# Language Reference Manual

## 1 Preamble

Herein is the language reference manual for the BALL programming language. BALL is a language designed for simulating baseball games, as well as allowing a programmer to access baseball statistics in order to create their own simulations.

### 1.1 Objectives

The main objective of BALL is to allow users to manipulate simulations of baseball games by creating their own simulation functions to interact with teams. BALL's language constructs are primarily centered around this goal, as well as minor 'helper' objectives, such as allowing statistical data to be viewed and compared. Another objective of BALL is to make simulation programming simple and straightforward, allowing the user to accomplish the necessary tasks without having to worry about obscure imports and packages.

### 1.2 Abstract Features

#### *1.2.1 Types*

BALL is a typed language, specifically with five types, `number`, `string`, `list`, `team`, and `player`. The `number` and `string` types are simple constants. This is further explained in the section 2.3 below. The type `list` is a collection of objects, which are instances of types. Thus it is possible to have a `list` of `team` objects, `player` objects, `string`s, `number`s, or other `list` objects. `list` objects are homogenous, meaning they cannot contain objects of multiple types. The `player` type is an object that has attributes of `string` and `number` types. These attributes include the name of the player, whether the player is a pitcher or not, and the statistics of that player. The `team` type is an object that has attributes of `string`, `number`, and `list` types. These attributes include the name of the team, the team's wins and losses, and the list of every player on the team.

#### *1.2.2 Language Features*

BALL is a compiled language. Source code is stored in files ending with the .ball extension (for example: `hello.ball`), and is provided to the BALL compiler, which translates the program into intermediate

code (using Java). The compiler then compiles this intermediate Java code into bytecode, which is then executed by BALL, provided that the user has a Java Virtual Machine. Thus the BALL compilation environment must exist within a Java-ready environment. BALL, like many other common programming languages such as Java upon which BALL is built, is an imperative programming language. It is similar in structure to Java, though some changes and innovations, explained later, have been made to facilitate simpler programming.

# 2 Lexical Conventions

BALL has the following five kinds of tokens: identifiers, keywords, constants, operators, and other separators. For example, the following line of code: `player aroid = “Alex Rodriguez” from Yankees;` breaks down to the following tokens:

| | |
|---|---|
| `player` | keyword |
| `aroid` | identifier |
| `=` | operator |
| `“Alex Rodriguez”` | constant |
| `from` | keyword |
| `Yankees` | identifier |
| `;` | separator |

BALL is a strictly case-sensitive language, also much like Java.

## 2.1 Comments

BALL comments follow the same format as Java. Single line comments begin with `//` and are terminated at the end of the line. Multi line comments begin `/*` and end with `*/`.

## 2.2 Identifiers

Identifiers are similar to in Java, with the exception that BALL identifiers may begin with a digit, provided that there is at least one alphabet character or underscore in the identifier. That is, identifiers are members of the language defined by the following regular expression:
`[a-zA-Z0-9_]*[a-zA-Z_][a-zA-Z0-9_]*`

## 2.3 Keywords

The following are the keywords in the BALL language: from, any, stat, where, foreach, in, do, stopdo, times, player, team, list, number, string, print, if, then, else, return, simfunction, activate, function, returns, nothing, is, end.

### *2.3.1 from*

The `from` keyword selects an element within a list object. Example source:

```
player aroid = “Alex Rodriguez” from Yankees;
```

### *2.3.2 any*

The `any` keyword selects a random (uniform distribution) element within a list object. Example source:

```
player dback = any Diamondbacks;
```

### *2.3.3 stat*

The `stat` keyword defines a new statistic function. Example source:

```
stat avg = h/ab;
```

### *2.3.4 where*

The `where` keyword selects a sublist within a list object. Example source:

```
list 300hitters = Dodgers where (avg > .300);
```

### *2.3.5 foreach*

The foreach keyword iterates through each element in a list object. Example source:

```
foreach p in Royals:
    print p’s name;
end
```

### *2.3.6 in*

The `in` keyword is integral to the `foreach` loop structure. See 2.3.5 above.

### *2.3.7 do*

The `do` keyword creates an iterating loop, similar to do/while or for in Java. Example source:

```
number n=0;
do 5 times:
```

```
    n++;
    print “hello ”+n;
end
```

### *2.3.8 stopdo*

The `stopdo` keyword is similar to the break keyword in Java. It escapes a do loop before the completion of all iterations. Example source:

```
do 5 times:
    player dback = any Diamondbacks;
    print dback’s name;
    if (dback’s name is “Jon Garland”) then:
        stopdo;
    end
end
```

### *2.3.9 times*

The `times` keyword is integral to the `do` loop. See 2.3.7 above.

### *2.3.10 player*

The `player` keyword defines a new player object. Example source:

```
player nat = any Nationals;
```

### *2.3.11 team*

The `team` keyword defines a new team object. Example source:

```
team champs = sim(Astros, Reds, 1);
```

### *2.3.12 list*

The `list` keyword defines a new list object. Example source:

```
list 300hitters = RedSox where (avg > .300);
```

### *2.3.13 number*

The `number` keyword defines a new number instance. Example source:

```
player bigZ = “Carlos Zambrano” from Cubs;
number n = bigZ’s era;
```

### *2.3.14 string*

The `string` keyword defines a new string instance. Example source:

```
String name = “Lance Berkman”;
```

### *2.3.15 print*

The print keyword prints something to the standard output. Example source:

```
player vlad = “Vladimir Guerrero” from Angels;
print vlad’s avg;
```

### *2.3.16 if*

The `if` keyword opens an if statement. Example source:

```
player bro = any Twins;
print bro’s name;
if (bro’s name is “Joe Mauer”) then:
    print “What an average!”;
end
```

### *2.3.17 then*

The `then` keyword is integral to the `if` statement. See 2.3.16 above.

### *2.3.18 else*

The else keyword begins and alternate statement to a previous if statement. Example source:

```
player roar = any Tigers;
print roar’s name;
if (roar’s name is “Justin Verlander”) then:
    print “Wow, heck of an era!”;
else:
    print “You’re not Verlander!”;
end
```

### *2.3.19 return*

The `return` keyword is used to return either the winning team in a `simfunction` (see 2.3.20 below) or a value required for a returning function (see 2.3.22). Example source:

```
function simple(player p) returns string:
     return p's name;
end
```

### *2.3.20 simfunction*

The `simfunction` keyword is used to create a new simulation function used to carry out the actual simulations of games. Example source: [Note that team1 and team2 are implicitly passed to the `simfunction` whenever `sim` is called. These are implicit keywords.]

```
simfunction badSim is:
  if (team1's Wpct > team2's Wpct ) then:
    return team1;
  else:
    return team2;
  end
end
```

### *2.3.21 activate*

The `activate` keyword is used to apply a `simfunction` for use by the function `sim`. Example source:

```
activate badSim;
```

### *2.3.22 function*

The `function` keyword is used to define a new function. Example source:

```
function simple(player p) returns string:
     return p's name;
end
```

### *2.3.23 returns*

The `returns` keyword is used to define the return type of a function. See 2.3.22 above.

### *2.3.24 nothing*

The `nothing` keyword replaces Java's void keyword. It is a return type for functions that don't return anything. Example source:

```
function simple(player p) returns nothing:
     print p's avg;
end
```

### *2.3.25 is*

The `is` keyword is integral to the simfunction declaration (see 2.3.20).

### *2.3.26 end*

The `end` keyword is required at the end of every loop, if-statement, function, and simfunction declaration. See 2.3.5, 2.3.7, 2.3.16, 2.3.20, 2.3.22 above.

## 2.4 Constants

Much like Java, BALL constants consist of two major categories, numeric constants and string constants. These are both touched upon briefly in section 1.2 "Types."

### *2.4.1 Numeric Constants*

Numeric constants, because they apply to the `number` type, are treated only as one type. Specifically, they can contain integer or float values. [Integer values, such as a number of hits; float values such as a batting average.] The unary minus operation is also supported, so a numeric constant can be negative or positive. Numeric constants are members of the language defined by the following regular expression.
[0-9]*\.?[0-9]+

### *2.4.2 String Constants*

String constants are character sequences. They must begin and end with quotation marks. Note that BALL does not have a char type. All character sequences must be within double-quotation marks and are treated as string constants. All special escaped characters that Java supports are supported by BALL, including but not limited to: `\n`, `\t`, `\\`, `\"`.

## 2.5 Operators

BALL contains six types of operators: assignment, logical, comparison, arithmetic, incrementor, and accessor.

### *2.5.1 Assignment Operators*

The following are the assignment operators:

=, +=, -=, *=, /=, %=

They act just like in Java.

### *2.5.2 Logical Operators*

The following are the logical operators:

`and`, `or`

The java equivalents of these are && and ||, respectively.

### *2.5.3 Comparison Operators*

The following are the comparison operators:

`is`, `isnot`, >, <, >=, <=

The java equivalents of is and isnot are == and !=, respectively. The other operators act just like in Java.

### *2.5.4 Arithmetic Operators*

The following are the arithmetic operators:

+,-,*,/,%

They act just like in Java. The unary minus also fits in this category.

### *2.5.5 Incrementors*

The following are the incrementor operators:

++,--

Just like in Java, these can be unary or postfix, and in either case acts just like in Java.

### *2.5.6 Accessor*

The following is the one accessor:

`'s`

It acts like a period in Java. It returns the value of an attribute of an object, thus `p's name` in BALL is the same as p.name in Java.

### 2.6 Separators

The following are the separators in BALL: whitespace, comma, semicolon, colon, the keyword `end`, parentheses, and square brackets. Some of these are delimiters. Whitespace is always ignored, except in strings. Square brackets are used for defining lists.

# 3 Expressions

A BALL expression is evaluated in a predefined hierarchy and returns a single value. An expression can return a value of any type. The precedence of the operators when evaluating an expression is the same as the order they are explained in this section, with highest precedence first. The associativity of the operators will be explained in each section. The order in which operands are evaluated is undefined, even when the expressions have side effects.

### 3.1 Atomic Expressions

```
atom_expression : identifier
                | number
                | string
                | list_initializer
                | "nothing"
                | "(" expression ")" // completes the cycle
                ;
```

An atomic expression is an identifier, number constant, string constant, list initializer, and an expression in parentheses. Identifiers will return the value of the variable bound to the identifier, if it is a properly declared variable of type number, list, string, team, or player.

### 3.2 Primary Expressions

```
primary_expression : atom_expression
                   | function_call
                   ;
```

A primary expression is either an atomic expression or a function call. A function call behaves just like how function calling happens in C, with a few exceptions.

Arguments are specified inside a list separated by commas (“,”) inside parentheses.

Functions can only be referenced by identifiers. BALL does not support storing functions inside constructs such as lists, and thus functions can only be referenced through their name that exists in the global scope.

## 3.3 Postfix Expressions

Postfix expressions are left-associative.

```
postfix_expression : primary_expression
                   | postfix_expression "'s" identifier
                   | postfix_expression "where" "(" expression ")"
                   | postfix_expression "++"
                   | postfix_expression "--"
                   ;
```

### *3.3.1 Attribute calling*

If the expression on the left side of the “'s” accessor operator evaluates to either a *team* or a *player* object, BALL searches for an attribute with the name *identifier* inside the left operand, and if an attribute with a matching name is found, the value of that attribute is returned. If there is no attribute with the name found, or if the left side of the expression doesn't evaluate to either a team or player, the compiler fails.

### *3.3.2 “where” expression*

"where" takes a *team* or *list* on its left side, and an expression enclosed in parentheses. The expression must have all identifiers inside resolvable in the current scope, except for identifiers that are attribute names of whatever the list is storing, or if the left expression returns a team, valid attributes of a player. For example:

```
Yankees where ( type is "batter" and AVG > 3 or max(3,1) );


Yankees where ( nonexistent_attribute is 0 ); /* compiler error */


number wins = 0, minimum_wins = 39;
[ Red_Sox, Orioles, Mets ] where ( wins > minimum_wins );
```

here, "type" and "AVG" are valid attributes for a player and "wins" is a valid attribute for a team. If an identifier is a name for an attribute in the list content's type AND a valid name for a variable or function in that scope, the compiler resolves it towards the contents' attribute, NOT the variable. Conceptually, "where" creates an inner scope with new names bound to the list/team content's attributes.

### *3.3.3 postfix increments*

The postfix operators "++" and "--" behave identically with C and Java's increment operations, as long as the expression before it evaluates to an identifier for a variable with type "number". The variable's value will be returned *before* it is incremented/decremented. "++" increments the variable and "--" decrements the variable.

## *3.4 unary expressions*

```
unary_expression : postfix_expression
                 | "++" unary_expression
                 | "--" unary_expression
                 | primary_expression "from" unary_expression
```

```
| "any" unary_expression
;
```

unary expressions are evaluated right to left (right-associative).

### *3.4.1 prefix increments*

The prefix increments “++” and “--” only makes sense on operands that evaluate to an identifier for a variable with type “number”. The variable's value is incremented/decremented *before* the value is returned. “++” increments the variable and “--” decrements the variable.

### *3.4.2 “from” expression*

“from” is a matcher operation, taking any object as the left operand and either a list or a team as the right operand. It then compares the left operand with the items inside the right operand (which are again objects if the right operand is a list, or players if it's a team), and returns one object that's an element of the right operand that matches the left operand, or “nothing” if none of the elements match.

How “from” compares the left operand with each item in the right operand:

- If the right operand is a list, the elements are checked in the order they are declared. If the list is a result of a concatenation of two sublists,
- If the right operand is a team, the elements are checked in the order they are specified in the CSV file.
- If the left operand and elements of the right operand have the same type, they are compared by evaluating the expression “`(left_opr is right_opr_elem)`” where left_opr is the left operand's value and right_opr_elem is the element of the right operand that is being matched.
- If the left operand is a string and the right operand is a team or a player, the string is compared with the team/player's NAME attribute using the “is” operator.
- Any other comparisons between two objects of different types returns false.

### *3.4.3 “any” expression*

“any” retrieves one random element from the operand, provided that the operand is of type *list* or *team* or an expression that evaluates to type *list* or *team*.

## 3.5 multiplicative operations

```
multiplication_expression : unary_expression
```

```
                        | multiplication_expression "*"
unary_expression
                        | multiplication_expression "/"
unary_expression
                        | multiplication_expression "%"
unary_expression
                        ;
```

All multiplicative operators are left-associative. The order in which the operands are evaluated (left first or right first) is undefined and implementation dependent. The operand expressions must have type “number”.

The result of the operator '*' is the product of the two operands. If an implementation implicitly stores numbers as both integers and floating point numbers, both operands must be converted to floating point if one of the operands has a floating-point internal type.

The result of the operator '/' is the division between the two operands. Result of division by zero is implementation defined, whether to fail at compilation or throw an error in runtime. Rules on conversion of the two operand types are identical with the multiplication operator.

The operator '%' returns the modulo of the two numbers. Rules of the division operator apply here as well.

## 3.6 addition operations

```
addition_expression : multiplication_expression
                    | addition_expression "+"
multiplication_expression
                    | addition_expression "-"
multiplication_expression
                    ;
```

Both addition and subtraction operators are left-associative. The order in which operands are evaluated is implementation dependent. The operand expressions for '-' must evaluate to type “number”. The operand expressions for '+' must evaluate to the same type, either “number”, “string”, or “list”.

'+' returns the addition between the two operands. Rules for internal conversion for numeric operand values are identical to the multiplication operator in section 3.5.5. If '+' receives string operands, the result is a string concatenation of the two. If '+' receives list operands, the result is a new list with the elements in the second list appended in order to the end of the first list.

'-' returns the subtraction between the two operands. Rules for internal conversion for the operand values are identical to the addition operator.

## 3.7 comparison expressions

```
comparison_expression : addition_expression

                      | comparison_expression "is"
addition_expression

                      | comparison_expression "isnot"
addition_expression

                      | comparison_expression ">"
addition_expression

                      | comparison_expression "<"
addition_expression

                      | comparison_expression ">="
addition_expression

                      | comparison_expression "<="
addition_expression

                      ;
```

Evaluation between a series of comparison operations is done from left to right (all comparison operators are left-associative).

The equality operator “is” will always return 0 (false) when the left hand and right hand expressions evaluate to different types. If the left and right operators have the same type, they will be checked for equality according to their type.

Numbers: number equality (==).

String: exact string matching, case sensitive.

Players, Teams, Lists: two of these objects are the same if they are internally the same instance located on the same address in memory.

“nothing” is always equal to another “nothing”.

## 3.8 logical expressions

```
logical_not_expression : comparison_expression
                       | "not" logical_not_expression
                       ;
logical_and_expression : logical_not_expression
                     | logical_and_expression "and"
logical_not_expression
                       ;
logical_or_expression : logical_and_expression
                      | logical_or_expression "or"
logical_and_expression
                      ;
```

Logical expressions are ordered, with decreasing precedence, as NOT, AND, and OR expressions. Each of these expressions take operands of type “number” and returns another number that represents the truth value of the expression. An expression is evaluated to “false” if it evaluates to the number zero. An expression is evaluated to “true” whenever it evaluates to anything else.

“not” takes a single expression and returns “false” if the original expression is “true” and “true” if the original expression is “false”.

“and” takes two expressions and returns “true” if both expressions return “true”, “false” otherwise.

“or” takes two expressions and returns “false” if both expressions return “false”, “true” otherwise.

The logical “or” expression is the expression with the lowest precedence, and is made of terms of expressions with higher precedence, so therefore an expression proper is a logical “or” expression.

```
expression : logical_or_expression

           ;
```

# 4 Declarations

Declarations specify how BALL will interpret a new identifier within the scope of the declaration. Declarations have the form:

```
declaration : type variable_declarators ";"

            ;
```

`variable_declarators` is a comma-separated list of declarators that contains the identifiers being declared:

```
variable_declarators : variable_declarator
                     | variable_declarators "," variable_declarator
                     ;
variable_declarator : identifier
                    | identifier "=" expression
                    ;
```

`variable_declarator` produces `identifier` as well as `identifier "=" expression` which allows for declaration of the form:

```
number aNumber;
```

as well as:

```
number aNumber, anotherNumber, yetAnotherNumber;
```

and additionallly:

```
number someNumber=9;
```

```
number aNumber=10, anotherNumber=11, yetAnotherNumber=12;
```

## 4.1 Type Specifiers

The type specifiers in BALL are:

```
type : "number"
     | "string"
     | "list"
     | "team"
     | "player"
     | "stat"
     | "nothing"
     ;
```

Type specifiers are mandatory when declaring an identifier for the first time. Identifiers may not be used without a type specification. Furthermore, only one type may be specified at a time, which means that

```
number aNumber, anotherNumber, yetAnotherNumber;
```

would declare `aNumber, anotherNumber, yetAnotherNumber` as numbers.

The type "nothing" is used when specifying return values for functions. For example:

```
function printHello (string helloString) returns nothing:
  print helloString;
end
```

Using `nothing` in other contexts, such as `nothing x = 0;` would have no effect.

Nevertheless, `nothing` can be used to check for the existence of an item:

```
if("Juan Carlos" from Dodgers is nothing) then:
   print "Juan Carlos is not in the dodgers!"
end
```

## 4.2 Meaning of Declarations

Declarations map an identifier to an expression. Since an expression can produce any `atom_expression` , this allows for declarations such as `number someNumber=9;` and `string someString="Hello!";.` But expressions may produce other combinations, and their return value will be mapped to the identifier. In this example, `aTeam` will be mapped to the return value of the `load` function:

```
team aTeam = load(myTeam.team);
```

Furthermore, identifiers could be mapped to the return values of arithmetic expressions or list manipulations.

## 4.3 Declaring a list

Since expression generates an `atom_expression`

```
atom_expression : identifier
                | number
                | string
```

```
                | list_initializer
                | "(" expression ")" // completes the cycle
                ;
```

and atom_expression produces a list_initializer,

```
list_initializer : "[" variable_list "]"
                 | "[" "]"
                 ;
variable_list : expression
              | variable_list "," expression
              ;
```

We can construct a list in the following way:

```
list someTeams = [Dodgers, Astros, Orioles];
```

Or we can use the return value of a list manipulation such as:

```
list someTeams = [Dodgers, Astros] + [Orioles, Pirates];
```

The identifier `someTeams` now resolves to `[Dodgers, Astros, Orioles, Pirates];`

Additionally, list manipulation accepts the '-' operator:

```
list someTeams = [Dodgers, Astros, Orioles, Pirates]; - [Orioles];
```

The identifier `someTeams` now resolves to `[Dodgers, Astros, Pirates];`

## 4.4 Declaring players and teams

Players and teams may not be constructed within any ball statement. The only available way for a user to define his own team is by writing a .team file for it. Teams can be declared and initialized by using the return value of an expression. Examples:

```
team Dodgers = load(Dodgers.team);

team winner = sim(Dodgers, Orioles, 1);
```

Players may only be initialized by extracting them of a team. Examples:

```
player Beltran = "Carlos Beltran" from Mets;

players randomMetsPlayer = any Mets;
```

### 4.5 Declaring a stat

A stat may be declared anywhere in a BALL program. Just like variables, stat declarations are of the form:

```
stat timesWon = 23;
```

Although stats can be initialized with a number, this is not interesting. Their main purpose is to hold a formula. A user should think of stats as small, portable functions. Players (and teams) have primitive stats such as IP (innings pitched), K (strikeouts), H (hits allowed), BB (walks allowed), and ER (earned runs) for Pitchers; AB (at-bats), R (runs), H (hits) 2B (doubles), 3B (triples), HR (home runs), and BB (walks) for Batters. These primitive stats can be accessed with: `player1's IP`, for example. When defining a new stat, the user will want to build upon primitive stats. For example:

```
stat ERA = ER/IP*9;
```

This line of code defines a new stat for pitchers: the Earned Run Average. A user may now access this stat from any pitcher just like he would access a primitive stat: `player1's ERA`.

As a side note, the user should be aware that he may collect a stat as a number at any time with code such as:

```
number player1ERA = player1's ERA;
```

# 5 Statements

Although similar to statements in C, BALL statements follow different rules. They fall into three groups:

```
statement : function_definition
```

```
          | sim_function_definition
          | body_statement
          ;
```

body_statement produces:

```
body_statement : if_statement
               | iteration_statement
               | jump_statement
               | declaration
               | expression_statement
               | activate_statement
               | print_statement
               | assignment_statement
               ;
```

Body statements are statements that are allowed in the body of if-statements, functions, loops, and such. We separated these statements from function definitions and sim function definitions since we do not want BALL to accept the definition of new functions within an if statement, loop, or another function.

Body statements are executed in sequence. Functions and simulation functions, however, may be declared anywhere in the program and may be called upon anywhere in the BALL program. The following code is valid:

```
print helloPrinter();
function helloPrinter() returns string:
  return "hello!";
end
```

and will print `hello!`

## 5.1 Function definition

Function definitions are generated by the following grammar:

```
function_definition : "function" identifier "(" parameter_list ")"
"returns" type ":" body_statement_list "end"

                    ;
```

Any statement starting with the "function" keyword is immediately identified as a function definition. The return type of the function is specified after the parameters, using the keyword "returns".

A function accepts a comma-separated list of parameters generated by:

```
parameter_list : parameter

               | parameter_list "," parameter

               ;

parameter : type identifier

          ;
```

This grammar generates classic Java and C -style function definitions with a friendly twist:

```
function iAddNumbers (number num1, number num2) returns number:

  return num1 + num2;

end
```

## 5.2 Simulation function definition

Simulation function definitions require only a brief explanation since they are in fact simpler to declare than other functions:

```
sim_function_definition : "simfunction" identifier "is:"
body_statement_list "end"
```

```
                ;
```

This grammar is very similar to `function_definition` but varies in the following ways:

- `simfunction` definitions do not accept any parameters. This is because the parameters are implicit: a simulation function can only handle two teams which can be accessed with the identifiers `team1` and `team2` at anytime inside the function. `team1` and `team2` represent the two teams that the `sim` function is currently simulating for.
- `simfunction` does not have an explicit return value. Again, this is because the return value is restricted to "team". A simulation function should always return a team (for use in another simulation function for example).

## 5.3 If Statements

Our grammar for if-statements is:

```
if_statement : "if" "(" expression ")" "then:" body_statement_list
"end"
             | "if" "(" expression ")" "then:" body_statement_list
else_statement "end"
             ;
```

These are similar to Java if-statements, but the delimiters are `"then:"` and `"end"` instead of { and }. `expression` should be any boolean expression or other expression that resolves into a boolean.

The first line generates statements of the form:

```
if (x < y) then:
  print "x < y!! x used to be: " + x + " and now x has increased to: "
+ ++x;
end
```

The second line adds capacity for "else" statements which have the grammar:

```
else_statement : "else:" body_statement_list
               ;
```

We can now produce:

```
if (x < y) then:
  print "x < y!! x used to be: " + x + " and now x has increased to: "
+ ++x;
else:
  print "x >= y!! x used to be: " + x + " and now x has decreased to:
" + --x;
end
```

## 5.4 Iteration Statements

There are two forms of loops in BALL. The grammar is simply:

```
iteration_statement : "do:" body_statement_list "end"
                    | "do" expression "times:" body_statement_list
"end"
                    | "foreach" identifier "in" identifier ":"
body_statement_list "end"
                    ;
```

The first and second lines produce do loops. A do loop simply loops a specified amount of times. They come in two forms which can be illustrated by the following examples:

```
number i = 1;
do:
  i *= 2;
  if (i >= 1024) then:
```

```
    stopdo;
  end
end
```

This do loop is declared with "do:" and ends with "end". Everything in between will be repeated indefinitely or until "stopdo;" is stated.

```
number i = 0;
do 5 times:
  i++
  print "looped " + i + " times!";
end
```

This second do loop is performed exactly 5 times since that is the number of loops specified. Note that the number of loops can be any expression, and code such as `do num1+num2 times:` is acceptable.

Finally, the third line of the grammar above produces `foreach` statements. A foreach statement iterates through a list of items (also a team) and performs an action on each of the items in sequence. A `foreach` statements may as well be halted with a "stopdo;" statement. `foreach` loops are of the form:

```
print "The Mets roster is:";
foreach player in Mets:
  print player's name;
end
```

## 5.5 Jump Statements

The C Reference Manual describes jump statements with: "Jump statements transfer control unconditionally." In BALL, we want to prevent programmer mistakes by omitting some of such tools

which many programmers flag as "bad practice". Hence, BALL only supports two jump statements: `return;` and `stopdo;` :

```
jump_statement : "stopdo" ";"
               | "return" expression ";"
               | "return" ";"
               ;
```

A `stopdo;` statement may appear only in an iteration and terminates the execution of the smallest enclosing such statement: control passes to the statement following the newly-terminated one.

`return;` may only be used in functions. When invoked, the function halts and returns to its caller. The function may return nothing, as is the case with functions of return type "nothing" (this also happens when letting the function flow off the end). Or the function may return an expression with resolves into a value.

## 5.6 Activate Statement

The activate statement is short, simple, and requires little explanation:

```
activate_statement : "activate" identifier ";"
                   ;
```

`activate` is followed by an identifier, which should be the name of a simulation function defined in the same BALL program. This statement simply sets the simulation function as the "active" function for use in the built-in function `sim`. For example:

```
activate mySimFunction;

sim(Astros, Mets, 3);
```

This will return the winner of three games between the Astros and the Mets using `mySimFunction` as simulation function. `activate` may be called multiple times in the same BALL program.

## 5.7 Print Statement

The print statement is very simple as well:

```
print_statement : "print" expression ";"
                ;
```

In a nutshell, `print` will print expression if expression resolves to a string or the toString value of any team or player that `expression` returns. It will also print any numbers. Additionally, print accept concatenations of any players, teams, strings or numbers. Example:

```
print player1's name + " is " + 5 + "th in the rankings!"
```

## 5.8 Assignment Statement

The assignment statement is an opportunity to re-define the interpretation of an identifier. The grammar for these assignments is:

```
assignment_statement : identifier assignment_operator expression ";"
                     ;
assignment_operator : "=" | "+=" | "-=" | "*=" | "/=" | "%=" ;
```

A typical variable re-assignment will be of the form:

```
number x = 10;
x = 11;
```

The final value of x is 11.

Assignments can be made with more than just the "=" operator. The operators `"+="`, `"-="`, `"*="`, `"/="`, `"%="` are shortcuts for:

```
i = i + num1;
i += num1;
i = i - num1;
i -= num1;
```

```
i = i * num1;

i *= num1;

i = i / num1;

i /= num1;

i = i % num1;

i % num1;
```

# 6 Scope

All declarations made outside of functions, loops, or if-statements are global. Functions themselves are also global. However, the scope of all declarations made inside of functions, loops, or if-statements is the function, loop, or if-statement itself. The `end` keyword destroys these local-scope declarations.

# 7 Built-in Functions

## 7.1 Load function

The `load` function is used to import external csv team files (see section 8 for csv format and specification) into team objects in a file. The syntax is:

```
load(string);
```

Example:

```
team Astros = load("Astros.team");
```

## 7.2 Rand function

The `rand` function is used to generate a random (float) number between two given arguments. The syntax is:

```
rand(number,number);
```

Example:

```
number randomNum = rand(0,1);
```

## 7.3 Max function

The `max` function is used to determine the largest of two given arguments. The syntax is:

```
max(number,number);
```

Example:

```
number larger = max(5,100);
```

## 7.4 Min function

The `min` function is used to determine the smallest of two given arguments. The syntax is:

```
min(number,number);
```

Example:

```
number smaller = min(10,40);
```

## 7.5 Top function

The `top` function is used to generate a sublist of objects representing the top number of items in that list in reference to the stat. The syntax is:

```
top(number,list,stat);
```

Example:

```
list top5Batters = top(5,Braves,avg);
```

## 7.6 Bottom function

The `bottom` function is used to generate a sublist of objects representing the bottom number of items in that list in reference to the stat. The syntax is:

```
Bottom(number,list,stat);
```

Example:

```
list bottom2Pitchers = bottom(2,Brewers,era);
```

## 7.7 Sim function

The `sim` function is BALL's main function. When this function is called, it takes the activated (see section 2.3.21) simfunction (see section 2.3.20) and runs it on the two teams a number of times. The syntax is:

```
sim(team,team,number);
```

Example:

```
sim(Rockies,Giants,5);
```

# 8 CSV Specifications

The following is an example of a CSV used as a team file. This file would be called `Astros.team` and contains information on the whole team, batters, and pitchers, and their stats.

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

# 9 Full Grammar

This is the yacc-compatible grammar for our language:

The grammar has the undefined terminal symbols: *identifier, string, number*.

```
%token identifier

%token string

%token number



%%

/***PROGRAM***/
program : statement_list

        ;


statement_list : statement

               | statement_list statement

               ;


statement : function_definition

          | sim_function_definition

          | body_statement

          ;


body_statement_list : body_statement

                    | body_statement_list body_statement

                    ;


/*Body Statements are all statements except function declarations*/
body_statement : if_statement
```

```
                | iteration_statement
                | jump_statement
                | declaration
                | expression_statement
                | activate_statement
                | print_statement
                | assignment_statement
                ;


/**FUNCTION_DEFINITION**/
function_definition : "function" identifier "(" parameter_list ")" "returns" type ":"
body_statement_list "end"
                    ;


parameter_list : parameter
                | parameter_list "," parameter
                ;


parameter : type identifier
          ;


/**SIM_FUNCTION_DEFINITION**/
sim_function_definition : "simfunction" identifier "is:" body_statement_list "end"
                        ;


/**ACTIVATE_STATEMENT**/
activate_statement : "activate" identifier ";"
                   ;


/**PRINT_STATEMENT**/
print_statement : "print" expression ";"
                ;


/**IF_STATEMENT**/
```

```
if_statement : "if" "(" expression ")" "then:" body_statement_list "end"

             | "if" "(" expression ")" "then:" body_statement_list else_statement
"end"

             ;


else_statement : "else:" body_statement_list

               ;


/**ITERATION_STATEMENT**/
iteration_statement : "do:" body_statement_list "end"

                    | "do" expression "times:" body_statement_list "end"

                    | "foreach" identifier "in" identifier ":" body_statement_list
"end"

                    ;



/**JUMP_STATEMENT**/
jump_statement : "stopdo" ";"

               | "return" expression ";"

               | "return" ";"

               ;


/**ASSIGNMENT_STATEMENT**/
assignment_statement : identifier assignment_operator expression ";"

                     ;


assignment_operator : "=" | "+=" | "-=" | "*=" | "/=" | "%=" ;


/**DECLARATION**/
declaration : type variable_declarators ";"

            ;


variable_declarators : variable_declarator

                     | variable_declarators "," variable_declarator
```

```
                    ;


variable_declarator : identifier
                    | identifier "=" expression
                    ;


/*TYPE*/
type : "number"
     | "string"
     | "list"
     | "team"
     | "player"
     | "stat"
     | "nothing"
     ;



/**EXPRESSION_STATEMENT**/
expression_statement : ";"
                     | expression ";"
                     ;


/*EXPRESSION*/
expression : logical_or_expression
           ;


/*LOGICAL*/
logical_or_expression : logical_and_expression
                      | logical_or_expression "or" logical_and_expression
                      ;


logical_and_expression : logical_not_expression
                       | logical_and_expression "and" logical_not_expression
```

```
                        ;


logical_not_expression : comparison_expression
                       | "not" logical_not_expression
                       ;


/*COMPARISON*/
comparison_expression : addition_expression
                      | comparison_expression "is"   addition_expression
                      | comparison_expression "isnot" addition_expression
                      | comparison_expression ">"    addition_expression
                      | comparison_expression "<"    addition_expression
                      | comparison_expression ">="   addition_expression
                      | comparison_expression "<="   addition_expression
                      ;


/*ARITHMETIC*/
addition_expression : multiplication_expression
                    | addition_expression "+" multiplication_expression
                    | addition_expression "-" multiplication_expression
                    ;


multiplication_expression : unary_expression
                          | multiplication_expression "*" unary_expression
                          | multiplication_expression "/" unary_expression
                          | multiplication_expression "%" unary_expression
                          ;


/*UNARY*/
unary_expression : postfix_expression
                 | "++" unary_expression
                 | "--" unary_expression
```

```
                | primary_expression "from" unary_expression
                | "any" unary_expression
                ;


/*POSTFIX*/
postfix_expression : primary_expression
                   | postfix_expression "'s" identifier // attribute/stats call
                   | postfix_expression "where" "(" expression ")"
                   | postfix_expression "++"
                   | postfix_expression "--"
                   ;


/*PRIMARY*/
primary_expression : atom_expression
                   | function_call
                   ;


/*FUNCTION_CALL*/
function_call : identifier "(" ")"
              | identifier "(" argument_list ")"
              ;


argument_list : expression
              | argument_list "," expression
              ;


/*ATOM_EXPRESSION*/
atom_expression : identifier
                | number
                | string
                | list_initializer
                | "nothing"
                | "(" expression ")" // completes the cycle
```

```
                ;

list_initializer : "[" variable_list "]"
                 | "[" "]"
                 ;

variable_list : expression
              | variable_list "," expression
              ;
%%
```