brightness+=10;

Objects.new(“BlankObject”);

Objects.BlankObject.var.newVariable = 42;

Action of creating new objects can return a handle to an object if **let** statement is used:

let newBob = Layer.bobTemplates.bob123.clone(“newBob”);

newBob.var.type = 10;

newBob.event.prepareBasedOnType();

let newPatrick = Objects.new(“newPatrick”);

1. Accesing and creating new variables inside all objects is done by using **.var** accessor on a selected object and giving an id of a user defined variable. Example:

Objects.MrVariable.var.newVariable = 12;

1. In order to create and destroy elements of every other module in the object use methods **new(string id)** and **destroy(string id)**, for example:

Objects.walter.particles.new(“snow”);

Objects.walter.particles.destroy(“rain”);

1. The list of engine instructions:
   1. break – interrupts the execution of the current scope.
   2. return – interrupts the execution of all events from the object.
   3. reboot – restarts the engine. All boot triggered events will be executed in the next engine iteration.
   4. power\_off – exits the program.
   5. <aggregaton> <source> [contextIDs] [cameraID] [layerID] [objectID] [moduleType] [moduleID] [attribute] [expression] [newContextID] – returns a context that fulfils the boolean expression and optional parameters.

Parameters:

* + aggregation (string): “all” – returns whole context, “first” – returns the first context, “last” – returns the last context, “random” – returns one random context;
  + source (string): “camera”, “layer”, “context”, “\_” – if context ids are not provided, choose one of the highest types of abstractions. Each camera contains its attributes and a bitmap (screen/window), while each layer contains its attributes and objects;
  + [optional] contextIDs (vector<string>) – context ids will be used when source is equal to “context” or “\_”. Each context can have one of these types: “camera”, “layer”, “object”, “text”, "editable\_text", "image", "movement", "collision", "particles", "event", "variable", "scrollbar", “pointer”, “value”. Type of the context affects other parameters;
  + [optional] expression (vector<ConditionClass>) – the list of locations of values and the list of operators that create a relationship between those values, which in turn results in a single boolean value. This whole expression is calculated for each entity separately and if it returns truth, entity is aggregated into the new context. But if expression is empty, this boolean is equal to true by default;
  + [optional] cameraID (string) – if source is equal to “camera” or provided context is of a camera type, only the camera with this id can be aggregated;
  + [optional] layerID (string) – if source is equal to “layer” or provided context is of a layer type, only the layer with this id can be aggregated;
  + [optional] objectID (string) – if source is equal to “layer” or provided context is of a layer or object type, only the object with this id can be aggregated;
  + [optional] moduleType (string) – if source is equal to “layer”, or provided context contains layers, objects or modules; only the module of this type can be aggregated;
  + [optional] moduleID (string) – if source is equal to “layer”, or provided context contains layers, objects or modules; only the module with this id can be aggregated;
  + [optional] attribute (string) – if provided, aggregate this attribute from selected entities;
  + [optional] newContextID (string) – giving an id to a new context creates a variable in the current scope or overwrites the context of an existing variable with the same id.
  1. index <source> [indexes] [contextIDs] [attribute] [newContextID] - takes a list of indexes and returns a context with entities found by indexes.

Parameters:

* + source (string): “camera”, “layer”, “context”, “\_” – if source is equal to “camera” or “layer”, contexts from contextIDs are treated as indexes.
  + [optional] indexes (vector<int/unsigned>) – the list of indexes used to find entities in vectors;
  + [optional] contextIDs (vector<string>): “camera”, “layer”, “object”, “text”, "editable\_text", "image", "movement", "collision", "particles", "event", "variable", "scrollbar", “pointer”, “value” – if source is not provided, the first id serves as the container for index search. The rest of ids, no matter the source, are treated as indexes – they have to be of “pointer” or “value” type.
  + [optional] attribute (string) - if provided, aggregate this attribute from selected entities;
  + [optional] newContextID (string) – giving an id to a new context creates a variable in the current scope or overwrites the context of an existing variable with the same id.
  1. sum <contextID> <contextID> [newContextID] - takes a pair of contexts and returns the sum of these sets.

Parameters:

* + contextID (string) – id of the selected context. This instruction accepts all types: “camera”, “layer”, “object”, “text”, "editable\_text", "image", "movement", "collision", "particles", "event", "variable", "scrollbar", “pointer”, “value”;
  + [optional] newContextID (string) – giving an id to a new context creates a variable in the current scope or overwrites the context of an existing variable with the same id.
  1. intersection <contextID> <contextID> [newContextID] - takes a pair of contexts and returns the intersection of these sets.

Parameters:

* + contextID (string) – id of the selected context. This instruction accepts all types: “camera”, “layer”, “object”, “text”, "editable\_text", "image", "movement", "collision", "particles", "event", "variable", "scrollbar", “pointer”, “value”;
  + [optional] newContextID (string) – giving an id to a new context creates a variable in the current scope or overwrites the context of an existing variable with the same id.
  1. difference <contextID> <contextID> [newContextID] - takes a pair of contexts and returns the difference of these sets.

Parameters:

* + contextID (string) – id of the selected context. This instruction accepts all types: “camera”, “layer”, “object”, “text”, "editable\_text", "image", "movement", "collision", "particles", "event", "variable", "scrollbar", “pointer”, “value”;
  + [optional] newContextID (string) – giving an id to a new context creates a variable in the current scope or overwrites the context of an existing variable with the same id.
  1. value <expression> [newContextID] – returns a context made out of given literals or values found in different sources.

Parameters:

* + expression (vector<ConditionClass>) – the list of locations of values and literals. Each value has one of these sources: “context”, “camera”, “layer”, “object”, “variable”, “literal”, “on\_boot”, “second\_passed”, “key\_pressed”, “key\_pressing”, “key\_released”, “any\_key\_pressed”, “any\_key\_pressing”, “any\_key\_released”, “mouse\_moved”, “mouse\_pressed”, “mouse\_pressing”, “mouse\_released”. If you choose “literal”, you can provide literals from these types: bool, int, double, string.
  + [optional] newContextID (string) – giving an id to a new context creates a variable in the current scope or overwrites the context of an existing variable with the same id.
  1. literal <type> <literals> [newContextID] – returns a context filled with provided literals.

Parameters:

* + type <string>: “bool”, “int”, “double”, “string”;
  + literals (vector<VariableModule>): bool, int, double, string;
  + [optional] newContextID (string) – giving an id to a new context creates a variable in the current scope or overwrites the context of an existing variable with the same id.
  1. random\_int literal <literal> <literal> [newContextID] – returns a random integer from the range provided in the pair of literals.

Parameters:

* + literal (VariableModule): bool, int, double, string;
  + [optional] newContextID (string) – giving an id to a new context creates a variable in the current scope or overwrites the context of an existing variable with the same id.
  1. random\_int context <contextID> <contextID> [newContextID] – returns a list of random integers from the ranges provided in the pair of contexts.

Parameters:

* + contextID (string) – id of the choosen context. Context types allowed by this instruction are: “value” and “pointer”;
  + [optional] newContextID (string) – giving an id to a new context creates a variable in the current scope or overwrites the context of an existing variable with the same id.
  1. find\_by\_id <source> [contextIDs] [cameraID] [layerID] [objectID] [moduleType] [moduleID] [attribute] [newContextID] – returns a context filled with entities which ids are equal to ones provided in parameters. If no ids are provided, aggregates all possible entities.

Parameters:

* + source (string): “camera”, “layer”, “context”, “\_” – if a context is not provided, choose one of the highest types of abstractions. Each camera contains its attributes and a bitmap (screen/window), while each layer contains its attributes and objects;
  + [optional] contextIDs (vector<string>) – context ids will be used when source is equal to “context” or “\_”. Each context can have one of these types: “camera”, “layer”, “object”, “text”, "editable\_text", "image", "movement", "collision", "particles", "event", "variable", "scrollbar", “pointer”, “value”. Type of the context affects other parameters;
  + [optional] cameraID (string) – if source is equal to “camera” or provided context is of a camera type, only the camera with this id can be aggregated;
  + [optional] layerID (string) – if source is equal to “layer” or provided context is of a layer type, only the layer with this id can be aggregated;
  + [optional] objectID (string) – if source is equal to “layer” or provided context is of a layer or object type, only the object with this id can be aggregated;
  + [optional] moduleType (string) – if source is equal to “layer”, or provided context contains layers, objects or modules; only the module of this type can be aggregated;
  + [optional] moduleID (string) – if source is equal to “layer”, or provided context contains layers, objects or modules; only the module with this id can be aggregated;
  + [optional] attribute (string) – if provided, aggregate this this attribute from selected entities;
  + [optional] newContextID (string) – giving an id to a new context creates a variable in the current scope or overwrites the context of an existing variable with the same id.
  1. let <contextID> <newContextID> - creates a variable with provided id from the selected or last existing context or overwrites the context of the existing variable with the same id. New variables will exist only in the current scope.

Parameters:

* + contextID (string) – id of the context. Context can have one of these types: “camera”, “layer”, “object”, “text”, "editable\_text", "image", "movement", "collision", "particles", "event", "variable", "scrollbar", “pointer”, “value”;
  + newContextID (string) – id of the new variable or the id of existing variable intended for a context overwrite.
  1. clone <contextID> <contextID> [changeOldID**]** - clones the values and structure of the right context to the left context.

Parameters:

* + contextID (string) – id of the context. Context can have one of these types: “camera”, “layer”, “object”, “text”, "editable\_text", "image", "movement", "collision", "particles", "event", "variable", "scrollbar", “pointer”, “value”;
  + [optional] changeOldID (bool) – if true, the left context will inherit the id of the right context (with the last number incrementation). True by default.
  1. <operator> <contextID> <contextID> [newContextID] – returns a context with a result of an arithmetic operation on the pair of contexts (with numeric values).

Parameters:

* + operator (string): "+", "-", "\*", "/";
  + contextID (string) – id of the context. This instruction accepts these two context types: “pointer” and “value”;
  + [optional] newContextID (string) – giving an id to a new context creates a variable in the current scope or overwrites the context of an existing variable with the same id.
  1. <operator> <contextID> – increments or decrements the numeric value found in the context by 1.

Parameters:

* + operator (string): "++" – increment by 1, "--" – decrement by 1;
  + contextID (string) – id of the context. This instruction accepts these two context types: “pointer” and “value”.
  1. <operator> <contextID> <contextID> – moves a numeric value from the right context to the left context based on the choosen operator.

Parameters:

* + operator (string): "=", "+=", "-=", "\*=", "/=" – these work exactly like in c++;
  + contextID (string) – id of the context. This instruction accepts these two context types: “pointer” and “value”.
  1. in <contextID> <contextID> [newContextID] - returns true if any entity from the left context occurs in the right context. In case of numeric values, instruction compares their values. For other entities like layers, instruction compares their ids, but only if both entities are of the same type.

Parameters:

* + contextID (string) – id of the context. This instruction accepts these two context types: “pointer” and “value”;
  + [optional] newContextID (string) – giving an id to a new context creates a variable in the current scope or overwrites the context of an existing variable with the same id.
  1. new <type> [layerID] [objectID] [size1] [size2] [newIDs] [newContextID] - creates new entities and returns a context with their pointers.

Parameters:

* + type/source (string): “camera”, “layer”, “object”, “text”, "editable\_text", "image", "movement", "collision", "particles", "event", "variable", "scrollbar" – the type of the new entities;
  + [semi-optional] layerID (string) – the id of the layer with the role of a container for new objects. Required only if the new entities are objects;
  + [semi-optional] objectID (string) – the id of the object with the role of a container for new modules. Required only if the new entities are modules;
  + [optional] size1 (int) – this is the number of new entities that will be created. If not provided, operation will try to get this number from size2 <parameter>
  + [optional] size2 (string) – id of the context of the type “pointer” or “value”. This is the number of new entities that will be created;
  + [optional] newIDs (string) – the id of the context with the type “pointer” or “value”. This context provides the list of strings that will become the new identificators for new objects. You can provide any number of ids and if the number is too low, the last id will be repeated. Although, to ensure the uniqueness in the given container, indexing numbers will be automatically added to ends of repeated ids and incremented by one for every next repetition. The same automatic action will be taken if no new ids are provided;
  + [optional] newContextID (string) – giving an id to a new context creates a variable in the current scope or overwrites the context of an existing variable with the same id.
  1. delete <contextID> - deletes all entities provided in the context.

Parameters:

* + contextID (string): “camera”, “layer”, “object”, “text”, "editable\_text", "image", "movement", "collision", "particles", "event", "variable", "scrollbar" – id of the context with entities selected for deletion.
  1. bind <contextID> <source> <scriptPaths> - adds or removes paths to scripts from objects.

Parameters:

* + contextID (string) – id of the context with objects;
  + source (string):
    - “context” or “c” – next parameter will require an id of the context with paths to scripts. Instruction will bind those paths to provided objects;
    - “literal” or “l” – next parameter will require a list of string values – script paths. Instruction will bind those paths to provided objects;
    - “remove\_context”, “rcontext” or “rc” – next parameter will require an id of the context with paths to scripts. Instruction will remove those paths from provided objects;
    - “remove\_literal”, “rliteral” or “rl” – next parameter will require an id of the context with paths to scripts. Instruction will remove those paths from provided objects;
    - “reset” – instruction will remove all binded paths from provided objects.
  + scriptPaths (string / vector<string>) – paths to scripts, their purpose depends from “source” parameter.
  1. build <contextID> [reset] – instruction translates all scripts binded to objects provided in the context and creates events based on those scripts.

Parameters:

* + contextID (string) – id of the context with objects intended for event building;
  + [optional] reset (bool) – if true, before creating new events, instruction removes all events from provided objects.