Genesis

Language Reference Manual

Sam Cohen	**slc2206**	Manager
Michael Wang	**mlw2167**	Language Guru
Leo Stilwell	**ls3223**	Language Guru
Jason Zhao	**jsz2107**	Systems Architect
Saahil Jain	**sj2675**	Tester

Contents

Introduction

Genesis is a special-purpose language inspired by popular game engines like Unity. The primary goal of Genesis is to facilitate the development of 2D single-player games. We integrate several popular programming structures / paradigms into our language as objects, and provide users customization by overriding functions. We designed Genesis to be fast and easy-to-use, while providing many features of modern programming languages like type and memory safety, zero-cost abstractions, and OO design.

Language Elements

Delimiters

Curly Braces are used to group lines together in control flow, functions, etc.

Semicolons are used to terminate statements.

Commas are used to separate values (like values in a list, arguments to a function, etc.).

Whitespace is used to separate tokens (E.g. 1 + 1)

Starting a game

The start of each file begins with the START keyword, which initializes a board. The rest of the file can detail the board with various actions. The user should pass in the dimensions of the board, the size of each square on the board, and the board's background color. The format is:

START *length*, *width*, *size of squares, background color*

For example,

```
START 1280, 960, 1, <255, 255, 255>
// Starts a 1280x960 board with square size 1 with a white background
```

Operators

This section will be divided into 4 sections for organizational purposes, logical operators, comparison operators, math operators and custom, game-specific operators.

Logical Operators

Our language implements logical AND(∧), OR(∨), and NOT(¬).

Operator	Keyword
Logical AND (∧)	&&
Logical OR (∨)	||
Logical NOT (¬)	!

These operators can be used in the following manner:

```
bool a = True;
bool b = False;
a && b; // Returns False
a || b; // Returns True
!a     // Returns False
```

Comparison Operators

Our language contains all of the standard comparison operators, greater than, less than. equals, not equals, greater than or equals, less than or equals.

Operator	Language Symbol
Equals	==
Not Equals	!=
Greater Than	>
Less Than	<
Greater Than or Equals to	>=
Less Than or Equals to	<=

These operators can be used in the following manner:

```
int a = 1;
int b = 2;
a > b  // Returns true
a == b // Returns false
```

```
a < b  // Returns true
```

Arithmetic Operators

Our language implements the standard mathematical arithmetic operators including multiply, divide, add, subtract as well as a modulus operator.

Operator	Language Symbol
Multiply	*
Divide	/
Add	+
Subtract	-
Modulo	%

These operators can be used in the following manner:

```
int a = 1;
int b = 2;

a + a // Returns 2
b / a // Returns 2
a - a // Returns 0
```

Game-specific Operators

Our language implements two game specific functions, involving collisions between objects and key presses. These can be called within update functions.

Operator	Symbol
Collision Returns TRUE if two clusters are touching	@
Key Press Returns TRUE if a specified key is pressed.	^

Control Flow

Our language implements if statements, if-else statements, if-else statements, and while loops:

Control Structure	Description

if (*statement*) {
  block
}
	The code in *block* gets executed if *statement* evaluates to TRUE.

if (*statement 1*) {
  block 1
} else {
  block 2
}
	If *statement 1* evaluates to TRUE, then *block 1* gets executed. Otherwise, *block 2* gets executed.

if (*statement 1*) {
  block 1
} elif (*statement 2*) {
  block 2
}
	If *statement 1* evaluates to TRUE, then *block 1* gets executed. Otherwise, *block 2* gets executed if *statement 2* evaluates to TRUE.

while (*statement)* {
  block
}
	While *statement* evaluates to TRUE, block is executed.

Comments

Comments are written using the following operators. There are no nested comments.

Operator	Description
//	After the // symbol, anything written on the same line will be ignored
/* */	Anything between /* */ will be ignored

Functions

User Defined Functions

Creating Functions

Users can define their own functions by using the keyword func. Functions can be defined anonymously or with an identifier. The user must specify the function's arguments. An anonymous function takes the form:

func (args) {}

Making an identifier for the function just involves assigning it to a function name, in the same style as objects, with the form:

func functionName = func (args){}

For example:

```
func foo = func (int a, int b){
      return a;
}
```

Functions defined in the global scope can be called from any scope, while functions defined in classes are member functions of that class and can only be referred to when referencing an instance of that class.

Calling Functions

Calling a function involves writing the function name and then writing the arguments passed to the function in order, separated by commas inside of parenthesis. The function call will evaluate to the value returned by the function

For example:

```
int x = foo(5,6); //x equals 5
```

The Update Function

Update is a reserved function that gets called every frame. It has gets passed the amount of time that has elapsed since the last frame has been rendered. It takes the format

func (float deltaTime){}

There is a single global update function and also update functions tied to clusters, which are executed until the clusters are deleted.

An example of a global update function is:

```
update = func (float deltaTime){
      print("frame was updated");
}
```

Print Function

Output can be printed to the console using the print function. For example:

```
print("hello world") //Prints hello world to the console.
```

Classes

Classes are user defined types that can contain their own variables and functions. Genesis' classes must be concrete (all member functions must be implemented) and have unique names. Optional parameters can be specified in the class definition that can be used during object instantiation. All variables and methods in the class are considered "public" (ie. they can be accessed from outside the class), and there is no inheritance.

```
// This class can be used to create a BST
class Node(val) {
      Node left;  // Recursive reference
      Node right; // Recursive reference
      int value = val;

      func doSomething() {
            print("Hii\n");
      }
}

Node root = Node(5);  // Creates a Node with value = 5
```

Data Types

Our language implements 4 basic data types, integers, floats, booleans and strings, in addition to several complex data types. Keywords are type-sensitive.

Variable names can be any string composed of alphanumeric characters and "_" (underscore) as long as they are not a reserved name or keyword.

Primitives

Integers (keyword: int)

32-bit numerical integers stored using two's-complement.

```
int a = 32;
int b = -13;
```

Floats (keyword: float)

Double precision (64-bit) IEEE 754 floating point numbers.

```
float a = 32.4;
```

String

String must start with a alphabetic character and can then include any alphanumeric character afterwards.

```
string str = "Hello, World!";
str.length() // Evaluates to 13
```

Booloean

Can be either true or false.

```
boolean a = True;
```

Casting

There is **no** implicit type casting. All casting must be done explicitly by passing in the variable to-be-casted into a globally available function of the name of the type to-be-casted-to.

```
float a = 32.4;
int   b = int(a); // b = 32

int   c = 1;
float d = float(c); // d = 1.0, explicit casting

a + c; // Error: types don't match
a + float(c); // Evaluates to float with value of 33.4
```

Complex Data Types

List

A List is a dynamically-sized container of objects. Lists support the operations append, remove, and index operations.

- [list].append(item): Appends item to the end of [list]
- [list].remove(index): Removes the item at the

```
int[] items;
items.append(1);
items.append(2); // items contains [1, 2]
items.remove(1); // items contains [1]
```

Color

Each Color object is represented by a RGB color sequence which defines the color displayed on the screen.

```
Color c = <110, 255, 0>;
```

Tile

Each tile is a 1 pixel by 1 pixel object placed at a position represented by an X, Y coordinate pair, along with the color of the tile. Tiles should belong to a cluster.

```
Color c = <110, 255, 0>;
Tile p = Tile(0, 89, c);
```

Cluster

A cluster is a collection of tiles which represent a general geometric shape.

Making a New Cluster

When making a new cluster, the user should specify which tiles are in the cluster, as well as an update function for the cluster that will be called by the system every time a new frame is rendered.

Cluster Operations

After being created, clusters can be moved within the game world and deleted from the world altogether, using the move() and delete() functions.

Function	Description
move(int *x*, int *y*)	Moves the cluster to the right x units and down y units.
delete()	Deletes the object.

The following code represents a cluster composed of a single tile of color c.

```
Tile[] tiles1;
tiles.append(Tile((0, 89), <255, 0, 0>));
Cluster c1 = Cluster((0, 0), tiles);
```

Sample Program

```
// non-recursive implementation of GCD
func int gcd(int p, int q) {
      while (q != 0) {
          int temp = q;
          q = p % q;
          p = temp;
        }
      return p;
    }
```