**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 tcoranslate your script into events with “translate()” method like this:

objects.bob.scripts.new\_script.translate();

1. 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();

If you want to name a normal triggerable event, use “label [name]” statement like this:

label if\_1{

if(me.var.hp <= 0){ me.events.deathScreen.run(); }

}

Named triggerable events can also be manually called, but it’s not recommended since they can be simply ignored (if conditions are not met) or repeated too mant times.

1. Engine can halt the execution of an event if the infinite loop is detected. If needed it can also deactivate a dangerous event – deactivated event cannot be triggered by other events. In order to activate such event, use methods: “run()” or “elevated\_run()”. The second method forces the engine to ignore dangerous behavior.
2. 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 operators: first, last, all, random. Aggregation templates:

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

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

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

Examples of aggregation:

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

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

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

let last\_bob = last(group == “bob”);

let all\_bobs = all(group == “bob”);

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

1. In order to repeat instructions or whole events use **while** loop. It will repeat 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 without a special directives like **atomic**. Without directives every instuctions on groups is treated like a simd instruction (single instruction multiple data).

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. In order to use groups of objects with if statements and while loops, user must use special directives to control the flow of the groups. Available directives are:

* atomic() – iterates through all selected groups, takes one object from each and executes all instructions inside its scope in SISD mode (single instruction single data). In other words atomic directive is an equivalent of for loop iterating through a vector. Minimal complexity of atomic directive scope is a product of the multiplication of all selected groups’ sizes.
* unique() – takes one object from the beginning of each selected group, removes taken objects from these groups and executes instructions inside its scope in SSID mode – then repeats the process until it cannot form a full tuple of objects. Minimal complexity of unique directive scope is a size of the smallest selected group.
* random() - takes one random object from each selected group, removes taken objects from these groups and executes instructions inside its scope in SSID mode – then repeats the process until it cannot form a full tuple of objects. Minimal complexity of random directive scope is a size of the smallest selected group.

Example:

let guy = all(group==”a”);

let thug = all(group==”b”);

atomic(guy, thug){

if(guy.var.money > 0){

let payment = rand(0, guy.var.money);

thug.var.money += payment;

guy.var.money -= payment;

}

else{

guy.var.hp -= 1;

}

}

Description of the example: Every guy is visited by every thug, but only one guy can be mugged by one thug at a time – in this way money won’t be duplicated.

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 order of actions in the engine loop:

-Events are put in objects

-primaryTriggerTypes in Events

-EventsLookupTable: updates when events are modified. Only updates vectors triggered by the right module.

-TriggeredObjects: is recreated every iteration, it is a base for main event loop.

1. QUESTION - Killing objects in a variable, decreases the size of this variable. Spawning new objects from the variable doesn't change the size of this variable. You need to aggregate objects to this variable again to change the size. Thus a question arises: if you re-aggregate a variable in a nested conditional statement, can you update the state of this variable outside the statement? Or should you ignore the changes?

**[script].****translate(string event\_id=”[script]”):**

Description: Method prompts the interpreter to translate selected script into an event named **event\_id** by creating a new event or updating existing one in the selected object. Default value of **event\_id** is the name/id of translated script.

Example: me.scripts.recursion.translate()

**[event].run(int n=1):**

Description: Method executes a chosen event **n** times, which is limited by the environmental variable **stack\_size**. Run() dynamically adds an object to the end of the **TriggeredObjects** list.

Example: me.events.recursion.run().

**[event].elevated\_run(int n=1):**

Description: Method executes a chosen event **n** times without any limitation and by ignoring all dangers like infinite loops, modification of system files or having sex with your mom.

Example: me.events.recursion.elevated\_run().

**[event].stop():**

Description: Method deactivates the event, making it indifferent to any triggers.

Example: me.events.eat\_bananas.stop().

**print(string text):**

Description: Command prints a provided text in the terminal. It’s the best tool for debugging!

Example: print(“Hello Goo!\n”);

**let [name] = [value]:**

Description: This statement assigns a literal or context to a new variable created in the script owner. Examples:

let banana = “”

**each\_second:**

Description: Simple condition check triggered every second. You can simulate it by incrementing a variable every iteration and checking its value in a simple if statement.

Result: bool

Example:

   each\_second{ me.var.money ++; }

**isolated\_if:**

Description: this if statement checks only the state of the currently analyzed object.

Result: bool

Example:

isolated\_if( me.var.money > 64 ) {

me.var.money --;

}

**half\_if:**

Description: this if statement inherits functionality of **isolated\_if** and compares currently analyzed object’s state (including constants) to a one specific object at the time.

Result: bool

Examples:

half\_if( me.var.money > you.var.money ) { … }

half\_if( you.var.money == 90 && bob.var.gold < 45 ) { … }

**full\_if:**

Description: this if statement compares constants, currently analyzed object’s state, one specific object or even two different objects in every comparison.

Result: bool

Examples:

full\_if( you.var.money > bob.var.money) { … }

full\_if( me.var.money == 90 && bob.var.gold < jeff.var.diamonds

|| steve.var.cash < patrick.var.bread ) { … }

**first:**

Description: this conditional statement returns the first object

Result: bool

Examples:

half\_if( me.var.money > you.var.money ) { … }

half\_if( you.var.money == 90 && bob.var.gold < 45 ) { … }