Tutorial

1 Intro

In this section, we will present a series of programs that demonstrate the features of the programming language. By reading this language tutorial, the user should have sufficient knowledge of the various functions the BALL language provides. BALL is a concise language inspired by Java and the game of baseball. If the user is familiar with both baseball and Java, the language should be relatively easy to learn and master.

This tutorial will show the user how to simulate games and even create their own simulation function. It will also show the user how to make simple programs that simply print stats of teams or players. With an understanding of how to write the following programs, writing more complex programs should be easy for the average user (assuming the user is familiar with certain programming constructs used in C and Java).

2 Simple Hello World

2.1 Program construction

Like all other languages, the first program to learn is a simple printing program. While most languages use the phrase “hello world,” the first program in BALL will print “play ball.” The program will look like the following:

```
print “play ball”;
```

2.2 Basic Compilation

Because our language is Java-based, running the program is system independent. To compile the program, run the command `ball` on the file `hello.ball` which is the name of the file created above.

```
$ ball hello.ball
```

If you have followed our tutorial correctly, the compiler should proceed and print the intended output, in this case:

```
play ball
```

In BALL, the print function is predefined and gives the user the ability to print any given data.

3 Simulating Hello World

Let's now look at a more complex program. The following program shows how to define a new sim function and then use it in simulating games.

3.1 Defining a Simulation Function

```
/* a more complex program */

simfunction simpleSim is:

  if (team1's W > team2's W) then:

    return team1;

  else:

    return team2;

  end

end


activate simpleSim; //activates the simpleSim function


team Indians = load("Indians.team");

team Orioles = load("Orioles.team");

print "Winner: " + sim(Indians, Orioles, 1);
```

This application can be considered as a program in three parts. Part 1 is the definition of the simulation function. Part 2 is the activation of said simulation function and part 3 is the printing and evaluation section.

Part 1 of the program starts with a comment. In ball comments are defined by the leading `//`. Any characters between the `//`and the next line are ignored by the compiler. Similarly `/* */` defines comments but `/* */` allows the user to have comments that span multiple lines. Comments can appear anywhere a blank or tab or newline can. The next line of the program defines a simulation function naming it `simpleSim`. The function takes two parameters of type team. In the BALL language, the simulation function takes two teams as parameters and returns the *winning* team. In the same line, `is:` is used like `{` is used in other programming languages but the keyword `is` is used only following the simulation function.

3.2 Conditionals

The next line uses a conditional statement called `if`. The `if` statement is a construct used in most programming languages, and in BALL it behaves the same as it does in C or Java. It evaluates the statement within the `( )` and based on its return value it executes either the following lines before the `else:`, or the lines following the `else:`. In this case, it either returns the first team, `team1` or the second one, `team2. team1` and `team2` are implicit definitions automatically understood by the compiler when inside of a simfunction declaration. They do not need to be explicitly defined.

Part 2 of this program uses the reserved keyword `activate` to tell the compiler which simulation function to use.

Lastly, the third part of this program creates two teams named `Indians` and `Orioles`. For each team, it calls the function `load` which uses a file specified by the user in string format. Lastly the program prints the winner of the simulation preceded by the string `“Winner: ”`.

3.3 Team Files

The files that `load` uses are of the following format:

```
Team Name: Houston Astros,Houston,Astros
Type:team stats
Header:W,L
74,88
Type:Batter
Header:Name,AB,R,H,2B,3B,HR,BB
Ivan Rodriguez,327,41,82,15,2,8,13
```

```
Lance Berkman,460,73,126,31,1,25,97
Kazuo Matsui,476,56,119,20,2,9,34
Miguel Tejada,635,83,199,46,1,14,19
Geoff Blum,381,34,94,14,1,10,33
Carlos Lee,610,65,183,35,1,26,41
Michael Bourn,606,97,173,27,12,3,63
Hunter Pence,585,76,165,26,5,25,58
Wandy Rodriguez,63,4,8,3,0,0,1
Roy Oswalt,49,4,6,0,0,0,2
Brian Moehler,42,0,1,0,0,0,1
Mike Hampton,37,6,12,1,0,1,2
Russ Ortiz,28,2,5,0,0,1,1
Felipe Paulino,25,0,1,0,0,0,0
Type:Pitcher
Header:Name,IP,K,H,BB,ER
Wandy Rodriguez,205.2,192,69,63,193
Roy Oswalt,181.1,183,83,42,138
Brian Moehler,154.2,187,94,51,91
Mike Hampton,112.0,128,66,46,74
Felipe Paulino,97.2,126,68,37,93
Russ Ortiz,85.2,95,53,48,65
```

Essentially, the file is composed of comma separated values named by the tags `header` and `type`. So in this case, the first line specifies the name of the team. It then specifies the team-wide stats `W` and `L`. The next section is tagged by `Header` and names what each of the values defined in the following lines are, in this case, `Name,AB,R,H,2B,3B,HR,BB`. All ".team" files must be of this special format.

3.4 Calling `sim`

`sim` is a function that takes three arguments, two teams and a number. Sim then returns a team based on the calculations done. Though the `sim` can only take teams and a number, because it returns a team it can also take nested `sim` functions. That functionality will be discussed in the following sections.

4 A More Complex Simulation

4.1 Basic Definitions

This section will examine a more complex simulation. In the following program, the user defines stats based on given stats defined in the `.team` file. Furthermore, the program defines functions that will be used in the new simulation function. It then computes the simulation. This will be a good model of a somewhat accurate simulation function.

```
/***STATS DEFINITION***/

/*Batter probabilities*/

stat bWalk = BB / PA;

stat bSingle = (Hits - (2B + 3B + HR))/ PA;

stat bDouble = 2B / PA;

stat bTriple = 3B / PA;

stat bHR = HR / PA;


/*Pitcher probabilities*/

stat pWalk = BB / BF;

stat pSingle = (Hits / BF) - pHR - pTriple - pDouble;

stat pDouble = Hits x .174 / BF;

stat pTriple = Hits x .024 / BF;

stat pHR = HR / BF;


/***END STATS DEFINITION***/


/*Global variables that hold combined probabilities given a pitcher and
a batter*/

number probWalk = 0;

number probSingle = 0;
number probDouble = 0;

number probTriple = 0;

number probHR = 0;


/*BASIC SIMULATION FUNCTION*/

simfunction basicSimulation is:

  number team1Score = 0;

  number team2Score = 0;


  /*Note: team1 and team2 are global variables that represent the teams
that the sim function is currently working with. The sim function sets
them automatically.*/


  /*team1 is batting and team2 is pitching! (5 times)*/

  do 5 times:

    /*Select random players from the best 9 of each team*/

    list bestBatters = top(9, team1 where (type is "batter"), AVG);

    player batter = any bestBatters;


    list bestPitchers = top(4, team2 where (type is "pitcher"), AVG);

    player pitcher = any bestPitchers;


    /*Set the global probabilities of this inning*/

    combineProb(batter, pitcher);
    /*Add a little randomness*/

    randomizeProbabilities();


    team1Score += probWalk*(1/4) + probSingle*(1/4) + probDouble*(2/4)
+ probTriple*(3/4) + probHR;

  end


  /*team2 is batting and team1 is pitching! (5 times)*/

  do 5 times:

    bestBatters = top(9, team2 where (type is "batter"), AVG);

    batter = any bestBatters;


    bestPitchers = top(4, team1 where (type is "pitcher"), AVG);

    pitcher = any bestPitchers;


    combineProb(batter, pitcher);

    randomizeProbabilities();


    team2Score += probWalk*(1/4) + probSingle*(1/4) + probDouble*(2/4)
+ probTriple*(3/4) + probHR;

  end


  if (team1Score > team2Score) then:

    print "Team " + team1’s name + " wins against team " + team2’s name
+ "!!";
```

```
    return team1;

  else:

    print "Team " + team2’s name + " wins against team " + team1’s name
+ "!!";

    return team2;

  end



end

/*END SIMULATION FUNCTION*/



/*Declaring a general-purpose function that calculates combined
probabilities given a pitcher and a batter. It takes the max of both:
average is not a good estimate.*/

function combineProb (player batter, player pitcher) returns nothing:

    probWalk = max(pitcher’s pWalk, batter’s bWalk);

    probSingle = max(pitcher’s pSingle, batter’s bSingle);

    probDouble = max(pitcher’s pDouble, batter’s bDouble);

    probTriple = max(pitcher’s pTriple, batter’s bTriple);

    probHR = max(pitcher’s pHR, batter’s bHR);

end



/*General Purpose function that takes the values generated by
combineProb and randomizes them (gives them 10% of jimmy-jammy-ness)
for more realistic results*/

function randomizeProbabilities() returns nothing:

    probWalk = rand(probWalk - probWalk*.1, probWalk + probWalk*.1);
    probSingle = rand(probSingle - probSingle*.1, probSingle +
probSingle*.1);

    probDouble = rand(probDouble - probDouble*.1, probDouble +
probDouble*.1);

    probTriple = rand(probTriple - probTriple*.1, probTriple +
probTriple*.1);

    probHR = rand(probHR - probHR*.1, probHR + probHR*.1);

end


activate basicSimulation;


team winner = sim( sim(Astros, Dodgers, 3), sim(Orioles, Twins, 3), 3);

print "Team " + winner’s name + " has won the championship!!\n";


print "Team " + winner’s name + " has the players:";

foreach p in winner do:

    print p’s name;

end
```

The first part of this program labeled “Stats Definition” in the comments does just that. It creates new stats which are all based on some combination of stats defined in the `.team` files. If a stat being created references a stat that is not defined in either the program or the `.team` file an error will be thrown.

The next section declares global variables that can be accessed by all functions in the program. In BALL all variables must be declared before they are used. Generally, this is done before the beginning of a function or executable statements. Any declaration consists of a type name and a list of variables. In the case of this program it is only declaring one variable at a time, but the section:

```
number probWalk = 0;
```

```
        number probSingle = 0;

        number probDouble = 0;
```

could be rewritten as:

```
        number probWalk = 0, probSingle = 0, probDouble = 0;
```

The type, `number`, tells the compiler that the following variables listed are of that type. Furthermore, it assigns the given value specified by the equals operator.

The basic simulation function is the key to the all programs created in BALL. In this case, the simulation function acts as though the game is simply a matchup of a pitcher versus a batter and the batters rotate. Furthermore, it simplifies scoring to be a summation of the hits values (where singles are worth 1/4, doubles are worth 1/2, triples are worth 3/4 and homeruns are worth 1).

1.4.2 Do Loops

In this function the `do` loop is introduced. It is a generalization of Java's `while` loop. In BALL the `do` loop will execute the code following it (until it reaches the `end` associated with that statement) the amount of times specified by the number immediately following the word `do`. As with Java's while loop, the body of the loop could be a single line or an arbitrarily long block of code enclosed by the `: end` constructs (similar to Java's "{" "}"). However, the `do` loop can be programmatically aborted by calling the keyword `stopdo;` which exits the loop and continues with the next section of code. This is similar to Java's `break;` command.

1.4.3 Lists

Next, the program creates a `list` of players, in this case the list composed of the sublist created by calling `top` which is a built in function that takes three parameters, the first is a number that will be the number of items in the list. The second parameter is a list and the third parameter is a stat.

```
    list bestBatters = top(9, team1 where (type is "batter"), AVG);

    player batter = any bestBatters;
```

These two lines set the `batter` to any batter in the list of batters. In this case, `any` is a keyword that selects a random item from a list.

1.4.4 Defining Functions

The user defined function `combineProb` is called taking two arguments in this case players.

```
combineProb(batter, pitcher);
```

Within the program, this function has not yet been defined. This is a feature of BALL, the user can define functions at any point in the program. The function is defined in the file and is the following:

```
function combineProb (player batter, player pitcher) returns nothing:
    probWalk = max(pitcher’s pWalk, batter’s bWalk);
    probSingle = max(pitcher’s pSingle, batter’s bSingle);
    probDouble = max(pitcher’s pDouble, batter’s bDouble);
    probTriple = max(pitcher’s pTriple, batter’s bTriple);
    probHR = max(pitcher’s pHR, batter’s bHR);
end
```

The first line defines the function which in this case takes three parameters and “returns nothing.” In BALL all function definitions must be of the following form:

`function` *name* `(arg1, arg2, ...) returns` *type* `:`

In the case of `combineProb` there are two parameters and nothing to return. In BALL, `nothing` is a keyword that maps to Java’s `void`. In `combineProb` the function `max` is called. Just as `top` and `bottom` are called above, `max` is a built in function that takes two parameters of type `number` and returns the higher number. Likewise, `min` has similar functionality except it returns the smaller of the two numbers. In `combineProb` the global variables defined at the beginning of the program are all modified.

After calling the functions, the program then compares the values `team1Score` and `team2Score` and prints the name of the team associated with the higher value.

1.4.5 Calling `sim`

Before calling the sim function, it is essential that the user defined function has been activated. So the program calls

```
activate basicSimulation;
```

which tells the compiler which sim function to use. Only after defining an active simulation function can the `sim` function be called. Furthermore, `sim` functions can be nested like the following:

```
    team winner = sim( sim(Astros, Dodgers, 3), sim(Orioles, Twins, 3), 3);
```

That will return the winner of a simulation between the winners of two different “match ups.”

1.5 Conclusion

The tutorial has now covered the core of BALL’s functionality. With the building blocks provided, the user can create interesting stats which can be used in innovative simulation functions. It is now possible for the user to write long and interesting programs that use user defined stats and simulations to output information of value. The following exercises suggest BALL programs of greater complexity than the ones described earlier.

1.6 Exercises

Exercise 1.6.1: Write a program that uses more than one simulation function and prints the winners of a 5 game series based on each sim function.

Exercise 1.6.2: Write a program that takes the simulation functions defined either by the previous program or the tutorial and simulate an entire season of any given team. Compare the results to the actual seasonal results.

Exercise 1.6.3: Write a program that simulates an entire season for all teams and then selects the top 8 teams and simulates Playoffs and World Series, returning the winner of the entire season, and congratulating them on their victory.