Language Tutorial

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

Section 1.0

GAME is designed to solve problems related to sports statistics as well as sports management. Today, decisions in the world of sports increasingly need to be backed by data and quantitative analysis, rather than stemming from purely qualitative assessments. This is important for athletes, coaches, and managers in improving their individual and team performance in addition to sports reporters seeking compelling data to back their stories. Thus, a lot of money and many careers are involved in the practice of data analytics in sports. Accessing, manipulating, updating, and analyzing data are essential in the pursuit of game-changing information -- GAME allows users to accomplish this objective.

GAME is unique in that it is specifically geared towards the manipulation of sports statistics. Like R, it will provide various statistical measures for the purpose of data analysis and will allow users to define additional ones. More than that, it will provide built-in object-oriented functionality, including sports-specific classes such as Player and Team that will allow users to develop their programs quickly and easily. GAME provides automatic conversion from data provided in specified file types -- JSON and XML -- to an internal, object-oriented representation of the data. The same service will be provided in the reverse direction, so that data can be exported to such files. GAME will make it extremely simple to examine portions of a dataset that are most relevant to the user through efficient data retrieval. For example, the Moment type will allow users to record noteworthy events during a game, and a list of such events can compose a chronological history (or “timeline”) of the game. Our language provides a unique combination of statistical tools; easy data access and manipulation; and sports-specific concepts in an object-oriented framework.

Section 1.1: Getting Started

According to Kernighan and Ritchie, the “only way to learn a new programming language is by writing programs in it.” This introduction teaches the user to create programs, compile them, and run them successfully in GAME. The first program is the same for all languages:

Print the words
hello, world

In GAME, the program to print “hello, world” is:

```
include “stdlib”

function main() {
	print(“hello, world!”)
}
```

The command **gamec program.game** compiles the source code. This produces the file

“program”. To run the program, the command is **game program**.

A GAME program consists of functions and variables. Functions contain statements that specify the computing operations to be done and variables store values used during computation. In the above example, main is the function. Functions can typically be named by the user, but *main* is different because the program is executed inside of it. main can call other functions -- some written by the user and others by libraries already present in the system.

The first line of the program `include “stdlib”` tells the compiler to include information about the standard functions and classes provided within the language. The print method is a library function that outputs information; in this case, it is used to output “hello, world!” to the user. Two of the primary primitives in the GAME language are “text” and “num”. The text primitive refers to a string of characters and num refers to a number, which is represented as a floating point value by default. The print method can be used to output numbers as well. In particular, the num_form function allows the user to specify the format of the number to be printed. For example, the following program prints out variations of the number 100:

```
include “stdlib”                  include information about standard library

function main() {                 define a function named main
       num j = 100                declares a variable named j storing 100
       text jText = num_form(“#”, j)     declares a variable named jText that
                                         prints the value of j before the decimal
                                         place: 100

       print(j)                   prints the internal representation “100.0”
       print(jText)               prints the number “100”
       print(num_form(“#.##”, j)) prints with two decimal places: “100.00”

       print(“Number: ” + num_form(“#.##”, j)) prints out “Number: 100.00”

}
```

This example shows how the print method can output variations of numbers in addition to text. This is done through the standard library function num_form(). Here, the first parameter specifies the format. The format “#” prints out the number before the decimal place, so 100.0 is printed as 100. Similarly, the format “#.##” prints out the number with two decimal spots, so it is printed as 100.00. It also shows string concatenation between text and numbers using “+”. Combining a text variable with a num variable using this operator will convert the entire expression into a text variable.

Section 1.2: Variables and Arithmetic Expressions

The next program calculates the average number of points per game scored by the basketball player Michael Jordan throughout his career.

```
include “stdlib”

function main() {
	num points
	 num games
	 num average

	points = 38279
	games = 1264
	average = points / games

	print(“Average Points Per Game by Michael Jordan: ” + average)
}
```

The lines “num points, games, average” declares three variables of type num: points, games, and average. These are used to store numbers. The lines “points = 38279” and “games = 1264” bind specific values to the associated variables; this is called initializing the variables. The statement “average = points / games” divides these values and stores the result in the variable average. The print statement outputs

“Average Points Per Game by Michael Jordan: 30.284018973417721518987341772 15”

Some languages may cast the result of division of integers into an integer value, but the result of division in GAME will be the actual value. Then, this can be converted into an integer using num_form(“#”), which will return the floor of the result.

Section 1.3: Loop Statements

There are several ways to write a program. For example, given the task of printing the sum of the numbers from 1 to 100, we can write the following programs:

```
# First Program
include “stdlib”

function main() {
	num num1 = 1
	num num2 = 2
	num num3 = 3
	…
	num num100 = 100

	print(“Total: ” + num1 + num2 + num3 + … + num100)
```

```
}

# Second Program
include “stdlib”

function main() {
	num sum = 1 + 2 + 3 + … + 100
	print(“Total: ” + sum)
}
```

Both of these programs accomplish the same task, but they require the user to type out each number, which is tedious and, more importantly, makes the program prone to mistakes in the calculation. A different way of writing this method is through the use of a **loop**:

```
include “stdlib”

function main() {
	num sum = 0

	loop (start num i = 1, while i <= 100, set i = i + 1) {
		sum = sum + i
	}

	print(“Total: ” + sum)
}
```

This program is considerably shorter than the first two programs because it eliminates the intermediate variables and is less error-prone since the calculations are done within the loop expression. In this code, sum is initialized to 0 and the loop iterates through every number between 1 and 100 and adds it to sum. The sum variable is updated with the new value after each computation.

The loop structure is organized into three parts: initialization, while condition, and set statement. The initialization portion allows users to define variables (and their starting values) for use within the code block. The while statement specifies the condition that will be checked at each iteration: The program will continue to run until the condition is false. The set statement is used to increment or update a variable. These three parts are written with the start, while, and set keywords, respectively. The loop statement is flexible and combines the functionality of *for* and *while* loops within C. The user can use as many start, while, and set statements inside of the loop. For example, summing together the numbers from 1 to 100 and every even number less than or equal to 200 can be accomplished:

```
include “stdlib”

function main() {
	num sum = 0

	loop (start num i = 0, start num j = 0, while i <= 100 and j <= 200, set i
		= i + 1, set j = j + 2) {

		sum = sum + i + j
	}

	print(“Total: ” + sum)
}
```

In this program, the variables sum while the variables i and j are initialized to 0 within the initialization statement. The while condition contains an **and** operator that causes the loop to run as long as both i and j are less than or equal to 200. Similarly, the **or** and **not** operators can be used inside the while condition as well. The loop’s set statements increment i by 1 and j by 2.

It is possible to omit one or more of a loop’s three components. Thus, the user can achieve the same result using variations on the loop statement:

```
include “stdlib”

function main() {
	num i = 0
	num sum = 0

	loop (while i <= 100) {
		sum = sum + i
		i = i + 1
	}

	print(“Total: ” + sum)
}
```

In the above program, the same goal of summing the numbers from 1 to 100 is achieved, but the *start* and *set* components of the loop have been left blank. This is possible because the initialization of the variable i was done before the loop statement and the incrementing of i is done inside the loop body.

Another way of writing this program is through **if** statements and **break**. The **if** statement makes it possible to execute a line or a block of code only when a specific condition holds. Following the if block, the user can provide an optional **else** statement, which will execute in the case when

the condition for the if statement evaluates to false. The **break** statement allows the user to exit a loop under certain circumstances. It is always used in conjunction with an if statement. From this, we can write the previous program in another way:

```
include “stdlib”

function main() {
	num i = 0
	num sum = 0

	loop () {
		sum = sum + i
		i = i + 1
		if (i > 100) {
			break
		}
	}

	print(“Total: ” + sum)
}
```

Note that the loop statement has no while condition specified. This can be referred to as an “infinite loop,” which will loop forever unless something within the loop body causes the loop to halt. This is what the break statement is used for: If i evaluates to more than 100, the loop will stop. This program is equivalent to the previous examples, but cleaner code would typically use a more standard loop statement with a specified while condition; however, there are certainly instances where a break statement can be useful.

Given the various implementations of the same program using a loop, the following serves as a rough guide on preferred design: The most straightforward approach typically contains only one while statement (using “and” and “or” operators to combine multiple conditions) in addition to a minimal number of start and set statements (using additional variables and if statements within the loop body if necessary).

Section 1.4: Lists

It is often helpful to organize data into a single data structure such as a list. This allows the user to group related objects, which can then be iterated in a convenient manner.

```
include “stdlib”

function main() {
	list(num) points_scored = { 15, 9, 27 }

	foreach (num i in points_scored) {	prints out “15 9 27”
		print(num_form(“#”, i) + “ ”)
	}

	points_scored.rem(9)			removes “9” from points_scored
	points_scored.add(22)			adds “22” to points_scored

	foreach (num i in points_scored) {	prints out “15 27 22”
		print(num_form(“#”, i) + “ ”)
	}

}
```

In this example, the list methods **rem([type] item)** and **add([type] item)** are used. rem() removes all instances of the item from the list, while add() appends the specified item to the back of the list. In addition to these methods, the list type supports **addAt(num index, num item)**, **remAt(num index)**, and **get(num item)**.

The *foreach (num i in points_scored)* statement is specifically designed for use with lists. It simply iterates through all objects of type *num* in the list *points_scored*, and in each iteration the variable i refers to the next element in the list. The following construct:

```
foreach (num i in points_scored) {
	print(num_form(“#”, i) + “ ”)
}
```

can be written equivalently as:

```
loop (start num i = 0, while i < length(points_scored), set i = i + 1) {
	print(num_form(“#”, points_scored[i]) + “ ”)
}
```

This demonstrates the notion of “random access” in lists: An element at position i of the list can be accessed using “mylist[i]”. The following example starts with a list and arrives at a sublist of the initial list, containing only the data that we are interested in -- in this case, all instances where

points_scored is greater than 20:

```
include “stdlib”

function main() {
	list(num) points_scored = { 15, 9, 27, 22, 13 }
	list(num) good_games = { }

	foreach (num i in points_scored) {		prints out “15 9 27”
		if (i > 20) {
			good_games.add(i)
		}
	}

	foreach (num i in good_games) {			prints out “27 22”
		print(num_form(“#”, i) + “ ”)
	}

}
```

Since this is a common task in data analytics, GAME provides a more convenient way of achieving the same thing using a **geteach** statement:

```
include “stdlib”

function main() {
	list(num) points_scored = { 15, 9, 27, 22, 13 }

	list(num) good_games = geteach (num i in points_scored where i > 20)

	foreach (num i in good_games){			prints out “27 22”
		print(num_form(“#”, i) + “ ”)
	}

}
```

The **geteach([type] [var] in [list] where [condition])** statement allows the user to generate a new list from [list], where the new list contains only those elements of [list] that meet the [condition].

Section 1.5: Functions

GAME allows users to define functions in order to promote organization and reuse of their code. The following program defines a function that takes in a list representing the number of points scored in different games and outputs the number of “good games,” defined according to the rule above.

```
include “stdlib”

num function countGoodGames(list(num) games) {
	list(num) good_games = geteach (num i in games where i > 20)
	return length(good_games)
}

function main() {
	list(num) points_scored = { 15, 9, 27, 22, 13 }
	num count = countGoodGames(points_scored)
	print(“Number of good games: ” + count)
}
```

The parameters are the inputs to the function: In the function definition, the user must specify the type of each parameter and separate each parameter with a comma. The return type is the output of the function -- its type is specified and immediately follows the keyword “function.” It is possible for a function to have no parameters and no return type. Here, the function countGoodGames() has a single parameter of type list. It returns an object of type num. In this example, the main function simply obtains the value returned by the function call and prints it out. If there is nothing else to be done with the variable count, we could have written the function so that it prints out the number in question, without returning anything, as follows:

```
include “stdlib”

function countGoodGames(list(num) games) {
	list(num) good_games = geteach(num i in games where i > 20)
	print(“Number of good games: “ + length(good_games) )
}
function main() {
	list(num) points_scored = { 15, 9, 27, 22, 13 }
	countGoodGames(points_scored)
}
```

Section 1.6: Classes

GAME is an object-oriented language, meaning the user can accomplish tasks through the manipulation and interaction of “objects.” Objects are entities that can consist of both attributes and operations (or methods). A “class” serves as a blueprint for objects by defining their attributes and operations: Objects are said to be “instances” of a class. To create an object, the user “instantiates” a class. Below, we create a class called “Player”:

```
class Player {
	text first
	text last
	num points_per_game
	list(num) points_scored

	function addGame(num points) {
		points_scored.add(points)
		points_per_game = average(points_scored)
	}

	function build(text f, text l, num ppg, list(num) p) {
		first = f
		last = l
		points_per_game = ppg
		points_scored = p
	}
}

num function average(list(num) games) {
	num result
	foreach (num i in games) {
		result = result + i
	}
	return result / length(games)
}
```

An object of type Player thus has four attributes: two text fields corresponding to a Player’s first and last names, a list of numbers corresponding to the number of points scored by the player in each game, and a number corresponding to the average number of points scored. The Player class also defines a method called addGame(), which inserts a new number into points_scored and updates the points_per_game attribute. Using this method to augment point_scored ensures that points_per_game will always be up to date. Note that the addGame() method calls another method, average(), that has been defined outside of the class.

In addition to addGame(), the Player class defines a method called build(). This is a special method that can be considered a “constructor” for the Player class -- that is, a method that

facilitates the initialization of attributes for a new Player object. After creating a new Player object, the user can simply call the build function with the appropriate parameters, and all the Player's attributes will be initialized. (This is shown below.) This tutorial will demonstrate other ways to instantiate an object, showing the build function is not strictly necessary; however, if such a "constructor" function is desired, it is strongly recommended to adhere to the convention of calling it "build" for consistency.

We can now instantiate the Player class in our main method. We show three different ways of creating an object of type Player:

```
include "stdlib"
include "player.game"

function main() {
	Player lebron = new Player
	lebron.first = "Lebron"                    method 1: initialize each variable
	lebron.last = "James"
	lebron.points_scored = { 25, 33, 22, 17, 28 }
	lebron.points_per_game = average(lebron.points_scored)

	Player dwyane = new Player
	list(num) dwyane_points = { 22, 18, 12, 29, 16 }

                                               method 2: use "build" function
	dwyane.build("Dwyane", "Wade", dwyane_points, average(dwyane_points))

	Player chris = new Player {                method 3: initialize with brackets
		first = "Chris"
		last = "Bosh"
		points_scored = { 11, 24, 15, 26, 21 }
		points_per_game = average(chris.points_scored)
	}

	list(Player) miami_heat = { lebron, dwyane, chris }
	foreach (Player i in miami_heat) {
		print(i.first + "scores " + i.points_per_game + " points per game.")
	}
}
```

We can expand on this example by defining another class called Team:

```
include “player.game”

class Team {
	list(Player) roster
	text city
	text name
}
```

And in the main function, after creating the three players lebron, dwyane, and chris:

```
include “stdlib”
include “player.game”
include “team.game”

function main() {
	…
	list(Player) miami_heat = { lebron, dwyane, chris }

	Team heat = new Team {
		roster = miami_heat
		city = “Miami”
		name = “Heat”
	}

	println(“The Miami Heat roster”)		println adds a newline at the end
	foreach (num i = 0 in heat.roster){
		println((heat.roster[i]).first + “ ” + (heat.roster[i]).last)
	}
}
```

Section 1.7: Major Standard Libraries and Classes

One of the main features of GAME is the selection of built-in classes and methods that are designed to greatly facilitate the construction of sports-related programs. While later versions of the language will certainly cater to a wide array of sports, this tutorial will initially focus on providing utilities for the sport of basketball.

Note that the user is not required to fill in all fields in order to use the below classes: If there are some fields that the user would rather not track, he or she can simply initialize them to **null**. Moreover, GAME makes it possible to extend these classes, if the user would like to add further fields and methods to the classes. These classes are based on the set of statistics that are typically tracked in box scores.

BasketballGame class *corresponds to a single basketball game played*

fields:

```
BasketballTeam home
BasketballTeam away
text date
BasketballPerformance home_team          team stats
BasketballPerformance away_team
list(BasketballPerformance) box_score    box score of the game:
                                         each player's stats
list(BasketballMoment) timeline          chronological timeline,
                                         consisting of noteworthy moments from the game
```

methods:

```
BasketballTeam winner()                  returns winner of the game
```

BasketballTeam class

fields:

```
list(BasketballPlayer) roster
text city
text name
list(BasketballGame) games_played
```

BasketballPerformance class *corresponds to a player or team's statistical performance*

fields:

```
text player_or_team              identifier for specific individual or team
num points
num rebounds
num assists
num steals
num blocks
num turnovers
num fg_attempted                 field goals
num fg_made
num ft_attempted                 free throws
num ft_made
num thr_attempted                three-pointers
num thr_made
```

methods:

```
num fg_percentage()
num ft_percentage()
num thr_percentage()
```

BasketballPlayer class

```
fields:
       text first_name
       text last_name
       BasketballTeam team
       num height
       num ppg                          abbreviations correspond to the stats in the
                                        Performance class, "per game"
       num apg
       num rpg
       num spg
       num bpg
       num tpg
       num fgapg
       num fgmpg
       num ftapg
       num ftmpg
       num thrapg
       num thrmpg
       list(BasketballPerformance) performances
       list(BasketballGame) games_played
methods:
       num fg_percentage()
       num ft_percentage()
       num thr_percentage()
       build(list(BasketballPerformance))      create Player from list of performances
(the averages)
```

BasketballMoment class

```
fields:
       num minute                minute from start of game, at which moment occurred
       text event                brief descriptor of the moment, such as "turnover"
                                 (allows for easy categorization)
       text description          more thorough description of the event
       list(Player) players_involved
```

Finally, in the stdlib.math file, there are the functions:

- mean(list(num))
- median(list(num))
- sqr(num)
- sqrt(num)
- stdev(list(num))
- min(list(num))
- max(list(num))
- floor(num)
- ceiling(num)

Note: in Section 1.6, we defined a function called average(list(num)). This is equivalent to the stlib.math function mean(list(num)).

Section 1.8: File Reading and Writing

One of GAME's significant advantages over other languages is the straightforward way that users can input data from a JSON file. Existing statistical programs, such as MATLAB and R, require users to have a deep understanding of each language in order to accomplish this task. The GAME language will take in a JSON or XML data file and populate the appropriate values from the file. In Section 1.6, we discussed three ways of creating an object of a class; using the **load** function, the GAME language introduces a fourth way of accomplishing this. Class structures will need to be defined by either the user or follow the classes defined in the standard library. Here, we will use the Team class from earlier:

```
include "player.game"

class Team {
	list(Player) roster
	text city
	text name
}
```

To create a new Team, we can use a **load** statement, which is written as:

load list(object_type) list_identifier from "[absolute/relative filepath]"

The following example will load the players for a team using an input file called knicks.json:

```
include "stdlib"

function main() {
	Team knicks = new Team {
		load knicks.roster from "knicks.json"
		knicks.city = "New York"
		knicks.name = "Knicks"
	}
}
```

This example loads the roster from the file called knicks.json, which is present within the same directory as the GAME program. It will look through the file and will create an object for each player before placing it inside the roster for the new Team instance. Another way of writing this method is by using the load command on a new list variable and assigning the roster to be this list:

```
include “stdlib”

function main() {
	load list(Player) players from “knicks.json"
	Team knicks = new Team {
		knicks.roster = players
		knicks.city = “New York”
		knicks.name = “Knicks”
	}
}
```

Since modifications are likely to be made to attributes of classes through programs, the GAME language also allows users to export the resulting file. This can be done through an explicit method called

export [list] to [filename of resulting JSON file]

For now, only JSON files are accepted as input files, but the next version of the GAME language will likely support additional file types. There are external programs that can convert file types into a JSON file and several tutorials that explain their straightforward organizational scheme.

Section 1.9: Advanced Features

The examples within this tutorial have only created classes within the same file. While this is acceptable, it may be more helpful to separate code into multiple files to prevent a single file from becoming too long. This can be done using the *include* statement seen throughout this tutorial. Similar to C, this is a way to include the code from a different file and effectively links multiple files. For example, say there are two files called program1.game and program2.game. program1.game contains a class called YouthBasketballTeam and program2.game has a class called YouthBasketballPlayer. The code for program1.game can contain the following:

```
include “program2.game”

class YouthBasketballTeam {
	list(YouthBasketballPlayer) roster;
	text name;
	text county;
	text state;
}
```

Even though YouthBasketballPlayer is not defined within program1.game, it can still use instances of this class due to the “include program2.game” line. This statement allows program1.game to use classes defined within program2.game and prevents a compiler error

from being thrown due to an unknown class type. As a result, the methods of these classes can be used as seen earlier.

The GAME language also supports graphing. The standard library will contain the methods **graph()**, **label()**, and **display()**. A user can use it by writing putting *include stdlib.graph* at the top of the program:

```
include “stdlib.graph”

function main() {
	load list(Player) players from players.json
	list(num) heights
	list(num) fgpercent

	loop (start num i = 0, while i < length(players), set i = i + 1) {
		heights.add(players[i].height)
		fgpercent.add(players[i].fg_percent())
	}

	graph(heights, fgpercent)		graphs height on x-axis,
						field goal percent on y-axis

	label(“x”, “Height”)			label x-axis
	label(“y”, “Field Goal Percentage”)	label y-axis
	label(“t”, “Height vs. Field Goal Percentage”) label the graph’s title
}
```

The format of the graph method is **graph([list of x parameters], [list of y parameters])**; the format of the label method is **label([x, y, or t], [name])**, where x is the x-axis, y is the y-axis, and t is the title of the graph; and the format of the show method is simply **display()**. The graph method will plot the corresponding parameters from each list (x, y) as Cartesian coordinates; the label method will name the axes and/or title; and the show method will generate and display the graph.

Section 1.10: Clarification

Throughout this tutorial, all of the examples have been focused on basketball; however, this was done to maintain a consistent context for teaching different aspects of the language without confusing the user with the details of different sports. These examples can easily be adapted to other sports since they feature broad principles that can be modified to fit the user’s needs. The GAME language is built to be an all-purpose language that supports any sport -- from international soccer to American football to a local wiffleball league. Specifically, GAME is intended to be a general sports language that is useful to a wide range of individuals.