BoredGames Language Reference Manual

A Language for Board Games

Brandon Kessler (bpk2107) and Kristen Wise (kew2132)

Table of Contents

1. Introduction

This manual describes the BoredGames language. BoredGames is designed for easy implementation of turn-based board games. It enables users to quickly and efficiently setup a board, establish rules for the game, and define how each turn should be played. All three of these tasks are split into different sections which are integrated together by our compiler.

2. Lexical Conventions

2.A Comments

BoredGames allows multiline comments. The syntax is as follows:

Comment Symbol	*Example*
/* */	/**Comment goes here**/

2.B Tokens

BoredGames has six types of tokens: identifiers, keywords, constants, string literals, operators, and separators. Some whitespace is required to separate adjacent identifiers, keywords, and constants.

2.C Identifiers

An identifier is a sequence of letters, digits, and underscores. An identifier must start with a letter or an underscore. Letters include the ASCII characters A-Z, and a-z. Digits are the characters 0-9. BoredGames is case sensitive.

2.D Keywords

The following identifiers are reserved for the use as keywords, and cannot be used otherwise:

Keyword	*Description*
bool	Boolean Data Type
double	Double Data Type
int	Integer Data Type
String	String Data Type
Coord	Coordinate Data Type
Matrix	2D Matrix Data Type
Board	The Game Board
Player	The Players in the Game
Pieces	The Game Pieces
Setup	Where Pregame Setup Occurs in a
Rules	Where the Rules of the Game are Defined
Play	Where how a Turn will Proceed is Specified
rule	A Specific Rule of the Game
if, else, elseif	Conditional expression, else is optional
loop	A Conditional Repetition of a Block of Code
return	A Value to be Returned from a Rule to its Caller
NextPlayer	Moves the Current Player to Whoever is Next
true	Boolean Constant
false	Boolean Constant
Dice	Generates a Roll of a Die

2.E Constants

There are a few types of constants in BoardGames:

- Integer constants which is a sequence of digits [0, 9]
- Boolean constants which are True and False
- Double constants consist of an integer part followed by a decimal part each of which are composed of a sequence of digits [0, 9]

2.F String Literals

A sequence of ASCII characters surround by double quotes. The following escape sequences may be used to represent the following characters:

Newline	'n
double quote	' "
single quote	' '

2.G Operators

BoredGames has comparison, arithmetic, boolean, and declaration operators. The syntax of these operators can be found later in the manual.

2.H Separators

Separator	*Description*
,	Separates Elements in a List
;	Ends the Current Line

3. Meaning of Identifiers

In BoredGames an identifier is a keyword, rule, or a name for a variable. Rules for how to name identifiers is defined in section 2.C

3.A Variables

A variable is an identifier that has a constant value assigned to it. Each variable must have a type defined by the user. Using a variable within the scope of its declaration will get the value bound to the variable.

3.B Rules

Rules are essentially functions that must return either true or false. Each rule has an identifier bound to it so that the rule can be referred to in the Play section of the program.

3.C Scope

The scope of a variable identifier begins at the end of its declaration and continues until the end of its particular block. These blocks are the Setup Section, the Play Section, or a particular rule. A rule identifier's scope begins at the end of the rule declaration and continues until the end of the program. This is to say that the rule identifier's scope continues for the remainder of the Rules and the Play sections.

3.D Basic Types

Basic Type	*Description*
Bool	Boolean Type
Int	Integer Type
double	Double Type
String	C Like String Type

3.E Structured Types

There is only one Structured Type in BoredGames and it is Matrix. Matrix has the format Matrix *type* where *type* is the type of all elements in the Matrix.

3.F Derived Types

There are two derived types in BoredGames. They are Coord and rule.

- Coord is an integer coordinate pair which can be written as (int, int)
- rule is a function that must return a boolean. A rule can only be defined in the Rules section of the program.

4. Expressions

For our purpose an lvalue is an identifier with a suitable type that is on the left hand side of expressions.

4.A Primary Expressions

primary-expression:

constant

identifier

string

(expression)

A constant is a constant literal, as opposed to a variable. An identifier has the type specified at its declaration. A string is represented by an array of chars. A parenthesized expression has the same type and value as the original expression.

4.B Postfix Expressions

Operators are bound to expressions from left to right.

postfix-expression:

primary-expression

postfix-expression [expression]

postfix-expression (argument-expression-list$_{optional}$)

postfix-expression . identifier

postfix-expression ++

postfix-expression --

argument-expression-list:

assignment-expression

argument-expression-list , assignment-expression

4.C Array References

Although arrays may not be directly defined in BoredGames, there are multiple array structures within Board, Pieces, and Player that need to be accessed. An expression in square brackets after a postfix expression indicates the access of an array reference. The type of the postfix expression must be one of the basic types available in BoredGames. The expression inside the brackets must be of integer type between 1 and the size of the array, signifying the index of the array being accessed.

4.D Matrix References

Similar to array references, Matrix references take an expression of type coordinate, or Coord, made of the x and y values representing an entry in the Matrix. The Matrix starts at the bottom left hand side indicated by coordinate (1,1), and has the number of rows and columns specified at the Matrix declaration. If the expression is not of coordinate type, the expression used to access a Matrix location can be declared within the brackets, using standard coordinate right-hand value declaration notation.

4.E Function Calls

Although functions may not be defined in BoredGames, there are predefined library functions available. A function call is a postfix operator denoted by parenthesis, when followed by a library function, and takes a variable-length set of arguments separated by commas. The argument list length depends on the particular input required for a given function, and can be zero. Some BoredGames functions have multiple versions that accept different input types depending on the length of the provided argument list. The expressions passed to a function should match the function parameters in type and should be within scope of the function's call. If these conditions are not met, it will result in an error. The postfix expression must return the type specified by the BoredGames function, and though it is not necessary for an lvalue to be supplied for every function in use, if one does exist, it must match the return value of the postfix expression.

4.F Object Structure Reference

A postfix expression followed by a dot and identifier is used to access details of the items in BoredGames. The specific members are unique to each item as seen below. The type of the object member can be of a structural, derived, or basic type, and can be assigned to an lvalue with a matching type or altered according to the specific type's rules. This is further discussed in the Board, Player, and Pieces sections.

4.G Postfix Incrementation

A postfix expression followed by a ++ or -- operator results in a postfix expression of the same algebraic type as the original expression. The value of the expression is incremented or decremented, respectively, by 1. The expression must be an lvalue, as opposed to a constant.

4.H Logical Operators

Logical operators are performed on Boolean values and return Boolean values.

logical-expression:

logical-operator (Boolean)

Boolean logical-operator Boolean

Operator	*Explanation*
!	Logical NOT: True becomes False; False becomes True
&&	Logical AND: True if both sides are True; False otherwise
||	Logical OR: True if either side is True; False if both sides are False

4.I Casts

An expression preceded by a parenthesized data type converts the type of the original expression into the one specified in the parenthesis.

cast-expression: (type-name) cast-expression

Valid casts are:

- *(double) int* /* adds zero decimal to the original integer */
- *(int) double* /* truncates the decimal portion */
- *(int) string* /* returns the numeric value of each character */

A casted expression can be set to an lvalue, but the entire expression including the cast type cannot be an lvalue.

```
int  i = 2;

double d = 2.0;

d = (double) i;          /* this is valid */

(int) d = i;             /* this is not valid */
```

4.J Numerical Operators

Operators return type double if a double is used in either expression otherwise they return an int. The Modulus operator can only be used on two ints. In order of precedence:

Operator	*Description*
*	Multiplication
/	Division
%	Integer Modulus
+	Addition
-	Subtraction

Numerical operators group left-to-right, and can be performed on algebraic data types. Integer division result in an integer with the remainder discarded.

4.K Relational and Equality Operators

Relational and equality operators return int values. If the expression is True, the return value is 1; if the expression is false, the return value is 0.

relational/equality-expression:

rel/eq-expression rel/eq-operator rel/eq-expression

In order of precedence:

Operator	Description
<	Less Than
>	Greater Than
<=	Less Than or Equal to
>=	Greater Than or Equal to
==	Equal to
!=	Not Equal to

Relational operators, the first four listed, bind tighter than equality operators, the last two listed. So when used in conjunction:

a>b == b>d

is equivalent to:

(a>b) == (b>d)

5. Declarations

5.A Type Specifiers

type-specifier:

int

double

bool

String

An identifier has exactly one type. The type specifier precedes the interval.

5.B Loop Statement

loop (*expression)* { *statement* }

The expression in the loop parenthesis is a Boolean, which is evaluated before the body of the loop is executed. The statement in the braces is a sequence of instructions that takes place if the expression Boolean is True. The expression will be repeatedly tested until it fails. Then, the statement breaks and the next portion of the program runs.

5.C Iterator Statement

The colon operator acts as an iterator when located between two integer constants or identifiers. It starts at the left-hand value and increments by 1 until it reaches the right-hand value. The colon can be used inside a loop expression or when referring to an array or matrix index.

constant:constant

constant:identifier

identifier:constant

identifier:identifier

5.D Selection Statements

if (*expression*){ *statement }*

if (*expression*){ *statement* }else{ *statement* }

if (*expression*){ *statement* }elseif (*expression*) *statement*

The expressions for all if and elseif statements must evaluate as a whole to the Boolean type, though relational and equality operators can be used to translate arithmetic types into Boolean, i.e. (5<6) evaluates to True. The if statement checks the expression, possibly executes the statement, and continues to the next line of the program. The if, else statement blocks insist that either one statement or the other is executed before the next section of the program is run. The if, elseif statement blocks, where elseif is repeated any number of times, connect multiple if statements so that only the first conditional that is met is executed. Once one is executed, or if none are, the program continues to the next section.

6. Program Components

6.A Board Item

6.A.1 Board Declaration

The Board is structurally a matrix of any size. The parameters passed are the size of the maximum number of rows and columns that will be needed. A position on the board can be set to inactive, through a function having no return value, so any active configuration can be achieved. Initially each location is set to active.

Board[number-of-rows, number-of-columns]

Board[coordinate].inactive()

6.A.2 Board Access

When Pieces are created, they can be set to a certain player's *inventory* (off of the Board) or a Player's *onBoard* (at a Board coordinate). This is discussed in the Piece section. BoredGames has two library functions *add(...)* and *move(...)* (discussed in Library Functions) that modify the location of Pieces on the Board. Board also has a function that can be used in the Rules section of the program to check if a position has any Pieces. It returns a Boolean, true or false. A Board's point value can be called by the function *point()*, and returns a value if one was given initially; otherwise it is invalid.

Board[coordinate].unoccupied()

Board[coordinate].point()

Every location on the Board has a Piece array as well, and it can be accessed by a dot referencing to Pieces and using brackets to enclose the index. Additionally, the Pieces can be accessed by providing Piece information enclosed in parenthesis. If no owner is specified, the current Player (maintained by the program) is used. This is further explained in the Pieces section.

6.B Player Item

6.B.1 Player Declaration

A Player item is created for each of the game users. When Pieces are created, they are assigned to a Player and given titles and, optionally, point values.

Player[player-name]

Player[player-name, point-value]

6.B.2 Player Access

Players have *inventories* and *onBoard* sections that distinguish between Pieces off and on the Board. When passed a Player name and Piece title, the location can be accessed using the location function; this returns a coordinate. Player *inventory* and *onBoard* sections are accessed as arrays, and the Pieces are stored in the order they are added. Methods of accessing Pieces are further described in the Pieces Item section.

Player . inventory/onBoard [array-index]

Player . inventory/onBoard (player-name$_{optional}$, piece-name, coordinate$_{optional}$)

A Player's name and point value, if provided, can be accessed through calls to Player. Any call to Player refers to the current Player.

Player . piece-access-method [array-index]

piece-access-method: name(), point()

6.C Pieces Item

6.C.1 Pieces Declaration

Pieces are created with a Player as an owner, the type of piece as a name, and the number of pieces of this type. Three required fields are player-name (type string), piece-name (type string), and number-of-pieces (type int). The point-value (type double or int) and move-matrix are both optional. The move-matrix is a Matrix with an enumerated list of move sequences a piece type can make. The Matrix must consist of moves up, down, left, right, ruDiag, rdDiag, luDiag, or ldDiag. Each row of the move-matrix represents a move and each column is the sequence of actions within the move. Since a matrix must have a constant number of columns in each row,

any extra columns in a row where the move sequence is enumerated in full should be designated by an empty String.

Pieces(player-name, piece-name, number-of-pieces)

Pieces(player-name, piece-name, number-of-pieces, point-value)

Pieces(player-name, piece-name, number-of-pieces, move-matrix)

Pieces(player-name, piece-name, number-of-pieces, point-value, move-matrix)

6.C.2 Pieces Access

Pieces are stored and can be accessed from either the Board or Player items. BoredGames handles the Pieces' underlying connections between the Board and Player, so there are no updates required. A Piece can be accessed through Player *inventory* or *onBoard*. When a Piece is in the inventory, the coordinate is automatically (0,0). Pieces can also accessed through the Board, using the Piece information or the index in the array of Pieces connected to the position on the Board. When array accesses to inventory, onBoard, or Pieces involve an integer enclosed in brackets, as shown in Pieces Access, it is treated as an index in the array, and the stored Piece is used. At any time, if a Piece description does not include a Player name, the program will assume that the current Player, as tracked by BoredGames, is meant.

Through Player:

Player.onBoard(player-name$_{optional}$, piece-name, coordinate$_{optional}$)

Player.inventory(player-name$_{optional}$, piece-name, coordinate$_{optional}$)

Through Board:

Board[coordinate].Pieces[array-index]

Board[coordinate].Pieces(player-name$_{optional}$, piece-name, coordinate$_{optional}$)

6.C.3 Pieces Functions

Pieces have information stored in them, and can be reached through at most five functions. Once a Piece is reached, by any of the above mentioned sequences, the functions can be applied.

player/board-access-method . piece-access-method . piece-function

piece-function: owner(), name(), point(), moves(), location()

The owner function returns the name of the Player to which the Piece was initially assigned. The return type is string. The name function returns the name of the Piece, also returning a String. The point function returns the point value assigned to the Piece, if one exists. This can be either

of type double or int, as required in the Setup. The moves function returns a Matrix of the move sequence initialized, with type String. The location function returns the coordinate of the Piece in question. If the Piece is on the Board, the Coord will be a valid Board location, and if the Piece is in an *inventory*, the Coord will be (0,0). These five functions are available regardless of the method of access to a Piece. When the Board and a coordinate are used, the *location()* function will return the same coordinate as given to the Board.

For example in tic-tac-toe, finding the location of a black "O" Piece is done using *Player.onBoard("B","O").location()* and returns a coordinate such as (2,2). If multiple Pieces match a given description, the first one in the list will be returned. For example, if a Piece is moved from an *inventory* to *onBoard*, and there are multiple Pieces fitting the description, the first matching one will be moved

6.D Dice Item

A Dice item is created with a name, and the number of sides it has. The following syntax is used when creating a die:

Dice(name-of-dice, number-of-sides)

Dice can be rolled and their rolls return a random int between 1 and the number of sides inclusive. The following syntax is used to roll a specific die.

Dice(name-of-dice).roll()

6.E Setup Declaration

The Setup block of a BoredGames program is where the initial Board and necessary Pieces for the game are specified. The move matrix can also be listed here, and Pieces can be added to the Board in a starting position. The block is enclosed in braces.

Setup { statements }

6.F Rules Declaration

The Rules section of a BoredGames program is made of a series of rules for the game, listed in order of dependence. A rule is declared using the rule keyword followed by an identifier and a colon. The actual rule is expressed using if-else statements. Variables in rules have access to variables initialized in the Play block. Because of the order of dependence, rules can refer to previously listed rules. The rules are called in Play, so the Rules section can have a combination rule to check any number of rules.

Rules { rules-statements }

rule rule-name: { statement }

e.g. *rule r1: { statement }*

rule r2: { statement-using-r1 }

In Rules, error bound checking can be performed with the library function *outOfBounds(new-coordinate)*, and if a move matrix is provided in the Setup, the validity of a Piece's move based on allowed transitions can be performed with the library function *validMove(Piece, new-coordinate)*. These are further explained in the Library Function section.

6.G Play Declaration

The Play section of a BoredGames program is meant to outline the progression of a turn. Output and input functions are used to give the user current information about the Board and take a user's move. BoredGames keeps track of the current player through NextPlayer, which can be called at any time and gives a way to alter the turn sequence. The original Player order is the order in which they were defined in Setup. The end of the game is also called with the function EndGame, which gives a message when certain user-defined win criteria are met. At the end of Play, the entire block is recalled using the previously specified player as the current player.

Play { statements }

7. Library Functions

Move

move(Pieces, Coord)

Move takes the specified Piece on the Board and moves it to the coordinate given. The Piece can be accessed in any way previously shown, through the Board or the Player. If multiple Pieces fit the Piece description, the first one reached will be moved.

Add

add(Pieces, Coord)

Add takes the specified Piece, which would be originally in a Player's *inventory*, and places it on the Board at the coordinate given. The Piece can be accessed in any way previously shown, through the Board or the Player. If multiple Pieces fit the Piece description, the first one reached will be added.

SizeOf

sizeOf(array/Matrix)

This function returns the size of an array or matrix, as an integer or coordinate respectively. It is called with the identifier of a BoredGame supplied array (such as *inventory* or *onBoard*) or user created matrix (such as the Board or move matrix).

Input/Output

input(string)/output(string)

These allow the program to print messages through *output()* and accept input through *input()*. *input(x)* stores user supplied input in the identifier x and *output(string)* prints the string in the parenthesis, and can use a concatenation with a plus (+) to connect strings with strings or strings with identifiers.

EndGame

EndGame(string)

EndGame() prints the message supplied and stops the repeated calls to Play. It ends the program entirely.