BOREDGAMES

Language for Board Games

I. Team:

Rujuta Karkhanis (rnk2112)
Brandon Kessler (bpk2107)
Kristen Wise (kew2132)

II. Language Description:

BoredGames is a language designed for easy implementation of turn-based board games. BoredGames will enable 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 will be integrated together by our complier. BoredGames aims to be generic enough to cover a wide range of board compositions, accessories, and rules.

III. Problems That We Can Solve:

Traditionally developing board games in languages such as C++ or Java is a very time consuming task. A great deal of setup is required and many dependencies need to be dealt with before the rules of the game can even start to be coded. Additionally, much of the code is not reusable from one game to another. BoredGames aims to fix these problems by providing the tools to quickly develop game boards, pieces, and players without the need to deal with the many intricacies that normally go along with these tasks so that the developer can jump right into defining the rules and procedures of the game. This simplified process also allows the user to easily make changes in the game without altering much of the code. We also will allow the user to easily traverse the board and make changes. Board games are often very complicated to create on a computer and we aim to simplify this process as much as possible while still providing the ability for users to develop a wide range of games through the powerful and diverse set of tools within our language. Some simple games that can be implemented in BoardGames include Tic-Tac-Toe, Checkers, Chess, and Monopoly.

IV: Language Use and Syntax:

A. General Structure:

The general structure of every program in our language is as follows:

```
Setup{
	/* General setup such as board, pieces and players goes here*/
}
Rules{
	/* the rules of the game go here. Examples of this are invalid move
checks and  win condition checks.*/
}
Play{
	/* How a player's turn should proceed from start to end goes here*/
}
```

In addition to following the above structure, each program in our language has a few requirements. Every line of the program must end in a semi-colon. Also, comments can be written only in the following format: /* Comment Goes here */. Another detail about our language is that all counting in the language starts at one. Finally, to escape characters place them inside a pair of single quotes.

B. Types and Tools:

The following primitive data types exist in our language: boolean which can be true or false, double which is used for decimal numbers, int which is used to store integers, and String which must be surrounded by double quotes. Along with these primitive types we also have a matrix type which is a standard two dimensional matrix and should be mainly used for storing piece move-sets and the coord type which stores coordinate pairs. We also have a few different tools available to the programmer which should make designing a board game very simple. The first tool is Board which, as its name would suggest, is the game board on which all moves and piece placements will be based on. Board contains the ability to check if a specific tile on the board is unoccupied and also the ability to provide the contents of a specific tile. A Board is required in each game. Our second tool is Players which is where the programmer places all information about the players in the game. Each Player can have a name, an inventory, and a point value, but only the name is required. At least one player is required in every game. The next tool is Pieces. Pieces allows the developer to create game pieces that can be either placed on the board or directly into a particular player's inventory. Pieces have names and optional point values, move-sets and owners. Pieces are not required in a game. Other tools include input, which takes keyboard input from the program's user, and output which displays the current board to the program's user. The final tools are Dice, Timer, and EndGame. Dice rolls a die with a specified amount of sides, Timer is an ascending or descending countdown, and EndGame ends the game, displays the board and displays a message to the users.

C. Operators:

All of the following operators are defined in our language with the same functionality they have in most standard languages. || is used for the logical or, && is used for the logical and, = is for

assignment, + is for addition and the concatenation of strings, - is for subtraction, * and / are for multiplication and division respectively, % is for the modulo operation, and ! is the logical not operation. We also have the following comparison operators: < is the less-than operator, > is the greater-than operator, == is the equality operator where equality is checked by value, <= is the less-than or equal-to operator, >= is the greater-than or equal-to operator, and != is the not equal to operator.

D. Program Control Flow:

In addition to the above we provide the following tools for program control: if and else statements, which behave as they do in other languages. The if statement must evaluate to true or false, the else is not required. If-else is not part of our language. The if and else statements syntax is as follows:

```
if(Boolean Statement){
	/* code here*/
}
else{
	/*code here*/
}
```

We also have a loop that can be used to repeat lines of code. There are two different types of loops in our language. The first takes a boolean statement and continues as long as the statement is true. The syntax is as follows:

```
loop(boolean statement){
	/* code here*/
}
```

The second takes a start and an end number and iterates through the loop end number – start number + 1 times. The syntax of this is as follows:

```
loop(start number:end number){
	/* code here */
}
```

In addition to using the start number:end number format for loops this format when describing coordinates of the Board.

E. Setup Section:

Now we will look at what occurs in each section of a program in our language. First we will look at the setup section. Since we are a board game language, we require the developer to define the size of the board using the following syntax:

```
Board(Number of Rows, Number of Columns);
```

This should be the first line in the setup section of the program. Next users should define the players by creating players using syntax such as:

```
Players.add(Player Name, Point Value *Optional*);
```

Once the players have been created, pieces can be added into the player's inventory or onto the board directly. Adding into a specific player's inventory can be done by using the following syntax:

```
Pieces.add(Player Name, Piece Name, Quantity of Piece);
```

Adding a piece onto the board can be done using the following:

```
Pieces.add(Player Name, Piece Name, Quantity Of Piece, Board Row, Board Column);
```

A set of valid moves can also be added to a piece using the following syntax:

```
Pieces.add(Player Name, Piece Name, Quantity Of Piece, Board Row, Board Column,
Moves);
```

Where Moves is a matrix where each row is sequence of valid moves that can occur on a single turn. Each element of the matrix must be from the following set {up, down, left, right, diagLeftUp, diagLeftDown, diagRightUp, diagRightDown}. An example of a move matrix for a knight in chess would be the following:

$$\begin{pmatrix} up & up & left \\ up & up & right \\ down & down & right \\ down & down & left \\ right & right & up \\ right & right & down \\ left & left & up \\ left & left & down \end{pmatrix}$$

Any move made by a user during runtime is automatically checked against the move matrix to make sure the move is valid.

F. Rules Section:

The Rules section of each program contains a set of rules that the programmer can check against during runtime to make sure the game proceeds normally. There need not be any rules defined in a game, but most games will at least require a rule for winning/losing. Each individual rule is essentially a function that can only return true or false. Every rule must start out in the following manner:

```
rule ruleName:  /* rule goes here */ ;
```

An example of a completed rule that checks if the tiles located at spot 1,1 on the board is unoccupied is:

```
rule r1: Board(1,1).unoccupied();
```

An important detail about rules is that they can only refer to other rules which are defined above them. Referring to rules defined below them will generate an error. A rule can be referred to by just using their names as if they were a boolean. The following code would then be valid in the play section if the example rule above is defined:

```
if(r1){
	EndGame("Board Spot 1,1 is empty. Game Over");
}
```

G. Play Section:

The Play section of the program is where the game procedure for each player's turn is defined. The language automatically keeps track of which player is currently taking a turn, and a reference to Player in the Play or Rules section accesses this player. After all the code in the Play section has been executed, the program calls Play again for the next Player's turn. This is why the NextPlayer (syntax is just `NextPlayer;`) command exists. NextPlayer automatically adjusts the current Player according to the order in which Players were initialized in the setup section. This gives the program writer the power to create situations in which a player gets to take multiple turns in a row, such as when rolling doubles in Monopoly, or a player's turn is skipped. The only way for Play to exit is for the EndGame command to be called. Upon its execution, the game is over and the program ends. EndGame's Syntax is as follows:

```
EndGame(Message To Send);
```

V. Code For Tic-Tac-Toe:

```
Setup{
	Board(3,3);  /* defines a 3x3 board */
	Player.add("A");  /* add a player; owner is A */

	Pieces.add("A","X",6));  /* owner A has 6 X pieces */

	Player.add("B");  /* add a player; owner is B */

	Pieces.add("B","O",6));  /* owner B has 6 O pieces */
}

Rules{
	/* bounds check */
	/* check left and right bounds */
	rule r1: if(in.x < 3 && in.x > 0) {
			return true;
```

```
            }
            else {
                return false;
            };
    /* check top and bottom bounds */
    rule r2: if(in.y < 3 && in.y > 0) {
                return true;
            }
            else {
                 return false;
            } ;
    /*check if both bounds conditions are satisfied*/
    rule r3: if(r1 && r2) {
                return true;
            }
            else {
                return false;
            } ;

    /* check if a tile is unoccupied */
    rule r4: Board(in.x,in.y).unoccupied();;

    /* check if the board is full */
    rule r5: int i = 1;
            int j = 1;
            boolean full = true;
            loop (i<4) {
                loop (j<4) {
                      if (Board(i,j).unoccupied()) {
                            full = false;
                      }
                      j = j+1;
                }
                i = i+1;
            }
            return full;;


    /* win conditions */
    /* horizontal win check*/
    rule r6: int i = 1;
            int j = 1;
            int val = 0;
            boolean win = false;
            /* every row */
            loop (i<4) {
                /* every col */
                loop (j<4) {
                      /* checks if the owner of the first piece on the board at
                       i,j is owned by the current Player*/
                      if (Board(i,j).Pieces(1).owner == Player ) {
                            val = val+1;
                      }
                      j = j+1;
                }
```

```
                    if (val==3) {
                          win = true;
                    }
                    else {
                          val = 0;
                          i = i+1;
                    }
                }
                return win;;

      /* vertical win check*/
      rule r7: int i = 1;
               int j = 1;
               boolean winChance;
               /* every column */
               loop (i < 4){
                    winChance = true;
                    /*every row*/
                    loop (j < 4) {
                          /* checks if the first piece on the board at j,i has the
                          same name as the first piece in the current player's
                          inventory */
                          if (winChance == true && Board(j, i).Pieces(1).name() !=
Player.inventory(1).name()) {
                                winChance = false;
                          }
                          j = j + 1;
                    }
                    if (winChance == true) {
                          return true;
                    }
                    i = i + 1;
               }
               return false;;

      /* diagonal win */
      rule r8: int i=1;
               int a=0;
               /* goes through each row */
               loop(1:3) {
                    /* checks if the first piece on the board at location i,j has the
                    same name as the first piece in the current player's inventory */
                    if (Board(i,i).Pieces(1).name == Player.inventory(1).name()) {
                          a = a+1;
                    }
                    i = i + 1;
               }
               if (a==3) {
                    return true;
               }
               else {
                    return false;
               };

      /* checks if any of the three win contions are satisfied */
```

```
      rule r9: if(r6 || r7 || r8) {
                  return true;
               }
               else {
                  return false;
               };
}

Play{
      /* displays the current board on the screen */
      output(Board);
      /* takes keyboard input from the user and stores it in the coordinate in */
      input(coord in);
      if(r3){
            Board.add(Player.inventory(1), in);
            if(r9){
                  /* break, display board and output  message */
                  endGame("Player " + player.owner + " Won");
            }
            else{
                  if(r6){
                        /* break, display board and output message*/
                        EndGame("Board Full! No Winner!");
                  }
            }
            NextPlayer;
      }
}
```