Language Reference Manual

Dylan Hicks, Theodore Marin, William McAuliff, James Wen, Sean Wong

TABLE OF CONTENTS

Introduction

This manual serves as a comprehensive description of GAME, the language built to make analyzing metrics easier. We detail its lexical conventions, specify the syntax, identifiers, and expressions, and provide a grammar that captures the set of syntactically-correct GAME programs.

Lexical Conventions

GAME programs can be broken down into tokens, of which there are five types: identifiers, keywords, constants, string literals, operators. Certain characters like space, tab, newline, and comments act as separators. These characters can aid in a GAME program's readability for users but are also sometimes necessary because, in certain situations, they delineate the start and end of individual tokens in a token stream. These characters can be grouped under the term, "whitespace".

Comments

Comments begin with the number sign (#) and are terminated by any newline.

```
#This is a comment
```

A comment can be placed at the end of a line that contains functional code.

```
text phrase = "This is not a comment" #This is a valid comment
```

However, a comment cannot occur within a text literal.

```
text phrase = "The following is not a comment #content"
```

The content in the portion of the text literal to the right of the number sign (#) is not lexically considered a comment in this circumstance.

Multi-line comments are currently designated with a series of comment lines that each start with the number sign (#) are individually terminated with newlines.

```
#This is a
#multi-line comment
#that spans three lines.
```

Identifiers

Identifiers have to start with an underscore (_), uppercase letter (a-z), or lowercase letter (A-Z).

Ex. Valid Identifiers:

```
a
m
_
A
```

```
Z
```

Ex. Invalid identifiers:

```
2
$
-
```

The initial character of the identifier can then optionally be followed by any combination of digits, underscores, and uppercase or lowercase letters.

Ex. Valid Identifiers:

```
aParam
my_instance
_CORE1
AbeLincoln
_________
_12345
```

Ex. Invalid identifiers:

```
2fort
$param
param-
```

Only identifiers that are 1-32 characters in length are valid. Rather than only have the compiler consider up to the first 32 characters of an identifier, it will throw a compile error when detecting identifiers longer than 32 characters.

If an identifier is lexically the same as a reserved keyword, a compile error will occur. For a comprehensive list of reserved keywords, please refer to the reserved keywords section.

However, identifiers that are lexically similar to reserved keywords but do not exactly match them are acceptable (but not recommended).

Ex. Valid Identifiers: (Not recommended for clarity purposes)

```
AND
aND
and_start
Start
start0
_start
```

Ex. Invalid identifiers:

```
and
start
```

Reserved Keywords

loop	start	while
set	if	else
function	return	null
new	true	false
class	include	load
from	num	text
list	geteach	in
where	export	to
break	continue	

These reserved keywords cannot be used as identifiers. Please refer to the Identifiers section for clarification.

Reserved Characters/Operators

+	-	*	/
%	==	=	#
!=	<	<=	>
>=	.	(	)
and	or	not	,
[	]	“	\n
{	}		

These reserved characters/operators have functional purposes and the compiler will parse them as such. Also, please note that none of these are valid characters for identifiers.

Types

num

The **num** is GAME’s standard data type for representing rational numbers (there is no support or representation for irrational numbers).

Every num is actually represented on the back-end side by a float in Python (which is usually represented as a double in C). Hence, the user or programmer does not have to worry about precision and which numerical data type to use as the num will provide the necessary precision if needed.

As such, num precision limits are machine-dependent.

However, even though a num is represented intrinsically as a float, it is easy for the user to use it as if it was an integer (did not have a decimal component) with built-in library functions. For example, if the programmer wanted to consider a series of team jersey numbers, using the num_form function from the standard library in conjunction with the jersey nums would allow for easy two-digit print formatting.

Ex.

```
num jersey = 20
num average = 17.98
num point_difference = -19
num total_points = 39128334
```

text

The **text** is GAME's data type for representing sequences of ascii characters (text literals). Every text is represented by a Python str. As such, text length limits are machine-dependent. For some characters to be properly represented in a text by the compiler, escape characters have to be used. Please refer to the Escape Characters section for more information.

Ex.

```
text attr = “Cool”
text quote = “\”He was the best player we’ve ever had.\””
```

Concatenating with the + operator is possible for multiple texts, text literals, nums, or a combination of these as long as at least one text is in the expression.

The following are valid operations:

```
text attr = “Cool” + “Game”
text attr2 = attr + 2
num ex = 3
text attr3 = attr + ex
```

The following is invalid:

```
text numbers = 2
text numbers = 2 + 3
```

```
list jerseys = {2, 3}
text numbers = “Jerseys “ + jerseys
```

Please note that the + operator is left associative and impacts the concatenation of texts as follows:

```
text attr = “Cool” + 2 + 2 #attr is now “Cool22”
text attr = 2 + 2 + “Cool” #attr is now “4Cool”
```

In the second case, `2+2` is evaluated because of the left association and then the result of that is concatenated with the text `“Cool”`.

bool

The **bool** is GAME’s data type for representing binary values (true or false).

list

The **list** is GAME’s data type for representing collections of similar data types or objects. There are library methods that can be called by a list: `list.add(`*`element`*`)`, `list.remove(`*`element`*`)`, `list.addAt(`*`num`* `index)`, `list.removeAt(`*`num`* `index)`.

Escape Sequences

Escape sequences are utilized in text data types and text strings when certain characters (that have other lexical meanings) are desired in the text.

For example, the following would cause a compile error due to unbalanced quotation marks:

```
text incorrect = “This does “not compile.”
```

The following would also cause a compile error due to the extra quotation marks forming a text and breaking the token stream with the extra, misplaced text:

```
text causes_extra_text = “This also does “not” compile.”
```

This is because the compiler perceives this syntax as such:

```
text causes_extra_text = “This also does “
not
  ” compile.”
```

`“ compile.”` is a hanging text.

This would be valid text as the quotations are demarcated with the proper escape sequences:

```
text incorrect = “This does \“compile.\” ”
```

Desired Character	Escape Sequence
\	\\
newline	\n
tab	\t
“	\”

Scope

Lexical Scope

The scope for an identifier begins right after its declaration and ends at the end of its current scope.

Please note that identifiers that are declared with the exact same name in the same scope will cause a compile-time error.
Ex. (Causes a compile-time error)

```
num x
num x = 5
```

Function Scope

The scope for an identifier declared in a function is the rest of the function up until the end of the function. Please note that the only constructs that can be placed outside of classes are include statements. functions, and other classes. Identifiers, text literals, etc. cannot be placed outside of classes.

Loop Scope

The scope for an identifier declared within a loop body or the loop start expression is the rest of the loop body up until the end of the loop operation. Please note that an identifier that shares a name with an existing identifier that has been declared outside of the loop will not throw a compile-time error. However, any references or statements that use the identifier in the loop will refer to the current value of the identifier declared in the loop.

Ex.

```
num i = 1
  loop (start num i = 0, while i <= 3, set i = i + 1) {
    num test = i # On first iteration, i’s value is 0, not 1
  }
```

Expressions

An expression is anything that returns a value.

Constant

constant	→	{ *constant* } *constant* , *constant* *num_constant* *txt_constant*
num_constant	→	r'\d+\.?d*'
txt_constant	→	r'"[[^"][\\"]]*"' (string constants surrounded by double quotes, \ is escape character)

A constant can be a number or text constant, or a list of other constants. The comma operator is left associative.

Function Declaration

function_def	→	function *id* (*function_args*) { \n *function_lines* } *var_type* function *id* (*function_args*) { \n *function_lines* return *expression* \n }

function_lines	→	*function_lines statement* \n *function_lines* \n ϵ

The *id* will be used to represent the function and must be unique. If a *var_type* is specified there must be a return statement with an *expression* resulting in that *var_type*.

Function Calls

expression	→	*obj_expression* . *id* (*function_args*)

statement	→	*object_expression* . *id* (*function_args*)

obj_expression	→	*obj_expression . id* *id*

id must be a previously defined function. If the function does not return anything it is a statement, if it returns some value it is an expression. An *obj_expression* allows you to access fields of an object if desired.

Class Declaration (initial naming and body)

class_def	→	class *id* { \n class_lines } class *id* extends id { \n class_lines }

class_lines	→	*class_lines function_def \n* *class_lines variable_def \n* *class_lines \n* ϵ

The first *id* represents the class and must be previously unused. The optionally second *id* references an higher up class to extend.

Class Referencing (Creating objects)

variable_def	→	*var_type id* = new *var_type* *var_type id* = new *var_type* { \n *mul_variable_def* }

mul_variable_def	→	*mul_variable_def variable_def \n* *variable_def \n*

To create an object you must specify the *var_type* and a previously unused *id*. Optionally, you may define any fields of the object. Any undefined fields will be set to null.

Operators

Multiplicative (*, /, %)

expression	→	*expression * expression* *expression / expression* *expression % expression*

The `*`, `/`, and `%` operators exhibit left to right associativity.

For the first production above, the value of the production is the product of the values of the *expression* non-terminals in the production, with the first *expression* non-terminal serving as the multipicand and the second *expression* non-terminal serving as the multiplier.

For the second production above, the value of the production is the quotient of the values of the *expression* non-terminals in the production, with the first *expression* non-terminal serving as the dividend and the second *expression* non-terminal serving as the divisor.

For the third production above, the value of the production is the remainder of the division of the values of the *expression* non-terminals in the production, with the first *expression* non-terminal serving as the dividend and the second *expression* non-terminal serving as the divisor.

Additive (+, -)

expression	→	*expression* + *expression* *expression* - *expression*

The + and - operators exhibit left to right associativity.

For the first production above, the value of the production is the sum of the values of the *expression* non-terminals in the production, with the first *expression* non-terminal serving as the augend and the second *expression* non-terminal serving as the addend. Please note that if the *expression* non-terminals are actually text values or text literals, then appending will occur. Refer to the text data type section for more information.

For the second production above, the value of the production is the difference of the values of the *expression* non-terminals in the production, with the first *expression* non-terminal serving as the minuend and the second *expression* non-terminal serving as the subtrahend.

Relational Operators: (>, >=, <, <=)

expression	→	*expression* > *expression* *expression* >= *expression* *expression* < *expression* *expression* <= *expression*

The >, >=, <, and <= operators exhibit left to right associativity.

For the first production above, the entire boolean value of the production is true if the first *expression* in the production has a greater value than the second expression in the production, otherwise, it is false.

For the second production above, the entire boolean value of the production is true if the first *expression* in the production has a greater or equal value than the second expression in the production, otherwise, it is false.
For the third production above, the entire boolean value of the production is true if the first *expression* in the production has a lesser value than the second *expression* in the production, otherwise, it is false.
For the fourth production above, the entire boolean value of the production is true if the first *expression* in the production has a lesser or equal value than the second expression in the production, otherwise, it is false.

Equality Operators (==, !=)

expression	→	*expression == expression* *expression != expression*

The `==` and `!=` operators exhibit left to right associativity.

For the `==` operator and first production above, the entire boolean value of the production is true if the first *expression* in the production has a value equal to the second *expression* in the production, otherwise, it is false.
For the `!=` operator and second production above, the entire boolean value of the production is true if the first *expression* in the production has a value that is not equal to the second *expression* in the production, otherwise, it is false.

and

expression	→	*expression* and *expression*

The `and` operator exhibits left to right associativity.

The entire boolean value of the production is true if both of the boolean values of the *expression* non-terminals in the production are true. Otherwise, the entire boolean value of the production is false.

- `and` is a short-circuit operator. If the first expression is false, the entire expression is false and the second expression is not evaluated.
Ex.
```
p and q
```
If p is false, the entire p and q expression is false and q is not evaluated.

Truth table:

p	q	p and q
false	false	false
false	true	false
true	true	true
true	false	false

or

expression	→	*expression* or *expression*

The `or` operator exhibits left to right associativity.

The entire boolean value of the production is true if the boolean value of either *expression* non-terminal in the production is true. Otherwise, the entire boolean value of the production is false.

- `or` is a short-circuit operator. If the first expression is true, the entire expression is true and the second expression is not evaluated.
Ex.
`p or q`
If p is true, the entire p and q expression is true and q is not evaluated.

Truth table:

p	q	p or q
false	false	false
false	true	true
true	true	true
true	false	true

not

expression	→	not *expression* (not precedence)

The `not` operator exhibits right to left associativity.

The entire boolean value of the production is the inverted result (false to true or true to false) of the *expression*.

Truth table:

p	not p
false	true
true	false

Initialization and Assignment Expressions: (=)

variable_def	→	*var_type id* *var_type assignment*

assignment	→	*id* = *expression*

The `=` operator exhibits right to left associativity.

The *var_type* is the data type (`list`, `num`, `text`, `bool`, or some class) of the variable and *id* is the identifier for that variable. The first production is initializing a variable of type *var_type* that is denoted by the identifier *id*. In the assignment production, the value on the right of the `=` operator is assigned to the variable denoted by the *id* identifier on the left of the `=` operator.

Statements

statement	→	*variable_def* *assignment* *loop* *if_statement* *data_statement* break continue

Once a function has been properly defined in a GAME, the body of the function is made up of a series of statements. An individual statement can lead to a *variable_def*, *loop*, *if_statement*, *data_statement*, *break*, *continue*.

Loops

loop	→	loop (*loop_expression*) { \n *function_lines* }

When a series of actions needs to be repeated for a some number of iterations, a loop allows GAME programs to do so in a concise manner. There is a standard type of loop, and two other types that are useful in certain scenarios.

The standard type of loop is expressed by starting with the word, "`loop`", followed by a pair of parenthesis and a pair of curly brackets. Within the parentheses, there should be a *loop_expression*. Enclosed with the curly brackets, there should be at least one newline character, followed by a series of *function_lines*. Errors will occur without one or more newline characters separating the opening curly bracket and the *function_lines*.

Loop Expressions

loop_expression	→	*loop_expression* , *loop_expression* start *variable_def* while *expression* set *assignment* ϵ

The *loop_expression* of a standard loop is made up of three primary components, `start`, `while`, and `set`, which can be written in any order and can occur any number of times.

The `start` keyword precedes a *variable_def*, which is the section of the loop where a new variable can be declared. Each of the loop expressions that begin with the `start` keyword will be executed by GAME only once, before any loop iterations.

The `while` keyword precedes an *expression* that must evaluate to a boolean primitive. Each of the loop expressions that begin with the `while` keyword will be executed by GAME immediately after the execution of the start loop expressions, and afterward will be executed before any loop iterations. All of the while loop expressions must evaluate to true in order for a loop iteration to occur. To put it another way, once one of the while loop expressions evaluates to false, the loop will halt.

The `set` keyword precedes an an *assignment*, which is the section of the loop where values are assigned to variables at the end of each loop iteration. Set loop expressions are especially useful as a way to increment or decrement variables being used as an index or loop counter.

Note: It is admissible for a loop to have a parenthesis holding only the empty string, in which case would represent an infinite loop in a GAME. Infinite loops can be desirable in some cases, which is why GAME allows for this behavior.

At a high level, the main loop in GAME can be thought of as an "enhanced while-loop" from languages like C and Java. In those languages, the while-loop only checks a single condition, requires that the programmer declare any index variables before writing the while-loop, and requires that the programmer write code to increment or decrement the variables within the while-loop's body. GAME's main loop moves these three components into the parentheses, so that the programmer can immediately see all of the code relevant to the loop. This is similar to the for-loop in appearance, but, as mentioned previously, the difference is that GAME allows for any number of initialization expressions, condition expressions, and increment/decrement expressions, and organized in any order.

For Each Loop

loop	→	foreach (*var_type id* in *id*) { \n *function_lines* }

One of the two specialized loops provided by GAME is the **"foreach" loop**. This type of loop is used in scenarios where there is a need to iterate over **each** element of a list. Within the pair of parentheses, the *var_type id* is the reference that will be used to point to individual elements during the iteration of the foreach loop, followed by the keyword "`in`", and then an *id*. This *id* is the reference to the list through which the `foreach` loop will iterate. Thus, an error will occur if the type of elements in the list do not match the type of the reference variable, *var_type id*. Enclosed with the curly brackets, there should be at least one newline character, followed by a series of *function_lines*. Errors will occur without one or more newline characters separating the opening curly bracket and the *function_lines*.

Get Each Loop

loop	→	*var_type id* = geteach (*var_type id* in *id* where *expression*)

The other type of specialized loop provided by GAME is the **"geteach" loop**. It is related to the foreach loop, in that it also requires a list, and it will iterate over each element of the list. There are two major differences, however, between the geteach loop and foreach loop.
(1) After the declaration of an element identifier, the specification of the list, the geteach loop has the reserved word "where", and then an expression. This expression must evaluate to a boolean, and to take advantage of the geteach loop's functionality, should utilize the list element identifier and perform a check on it in some way.
(2) A new temporary list is built at the start of the `geteach` loop. For every iteration that the expression in the geteach loop evaluates to true, the element identifier is added to this list. Once iteration is complete, the geteach loop returns the temporary list of elements. Thus, the

programmer must declare a list variable with the same type that can point to the list returned by the geteach loop.

`s

if_statement	→	if (*expression*) { \n *function_lines* } if (*expression*) { \n *function_lines* } else { \n *function_lines* }

If the *expression* evaluates to true, the *function_lines* are executed. Optionally if *expression* is false a second set of *function_lines* may be run that follow else.

Data Statements

JSON files can be employed by GAME programs to load and store data. Collectively, these two actions are represented by *data_statement* in the grammar. A single *data_statement* consists of two reserved words and two expressions, in a particular order.

data_statement	→	load *expression* from *expression* export *expression* to *expression*

Notice that the syntax is meant to approximate natural language. The first word of the *data_statement* indicates whether a load or export action will take place. This is followed by an *expression*, which should be an identifier for a primitive, list, or object variable. The identifier can also be for an object attribute, using dot notation. Then, depending on the first reserved word, the corresponding reserved word, '`from`' or 'to', should be used. Finally, the *data_statement* has a second *expression*, which indicates the filepath that should be followed to retrieve the JSON file for loading or exporting data. For a load statement, GAME will throw an error if the file does not exist. For an export statement, GAME will simply create a new JSON file with the name and filepath specified in the second *expression*.

Loading

data_statement	→	load *expression* from *expression*

One method to initialize variables in GAME is by loading data from a JSON file. There is a precise syntax to use in this process, which is the same for primitives, objects, and lists:

```
load [valid-identifier] from [filepath]
```

The JSON file **must** contain the data necessary to "fill" the variable. Otherwise, GAME will throw a compile-time error.

(1) In the case where the variable to be loaded is a primitive, the JSON should hold the associated data. The JSON data's type must match the GAME variable's type.

(2) In the case where the variable to be loaded is an object, the JSON should hold data to fill **each** of the attributes in the object's class declaration, leaving none of the attributes null. For the attributes that are primitives, the first case applies. If any of the attributes within the class declaration are objects or lists, then the GAME compiler works recursively to fill these attributes as well, applying cases two and three respectively.

(3) In the case where the variable to be loaded is a list, the JSON should return an array of data that matches the list's type. When the list holds GAME primitives, the first case applies. If the list contains objects or more lists, then the GAME compiler works recursively to fill these as well, applying cases two and three respectively.

Note: The first key to the data in the JSON **must** be, "`GAME`". After this, the GAME compiler will use variable identifiers as the keys to retrieve the associated values from the JSON.

For example, suppose there is a `Team` class with two attributes: `yearFounded`, of type `num`, and `mvp`, of type `Player`. The `Player` class has two attributes: `name`, of type `text` and `height`, of type `num`. If the program loads a `Team` object from a JSON file, then GAME will use "`GAME`" as the first key to access data within the JSON file. Then, it will use "`yearFounded`" and "`mvp`" as the keys to access the data needed for the `Team` object. To load the `Player` object, `mvp`, GAME will use "`name`" as the key for the text data and "`height`" as the key for the num data.

This example is to illustrate the importance of keeping the keys of the JSON file consistent with the variable identifiers in the source code, otherwise errors will occur.

GAME code	JSON file

`Class Team {`
`  num yearFounded`
`  Player mvp`
`}`
`Class Player {`
`  text name`
`  num height`
`}`
`main (){`
`  Team miamiHeat`
`  load miamiHeat from [path to JSON file]`
`}`
	`{`
	`  "GAME" : {`
	`  "yearFounded" : 1988,`
	`  "mvp" : {`
	`  "name" : "Lebron James",`
	`  "height" : 2.03`
	`  }`
	`  }`
	`}`

Exporting

data_statement	→	export *expression* to *expression*

Along with the ability to bring a programmer's data into the program code and manipulate it, GAME has the corresponding ability to store this updated data into a JSON file. The process of exporting data into a JSON file will be done according to the same structure delineated in the section for importation.

Break and Continue

statement	→	break continue

The statements break and continue should only be placed within a loop, and they apply to the loop immediately closest to it. This means multiple break statements are needed to escape nested loops. A continue statement halts the current progress of execution through a loop, and returns execution to the top of the loop.

Grammar

lines	→	*lines class_def* \n *lines function_def \n* *lines* \n ϵ
class_lines	→	*class_lines function_def \n* *class_lines variable_def \n* *class_lines \n* ϵ
function_lines	→	*function_lines statement* \n *function_lines* \n ϵ
statement	→	*variable_def* *assignment* *loop* *if_statement* *data_statement*

		obj_expression . *id* (*function_args*) break continue
class_def	→	class *id* { \n *class_lines* } class *id* extends *id* { \n *class_lines* }
function_def	→	function *id* (*function_args*) { \n *function_lines* } *var_type* function *id* (*function_args*) { \n *function_lines* return *expression* \n }
function_args	→	*function_args* , *function_args* *var_type id*
loop	→	loop (*loop_expression*) { \n *function_lines* } foreach (*var_type id* in *id*) { \n *function_lines* } *var_type id* = geteach (*var_type id* in *id* where *expression*)
loop_expression	→	*loop_expression* , *loop_expression* start *variable_def* while *expression* set *assignment* ϵ
if_statement	→	if (*expression*) { \n *function_lines* } if (*expression*) { \n *function_lines* } else { \n *function_lines* }
data_statement	→	load *expression* from *expression* export *expression* to *expression*
expression	→	*expression* + *expression* *expression* - *expression* *expression* * *expression* *expression* / *expression* *expression* % *expression* *expression* > *expression* *expression* >= *expression* *expression* < *expression* *expression* <= *expression* *expression* == *expression* *expression* != *expression* *expression* and *expression* *expression* or *expression* not *expression* (not precedence) - *expression* (minus precedence) *assignment* *obj_expression* . *id* (*function_args*) (*expression*)

		obj_expression *obj_expression* [*expression*] *constant*
assignment	→	*id* = *expression*
obj_expression	→	*obj_expression* . *id* *id*
variable_def	→	*var_type id* *var_type assignment* *var_type id* = new *var_type* *var_type id* = new *var_type* { \n *mul_variable_def* }
mul_variable_def	→	*mul_variable_def variable_def* \n *variable_def* \n
var_type	→	text num bool *id* list (*var_type*)
constant	→	{ *constant* } *constant* , *constant* *num_constant* *txt_constant*
num_constant	→	r'\d+\.?d*'
txt_constant	→	r'"[[^"][\\"]]*"' (string constants surrounded by double quotes, \ is escape character)
id	→	r'[a-zA-Z_][a-zA-Z0-9_]{0, 99}'

Operator Precedence

	Type	Associativity	Precedence
.	binary	left	1
- not	unary	right	2
* / %	binary	left	3
+ -	binary	left	4
> >= < <=	binary	left	5

== !=	binary	left	6
and	binary	left	7
or	binary	left	8
=	unary	right	9
,	binary	left	10