



a language for designing board games

Prepared by Team Strat  
May 9, 2010

Lauren Pully  
*Project Manager*  
lep2128@columbia.edu

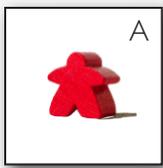
Jesse Bentert  
*Language Guru*  
jrb2137@columbia.edu

John Graham  
*System Architect*  
jwg2116@columbia.edu

Daniel Wilkey  
*System Integrator*  
dgw2109@columbia.edu

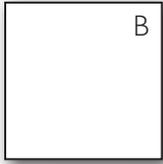
Yipeng Huang  
*Tester & Validator*  
yh2315@columbia.edu





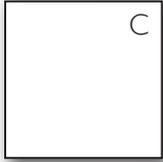
A

Introduction



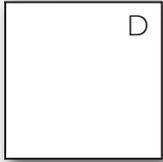
B

Language Tutorial



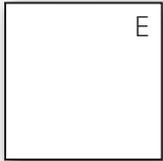
C

Primer on Organizing  
ROLL Code



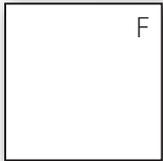
D

Language Reference  
Manual



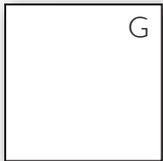
E

Project Plan



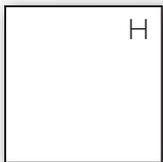
F

Language Evolution



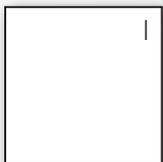
G

Translator  
Architecture



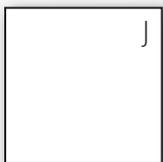
H

Development  
Environment



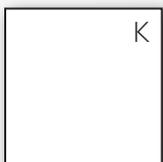
I

Test Plan



J

Conclusions



K

Appendix

# Introduction



# ROLL

## *a language for designing board games*

### ***Introduction***

ROLL is an easy-to-use programming language designed to facilitate the implementation of children's board games. Aspiring ROLL developers are not required to have an extensive programming background, only a creative and unique idea for a game. Well-implemented games in ROLL contain a custom game board, multiple players, an element of chance, and choices available to the players that influence the outcome of the game. A program written in ROLL could serve as the back-end of a complete game program that may include a graphical user interface implemented in another language.

### ***Data Types***

*Player:* An end-user playing the game that has properties including a name and pieces. Players have the ability to execute moves and are presented with choices on their turns during game play. Depending on the specific game, the choices the player makes should directly impact their odds of winning.

*Piece:* A property of a player that interacts with the board. It is aware of the current tile that its pieces occupy.

*Game:* An abstract entity that comprises all aspects of the ROLL framework. A game has a board, a list of players, and either a set of dice or a deck of cards. A game also contains an associated logic that dictates the manner in which the game progresses, the steps required to win it, and the goal state.

*Board:* A board is an entity that maintains the state of the current game. A board has tiles, occupying pieces, and associated actions on each tile.

*Tile:* A property of a board that contains a unique index, the location of successor tiles accessible to a player during a move action, and an associated action-listener tied to the event of landing upon it.

*Dice:* Dice are entities that generate random numbers, which then influence the game's progression in some way. The ROLL programmer decides how the random number generated during a player's turn influences the player's progress in the game.

*Deck:* Deck is an aggregation of cards, one of which may be drawn during a player's turn. The ROLL programmer decides how a card influences the player's progress in the game.

### ***Features***

Roll is motivated by the notion that many people have creative ideas for computer board games, but not the time nor the patience to write their own game logic framework. ROLL helps

programmers by providing a generalized game framework for running games, allowing programmers to focus their efforts on the specifics of their ideas. Programming is hard, and we want to make creating new board games easy. By creating a language that gives programmers the flexibility to create complex board games, the possibilities are unlimited.

### **Simple**

ROLL is intuitive. Programmers can pick it up and create games without extensive coding experience. The syntax is easy to learn and a board game's features can be easily modified.

### **Flexible**

A programmer using ROLL can implement games with varying degrees of complexity. A simple game can have players take turns rolling a die, determining a winner based on the first player to reach an end location. However, ROLL allows for more complexity: players can have more input into how a piece moves, the dice can be replaced by deck of unique cards, and even individual tiles in the board can have special properties. Game tiles, for example, can cause players to draw cards, move players to specific tiles, modify a player's property (i.e. amount of money), or even win the game. A programmer drawing upon complex elements of ROLL can thereby introduce strategy to winning the game.

### **Dynamic**

ROLL provides the freedom to leave decisions up to the player, allowing prompts throughout game play for user input. For example, the player might be able to choose how many people are going to play the game, or he might be able to decide in which direction he wants to move on a particular turn.

### **Extendable**

A game program created using ROLL would be playable through the command-line interface. The game developer using ROLL can choose to create a more refined textual or graphical user interface. ROLL facilitates such extensions by outputting game state information and player input prompts in a consistent and easily readable format. This output can integrate seamlessly with a variety of possible front-ends written in any popular graphics-capable language.

### ***Using ROLL***

#### **Property-Oriented**

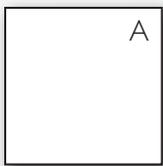
A game written in ROLL would be organized into sections of code defining properties of the individual elements that comprise a game. A programmer using ROLL would define properties of the board, tiles, players, dice, and deck, and then specify the actions that are available to the player.

## **Parsing and Compilation**

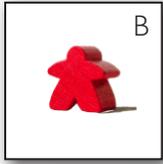
The logic framework for ROLL is written in Java. Source code from ROLL is compiled into a set of Java class files before execution. Java serves as a robust and high-performance platform for running programs written in ROLL. The compiled program created in ROLL benefits from the platform neutrality and interpreted nature of programs that run on the Java virtual machine.

## ***Summary***

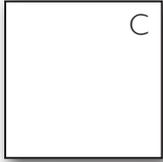
ROLL simplifies computer board game creation for the everyday programmer!



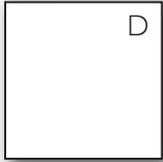
Introduction



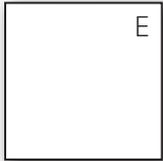
Language Tutorial



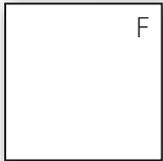
Primer on Organizing  
ROLL Code



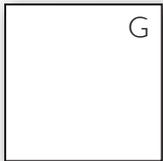
Language Reference  
Manual



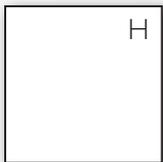
Project Plan



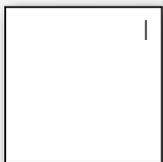
Language Evolution



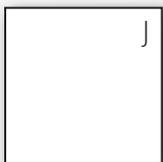
Translator  
Architecture



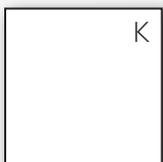
Development  
Environment



Test Plan



Conclusions



Appendix

# Language Tutorial



# ROLL Language Tutorial

Lauren Pully - lep2128  
Jesse Bentert - jrb2137  
John Graham - jwg2116  
Dan Wilkey - dgw2109  
Yipeng Huang - yh2315

- 1. Introduction .....2
- 2. The Default Game .....2
- 3. The Hello World Program.....3
- 4. The Default Game: What Lies Beneath .....7
- 5. Raising the Stakes: a More Complex Program .....15
- 6. Conclusion.....25

# 1. Introduction

This tutorial will guide someone unfamiliar with the ROLL language through the process of writing basic programs. It takes a top-down approach, introducing the most basic program to start and progressing to a few that are bit more complex. From a high level perspective, a ROLL program is comprised of three sections, which we will refer to as blocks: Players, Board, and Dice/Deck. Each block can be customized by the programmer to implement the specific rules associated with his or her game. The following sample programs will demonstrate how to create board games using ROLL. We begin with the simplest possible program, one containing no body.

## 2. The Default Game

### Program 1

```
1Game Default {}
```

The program shown above is clearly not very interesting in terms of code. The first word, `Game`, indicates the declaration of a new ROLL program. The second word, in this case `Default`, specifies the name of the board game (i.e. any word may go here). Within the curly braces would be the implementation of the game; however in this case none is provided. The programmer need only define those blocks he wishes to tailor to the specifics of his game. For each of the rest, the ROLL framework's corresponding default implementation will be used in its place. Note that for each of the blocks defined, their order is fixed: Players followed by Board followed by Dice/Deck (A slash is used between Dice and Deck to indicate at most one of the

two can be implemented). If a ROLL program is compiled without any implementation (such as this one), all three default blocks are generated, yielding the so-called 'Default Game'. The default ROLL game boasts 10 tiles, a single die with six faces, and can accommodate anywhere from two to six players. Players will sequentially take turns rolling the die and moving their piece forward the number of spaces shown on the die. The game ends when a player reaches the tenth and final tile and is declared the winner. When this program is run, an entire game is simulated and printed out. The only user input required for this game is the number of players and their names. The output of this game would be similar to the following:

Figure 1

```
*****Default*****
How many people are playing this game?
Enter a number between 2 and 6: 2
Please enter player # 1's name:
Jesse
Please enter player # 2's name:
Dan
It is Jesse's turn
Jesse rolled a 6
Jesse moved piece 0 to tile 6

It is Dan's turn
Dan rolled a 2
Dan moved piece 0 to tile 2

It is Jesse's turn
Jesse rolled a 3
Jesse moved piece 0 to tile 9
Jesse WINS!!!
```

### 3. The Hello World Program

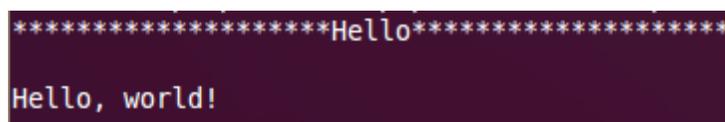
Many languages begin with a "Hello World" program. Although such a program is not particularly relevant to this language, here is what it would look like.

## Program 2

```
1 Game Hello {
2     //This is a comment
3     Players {
4         define setupPlayers() {
5             print("Hello, world!");
6         }
7     }
8     Board {
9         NumTiles = 10;
10        define preRoll(int playerID) {
11            declareWinner();
12        }
13    }
14 }
```

The output of this program is essentially just the text string “Hello, World”; however it is preceded by a line that starts every game: the name of the game being played surrounded by asterisks. Although there are a few more layers of complexity here that will be discussed later on, the general idea of the program is simple. It starts a game, displays “Hello, world!” and then begins the first player's turn. The beginning of the turn is implemented to end the game and the program terminates.

Figure 2



```
*****Hello*****
Hello, world!
```

Now let's discuss some of the syntax of the Hello World program. The first line of code within the game definition is a comment. As with popular languages like C and Java, two consecutive forward slashes indicate a line comment. All characters from the slashes to the end of the line are ignored by the compiler. Note that there are no block comments in ROLL.

Following the comment are definitions of two of the three possible blocks: `Players` and `Board`. Notice that blocks in ROLL do not accept name parameters, as was seen with `Game`.

To begin a new block, one needs only to provide the block title, followed by a set of curly braces (which enclose the implementation of that block). Each of the blocks contains several associated statement categories that the programmer may choose to provide. As will be seen later, these statements can take three forms: setting block-level fields, creating objects, and defining functions. Most block statements can be omitted, in which case a default value is used for them (the one exception comes in the Deck block and is discussed later). Indeed, for the Hello World program, most of the block-level statement categories of `Players` and `Board` are left out.

In the `Players` Block, one statement category not omitted is the `setupPlayers()` function, which is where the programmer gleans information from the user about the current set of players. For example, here one could obtain the number of players for the current game as well as their names. When the program executes, this function will be the first called. `setupPlayers()` is an example of what we call *dynamic functions*: functions that the programmer is allowed to provide implementation for, but cannot ever call himself. These are called automatically by the ROLL framework at specific times during the game's flow of control. The names and signatures of dynamic functions are predefined; the programmer may not define and implement their own custom functions. Refer to the LRM for a complete list of dynamic functions, along with descriptions of the arguments they accept, in which blocks they may be defined, and at what point in the flow of control these functions are called. Note the syntax for defining dynamic functions in ROLL, as seen in `setupPlayers()` above. The keyword `define` is followed by the name of the function and then a set of parentheses containing the formal parameters. In ROLL, `setupPlayers()` takes no arguments.

In this case, `setupPlayers ( )` does not do very much. It calls `print ( )`, a function which prints text to the standard output. This is an example of the second type of function in ROLL: *static functions*. These are functions that the programmer cannot define himself; they may only be called (within dynamic functions). To call `print ( )`, a text string (surrounded by double quotes) is explicitly provided as the actual parameter. All print statements are followed by a newline when output.

Notice the manner in which lines of code are terminated. All lines conclude with semicolons, save for one general exception: functions and block definitions take curly braces to delimit their beginning and ending points. See that lines 1 and 14 do not end in semicolons, since `Game` meets this criterion.

On line 8 the `Board` block is defined. This is where one provides the details of their game board. The board-level field `NumTiles`, seen on line 9, specifies the number of tiles on the board. In our Hello World program, this field does not serve any purpose, except to demonstrate how block-level fields are set. On line 10 we define the `preRoll ( )` function (another dynamic function), which is executed before each player's turn. This function is meant to give the programmer an opportunity to handle any actions that must be carried out every time a new turn starts. Here, it just calls `declareWinner ( )`, a static function that terminates the current game. This call is made to suppress the default game from running (which otherwise always happens, as shown by the simulated game output in figure 1) by declaring a winner before any player has a turn. This will cause only the text "Hello, world!" to be printed and none of the standard game state information. This program does not actually play a game and only serves for demonstration.

## 4. The Default Game: What Lies Beneath

Although Program 1 of this tutorial does technically provide all the necessary implementation to construct the default game (i.e. none), let's look at a ROLL program one could write that would provide the exact same functionality without relying on any default behavior. Each of the three main blocks will be discussed individually, albeit out of order. To construct a working program out of them, you would place their definitions (Programs 3-3, 3-1, 3-2) into a single file, within an enclosing named Game object. Notice that we listed Program 3-3 first in the above ordering. This refers to the `Players` block, which as discussed before must come first in the program. However, due to some complex productions it contains, this block will be discussed last herein. We begin with the `Board` implementation.

Program 3- 1

```
1 Board {
2     NumTiles = 10;
3     make Tile(id: 0, next: 1, prev: 0);
4     make Tile(id: 1, next: 2, prev: 0);
5     make Tile(id: 2, next: 3, prev: 1);
6     make Tile(id: 3, next: 4, prev: 2);
7     make Tile(id: 4, next: 5, prev: 3);
8     make Tile(id: 5, next: 6, prev: 4);
9     make Tile(id: 6, next: 7, prev: 5);
10    make Tile(id: 7, next: 8, prev: 6);
11    make Tile(id: 8, next: 9, prev: 7);
12    make Tile(id: 9, next: 9, prev: 8);
13
14    define goalCheck(int playerID, int tileID)
15    {
16        if(tileID == 9)
17        {
18            declareWinner(playerID);
19        }
20    }
21 }
```

On line 1, we define the `Board` block in the same fashion as we did for the Hello World game. On line 2 we set the `Board` block's only field, `NumTiles`, which represents the number of tiles there will be on the board. Beginning on line 3, the individual tiles of the board are defined. All tile instantiations begin with the keyword `make`, which indicates that an object is being constructed. The next word specifies the kind of object; the one in this example is `Tile`. `Tile` takes a variable number of arguments for its instantiation, which is why arguments are named individually. For each tile, the first argument that is set is the `id` argument. `id` is a unique integer value between 0 and `NumTiles-1` used to identify that `Tile`. The next two specified arguments are `next` and `prev` (previous). These represent the ids of the following and preceding tiles, respectively, in the context of a sequential movement of a piece along the tiles.

For example, when a player rolls, his piece would proceed from its current tile location to the tile with `id` matching the value stored in that tile's `next` attribute. The player would then continue as such, following the chain of `next` links, until having moved the appropriate number of tiles. Conversely, if moving backwards along the list of tiles, the piece would proceed in precisely the same manner, but this time following the chain of `prev` links. Named argument assignment is done with the colon operator using the general form, `<argument name> : <argument value>`. The `id` of the tile on line 6 is therefore assigned the value 3. Additionally, the `id` of its `next` attribute is set to 4 and the `id` of its `prev` attribute is set to 2. All of the tiles follow this general form, except for the first one (on line 3) and the last one (on line 12). For the tile on line 12, the `next` attribute of this tile is set to itself, 9. The reason for this, which will become clear in discussing the `roll()` function, is that we do not want the piece to go off of

the end of the board. If someone rolls a 6 from tile 8, for instance, then his piece will move to tile 9 and stay there for the remainder of the 5 moves. For the same reason, the `prev` attribute of the tile on line 3 is also set to itself. Thus, the idea behind the tiles in the `Board` block is pretty intuitive. Pieces are moved according to the value on the die (or on the card drawn) and iterated along the list of tiles on the board. Once a player's move has completed, the `goalCheck()` dynamic function is called.

You can see the syntax for defining `goalCheck()` in Program 3 is exactly the same as the `setupPlayers()` definition in Program 2. The arguments here are the id number of the player whose turn it is (`playerID`), and the id of the tile on which the player has landed (`tileID`). Note that here, and in all dynamic functions in `ROLL`, the programmer has no control over the formal parameters or their types. So for `goalcheck()`, the function has to be called with the parameters shown. Line 16 shows the only conditional statement in `ROLL`, the `if-then-else`. The condition here checks if the variable `TileID` is equal to 9. The parentheses to the right of the `if` contain a condition to test that must evaluate to true or false. If the expression evaluates to true (in this case indicating the person is currently on the last tile), then the body of the conditional is executed. In the body of this `if` statement, we again see the `declareWinner()` function. However, this time it contains an argument: the id of the winning player. In this simple default game, the first person to reach the 10th and final tile is declared the winner.

Note that in writing the `Board` block, we first set the block-level fields (just `NumTiles`), then created objects (`Tiles`), and lastly defined a function (`goalCheck`). All blocks in `ROLL` must follow this order for statement categories: fields, objects, and then functions.

Now we move our attention to the definition of the `Dice` block for the default game. The programmer decides the number of dice in the game and the number of faces each die will have.

### Program 3- 2

```
1 Dice {
2     make Die(faces: 6);
3
4     define roll(int amountRolled, int playerID) {
5         print(PlayerList[playerID].name | " rolled a " | amountRolled);
6         move(playerID, 0, amountRolled);
7     }
8 }
```

The `Dice` block has no block-level fields, so we start with its objects. For each die the programmer wants in his game, he has to explicitly make an individual `Die` object; this is done by typing `make Die()` once for each die. The number of faces on the die is declared by setting the value of the `faces` attribute (line 2). For games with multiple dice, the sum of the values rolled on all the dice determines the total roll for a given player in a single turn. Each roll of a die is simulated by randomly generating a number from 1 to the number of faces on that die. Since all of the game's dice are rolled for every player's turn, the programmer can adjust the relative probabilities of different total values occurring through the number of dice and the number of faces on each die. For example, if there are two dice declared, each with six faces, then the probability of rolling either a 2 or 12 is very small (1 in 36), while the probability of getting a 7 is much higher (1 in 6). In the game shown, there is a single `Die` declared, so each value has equal probability.

Within the `Dice` definition, the programmer can implement the `roll()` function, shown on lines 4-7. `roll()` accepts two arguments, the amount that was rolled and the `playerID`

whose turn it is. This function serves as the intermediary between rolling the dice and moving the player's piece. Here, the programmer has the opportunity to vary how a value of the die is interpreted. For example, rolling a certain value might allow the player's piece to move backwards. If a player's piece or pieces are to be moved during a roll, then `move ( )` or `moveReverse ( )` should be called; these are static functions that execute the move action.

`move ( )` accepts three arguments: the id of the player whose piece is being moved, the id of his piece to be moved, and the number of spaces to move it. All ids in `ROLL` are successive integers beginning with 0. In the default case, each player is limited to one piece, so 0 is used to refer to the first (and only) player piece. Lastly, since the number of spaces to move the piece is exactly the amount rolled, it is passed directly as an argument to the `move ( )` function.

The final block that we discuss is `Players`. Here the programmer specifies the minimum and maximum number of players, the number of pieces each player has, and the location on the board where each player's piece(s) begins and ends. Additionally, as alluded to in the discussion of the Hello World program, any run-time player attributes that need to be recorded, such as player names and the exact number of people playing the current game, may be set in the `setupPlayers ( )` method, shown below.

### Program 3- 3

```
1 Players {
2     MaxPlayers = 6;
3     MinPlayers = 2;
4     NumPieces = 1;
5     StartOn = {0,0,0,0,0,0};
6     define setupPlayers()
7     {
8         print("How many people are playing this game?");
9         promptRange(NumPlayers, MinPlayers, MaxPlayers);
10
11        for(int i : {0 ~ NumPlayers-1})
12        {
13            print("Please enter player # " | i+1 | "'s name:");
14            promptName(PlayerList[i].name);
15        }
16    }
17 }
```

We start by setting the block-level fields in lines 2 through 5. Note that they must appear in this order (although you may omit any if you want to use its default value). In lines 2 and 3, the maximum and minimum number of players allowed is set. In line 4, the number of pieces per player is defined by setting the field `NumPieces`. The starting location of each player is set on line 5, using a new construction. The notation seen here, a comma-delimited list of integers surrounded by curly braces, is the general form for in-place integer array instantiation. In this case we have created an array of six integers (because there are at most six players), all of value 0. This means that to start the game, every player's piece is placed on the 0th tile. Note that since the minimum number of players in this particular game is only two, it is possible that not all six values placed in the array will be used for a given game. Only the first  $n$  values are used in a game of  $n$  players. The first number placed in this array is the 0th index of the array, and thus corresponds to the player with `id 0`. For example, if a 2 is placed in the fourth position of

the array, then all pieces of the player with `id 3` (4 minus 1, to account for arrays beginning at index 0) will begin on the tile with `id 2`.

The `Players` block has no objects that may be created, so we proceed to its dynamic function. In line 6, the `setupPlayers()` function is defined. In the default game, this is where the programmer asks how many players there are, and what each of their names is. Line 8 uses the `print()` statement, seen before. In line 9 we introduce one of the three main input constructions, `promptRange()`, which accepts three arguments. The first argument is the integer variable to be set by user input. In this case, `NumPlayers` is a block-level field of `Players` that should be set for every played game. The second argument is a lower limit to the value that may be entered. The third argument is an upper limit to this value. Therefore, because `MinPlayers` is 2 and `MaxPlayers` is 6, the user must enter a number between these two values, inclusively. The preceding `print()` statement in line 8 is meant to give context to the selection provided in line 9.

The loop construct in `ROLL` is called in line 11, the `for-each` loop. The general form of a `for-each` loop in `ROLL` is `for( int <variable> : <array> )`. Each value in the array to the right of the colon is sequentially given to the variable to the left of the colon, and the block of code within its curly braces is executed once for each of these values. Notice from the required variable type, the programmer is only allowed to iterate over integer arrays in `ROLL`. We see here new shorthand for in-place instantiation of an integer array. The array is populated with all integer values between the number to the left of the tilde and the one to the right of it, inclusively. For example, if `NumPlayers` were set to 4, then this array would take on the values 1, 2, 3, and 4. In this case, a primitive, 1, is used as the lower bound and an arithmetic

expression using a variable, `NumPlayers-1`, is used as the upper bound. Any combination of these is acceptable for both bounds.

In the body of the `for-each` loop, on line 13, we see another `print()` statement. As before, it contains a text sequence; however, several arguments are present this time, each separated by `|` (vertical bar) operators. This operator in ROLL is used for text concatenation. When placed between text fields (which are surrounded by double quotes), the operator appends the second onto the end of the first. When an argument is not encased in double quotes, it is assumed to be either a primitive or a variable. In this case, it is converted to its textual equivalent and then concatenation proceeds as before. As seen with the second concatenated parameter, the addition operator in ROLL (`+`) is different from the concatenation operator (`|`), thus all possible arithmetic expressions will be parsed unambiguously, unlike some other popular general-purpose languages.

Line 13 shows the second method of input, `promptName()`. This function accepts a single argument, which has to be a player's name attribute (player attributes are discussed in the next paragraph). It allows the user to enter any sequence of characters with which to set the field. As before, the print statement indicates the context for the value being set.

Within the body of the `promptName()` function call, a new construction is seen for accessing objects by their ids. We now introduce the field `PlayerList`, which is an array of the player objects. It is never set explicitly by the programmer (it is created behind the scenes), but the programmer has access to it in any of the three blocks. One can access a specific player object from this list by specifying the id of that player within square brackets that follow the list name, a la an array index in programming languages like Java. So `PlayerList[2]` returns

the player object whose id is 2. Moreover, each `Player` object has two accessible public attributes that can be accessed by appending a period and then the attribute title. The first of these is `name`, a text attribute in which a player's name may be stored (line 14). The other is `PieceList`, the list of that player's piece objects. Individual pieces are specified in the same manner that individual players are selected from the `PlayerList`: by piece id in square brackets. For instance, a programmer can refer to the first player's second piece using `PlayerList[0].PieceList[1]`. For a complete list of ROLL objects and their accessible attributes, please refer to the language reference manual.

And thus we have finished the default definitions and implementations of blocks: `Players`, `Dice`, and `Board`. The above block implementations provide identical functionality to leaving all of them out. Knowing this, it would be unnecessary to write a ROLL program taken exactly from the code provided above. From here on out, the Language Tutorial will discuss the more advanced (and certainly fun!) features of ROLL.

## **5. Raising the Stakes: a More Complex Program**

To demonstrate the more advanced ROLL concepts, we will show a variation of the popular board game Chutes and Ladders. Chutes and Ladders is another progress game in which the first player to reach the end wins. This game features ladders on some spaces, which when landed upon cause the player to move ahead a certain number of spaces. Conversely, some tiles feature chutes, which cause the player to go back a certain number of spaces. We have also provided additional functionality not found in the classic Chutes and Ladders game to illustrate some other ROLL features. For example, each `Player` will have two pieces instead of one, and

both of the `Player`'s pieces must reach the last tile in order to win. Additionally, instead of rolling a die to determine how many spaces to move, we are going to instead use a simple deck of cards to illustrate how a programmer could use the deck block in place of the dice block.

This program begins with another `Players` definition.

#### Program 4- 1

```
1 Players {  
2     GUI = 1;  
3     NumPieces = 2;  
4     FinishOn = {10,10,10,10,10};  
5 }
```

As you can see above, this is a skeletal implementation of the `Players` block. Many of the fields seen in the Default Game `Players` block (Program 3-3) are omitted because we are satisfied with the underlying default implementations. The first line provided in the above block is the `gui` toggle variable. By default, this is set to 0, which causes the game to be played from the standard out, as seen with Figures 1 and 2. In this case, we are explicitly setting its value to 1 in order to allow this game to use a default `gui` that the `ROLL` language provides. Screen shots of this interface will be shown after full discussion of program 4. The next value set in this block is the `NumPieces` field, which we set to 2.

We now introduce a new field known as `FinishOn`. This parallels the `StartOn` array seen earlier. Instead of aggregating the indices of tiles on which a player's pieces will begin, this array contains the indices of tiles on which their pieces will end. The interpretation of this concept is left up to the user. Our intention is that this array will be used in the `goalCheck()` function to determine whether or not a player has won. Clearly, this array is most pertinent in games whose object is to reach some ending tile. Because Program 4 is such a game, its

goalCheck ( ) function (which will be shown later) makes use of this variable. We now shift our attention to the Board block.

#### Program 4- 2

```
1 Board {
2   NumTiles = 11;
3   make Tile(id: 1, next: 2, prev: 0, accessible:{5}, landsOn: ladder);
4   make Tile(id: 3, next: 4, prev: 2, accessible:{8}, landsOn: ladder);
5   make Tile(id: 6, next: 7, prev: 5, accessible:{0}, landsOn: chute);
6   make Tile(id: 7, next: 8, prev: 6, accessible:{4}, landsOn: chute);
7
8   function ladder = define landsOn(int playerID, int pieceID, int tileID)
9   {
10      print("Hooray! " | PlayerList[playerID].name | "'s piece number " | (pieceID + 1)
11          | " went up the ladder to tile " | TileList[tileID].accessible[0]);
12      jump(playerID, pieceID, TileList[tileID].accessible[0]);
13  }
14
15  function chute = define landsOn(int playerID, int pieceID, int tileID)
16  {
17      print("Oh no! " | PlayerList[playerID].name | "'s piece number " | (pieceID + 1)
18          | " went down the chute to tile " | TileList[tileID].accessible[0]);
19      jump(playerID, pieceID, TileList[tileID].accessible[0]);
20  }
21
22  define goalCheck(int playerID, int tileID)
23  {
24      int gameOver = 1;
25      for(int p : {0 ~ NumPieces - 1})
26      {
27          if(PlayerList[playerID].PieceList[p].occupiedTileID != FinishOn[playerID])
28          {
29              gameOver = 0;
30          }
31      }
32      if(gameOver != 0)
33      {
34          declareWinner(playerID);
35      }
36  }
37 }
```

This game is declared to have 11 tiles (set on line 2); however, only 4 of these 11 are explicitly instantiated. The tiles not instantiated are automatically generated using standard attributes. By standard attributes, we mean that the `next` field is set to one greater than the `id`, and the `prev` field is set to one less than the `id`. The only exceptions to these rules are the first tile, where the `prev` index is set to itself, 0, and the last tile, whose `next` index is also set to

itself, `NumTiles - 1`. These exceptions are made because it is assumed that the board has definite beginning and ending points.\* By default, one of these standard attribute tiles is created for every missing `id`. In this case, the `id`'s 0, 2, 4, 5, 8, 9, and 10 are not instantiated, and so a standard attribute tile will be generated for each of these. We now introduce two new tile attributes: `accessible` and `landsOn`.

The `accessible` attribute allows you to associate an integer array with a tile. Its argument is the array instantiation syntax seen before. Our intention is that the programmer may place tile `ids` in the array, which may be of use to indicate tiles besides `next` and `prev` that are accessible when special conditions are satisfied. In this case, if a player's turn ends on a ladder or chute, then his piece should "jump" to a separate tile. Since there is only one tile to which he can move, our `accessible` array is of length 1. The number in the `accessible` array provides the `id` of the end tile of the chute or ladder. To handle this special action, the programmer must implement a `landsOn ( )` function, whose name is given as the final attribute of these instantiations.

The `landsOn ( )` dynamic function is called once a player's piece has finished moving for a given turn. The function is specific to each tile, and is only called if the tile landed upon has a `landsOn ( )` function specified. In this program, we define two `landsOn ( )` functions, one for ladder tiles and one for chute tiles.

---

\* If instead you wanted to make the board circular, you would need to set the `next` field of the last tile to the first tile and the `prev` field of the first tile to the last tile (in the style of linked lists).

The `landsOn ( )` functions are our first example of named function definitions. You can see on line 8 that the first of these is named `ladder`. The reasoning for naming the `landsOn ( )` functions is that oftentimes the same function will be associated with several tiles. This allows for code reuse, as only the function name is passed as an argument to each `Tile`.

The `landsOn ( )` function accepts three arguments: the id of the player who is moving, the id of the piece that has landed, and the id of the tile on which it has landed. As always, the programmer has no control over the arguments. In the print statement on lines 10-11, text concatenation is done as before. Note that the last concatenated text sequence references the id of the target tile through the accessible array. In this case, there is only one such tile, so it's accessed by `id 0`.

Line 12 introduces the `jump ( )` function. Similar to `move ( )` and `moveReverse ( )`, this construction allows a programmer to change the occupied tile of a player's piece. In contrast to its two counterparts, `jump ( )` moves the piece directly to the target tile, without traversing the normal chain of next/previous links. The piece should jump directly to the target tile. This behavior is achieved by accepting the id of the target tile as the third argument, rather than accepting the number of spaces to traverse (which is what `move ( )` and `moveReverse ( )` do). Additionally, the `landsOn ( )` function of the target tile is not called after the jump is executed. The first two arguments allow specification of the piece to be moved through the id of its owning player and the id of the piece. Due to its behavior, `jump ( )` is most commonly called from within the `landsOn ( )` function.

Notice that in the chute `landsOn ( )` function, the `jump ( )` call is exactly the same as its counterpart in `ladder`. The only difference between these two `landsOn ( )` functions is the

specifics of their `print()` statements. One indicates the action is favorable (ladder) while the other informs the user of a less than desirable result (chute).

Beginning on line 22, we see a slightly more complex example of a `goalcheck()` function. This function needs to verify that all of a player's pieces have reached the goal tile, as opposed to just one. If this condition is met, that player is declared the winner.

On line 24, we use an integer flag to keep track of the current game state. This is the first time an integer variable is explicitly instantiated. The general form for integer declaration is: `int <variable name>;` (optionally, assignment can occur on this same line as demonstrated here). We begin with the assumption that the game has ended, then test for conditions that would prove otherwise. On lines 25 through 31, we see another example of the `for-each` loop. During each iteration of the loop, we test whether or not the current piece is occupying the goal tile. If this condition is ever false, the `gameOver` variable is set to 0 to indicate that the game has not yet ended. In order to determine if the current player's piece is on its goal tile, we refer to the index of the `FinishOn` array matching his player id. If the value stored there is equivalent to the occupied tile of the current piece, the condition is satisfied. Note that in this particular game, all players must reach the same tile in order to win (refer to line 4 of Program 4-1); therefore it is possible to replace the call to the `FinishOn` array with the number 10. After the `for-each` loop has terminated, the value of the `gameOver` variable is inspected. If still 1, the current player is declared the winner, and the game terminates.

Note that the board block is the only one that allows for more than one function to be defined (there are the `preRoll`, `goalCheck`, and `landsOn` functions). The order in which

you place the functions does not matter as long as all of the `landsOn` functions appear consecutively.

We now move on to the definition of the `Deck`; it is used here in place of a `Dice` block. In the programs seen so far, the movement of pieces was dictated by the value rolled on the dice. Alternatively, the programmer might want to use a deck of cards to dictate this movement. This can be done by defining a deck block. A `Deck` is a set of cards from which to draw on each player turn. If a `Deck` is defined, then a `Dice` block cannot also be defined. Each card in the deck has a value associated with it, which can be interpreted however the programmer desires. A difference between a deck and a set of dice is that when we used dice, the same `roll()` function had to be called no matter what value was rolled. With a deck, however, a different `roll()` function can be associated with each unique card. Also, a deck gives the programmer control over the quantity of each value in the deck, as opposed to a die where all you can specify is the number of faces (and each face has an equal probability of being selected). Lastly, the `Deck` block contains the only example in `ROLL` of a field that is not defaulted. The programmer is required to define at least one `Card` object if the `Deck` is defined. Not only does it not make sense that the deck would have no cards (meaning there is nothing to handle the flow of player turn control!), but additionally this default behavior is provided in the more basic `Dice` block. The programmer only writes a `Deck` if the `Dice` defaults are insufficient for his game. Let's now take a look at the `Deck` block for our sample program.

### Program 4- 3

```
1 Deck {
2     HasReplacement = 0;
3     make Card(value : 1, quantity: 3, roll:goAgain);
4     make Card(value : 2, quantity: 3, roll:directionRoll);
5     make Card(value : 3, quantity: 2, roll:goAgain);
6     make Card(value : 4, quantity: 2, roll:directionRoll);
7     make Card(value : 5, quantity: 1, roll:directionRoll);
8
9     function directionRoll = define roll(int amountRolled, int playerID) {
10
11         print("You drew a " | amountRolled);
12         print("Which piece would you like to move?");
13         int choice;
14         promptList(choice, {0~NumPieces-1});
15
16         print("In which direction would you like to move? 0-forwards, 1-backwards");
17         int direction;
18         promptList(direction, {0,1});
19         if(direction == 0)
20         {
21             print("Moving piece " | choice | " forward");
22             move(playerID, choice, amountRolled);
23         }
24         else
25         {
26             print("Moving piece " | choice | " backwards");
27             moveReverse(playerID, choice, amountRolled);
28         }
29     }
30
31     function goAgain = define roll(int amountRolled, int playerID) {
32         print("You drew a " | amountRolled);
33         print("Which piece would you like to move?");
34         int choice;
35         promptList(choice, {0~NumPieces-1});
36         move(playerID, choice, amountRolled);
37         print("GO AGAIN!");
38         NextTurn = playerID;
39     }
40
41 }
```

Line 1 contains the syntax for instantiating the deck. The first field that has to be set is `HasReplacement`, which must be either 0 or 1. If it is 0, then as cards are drawn from the deck, the deck gets smaller and smaller until none are left, at which time all cards are then shuffled and the process begins again. If it is 1, then after each card is drawn, it is placed back into the deck, and the deck is reshuffled. Lines 3 through 7 show how to create individual `Card`

objects in the deck. The first parameter for creating a `Card` is `value`, which can be any integer. The next is `quantity`, which indicates how many of that type of card will be in the deck. The last is `roll`, which allows the programmer to specify which `roll()` function he would like to be called when that card is drawn (which he will define below it).

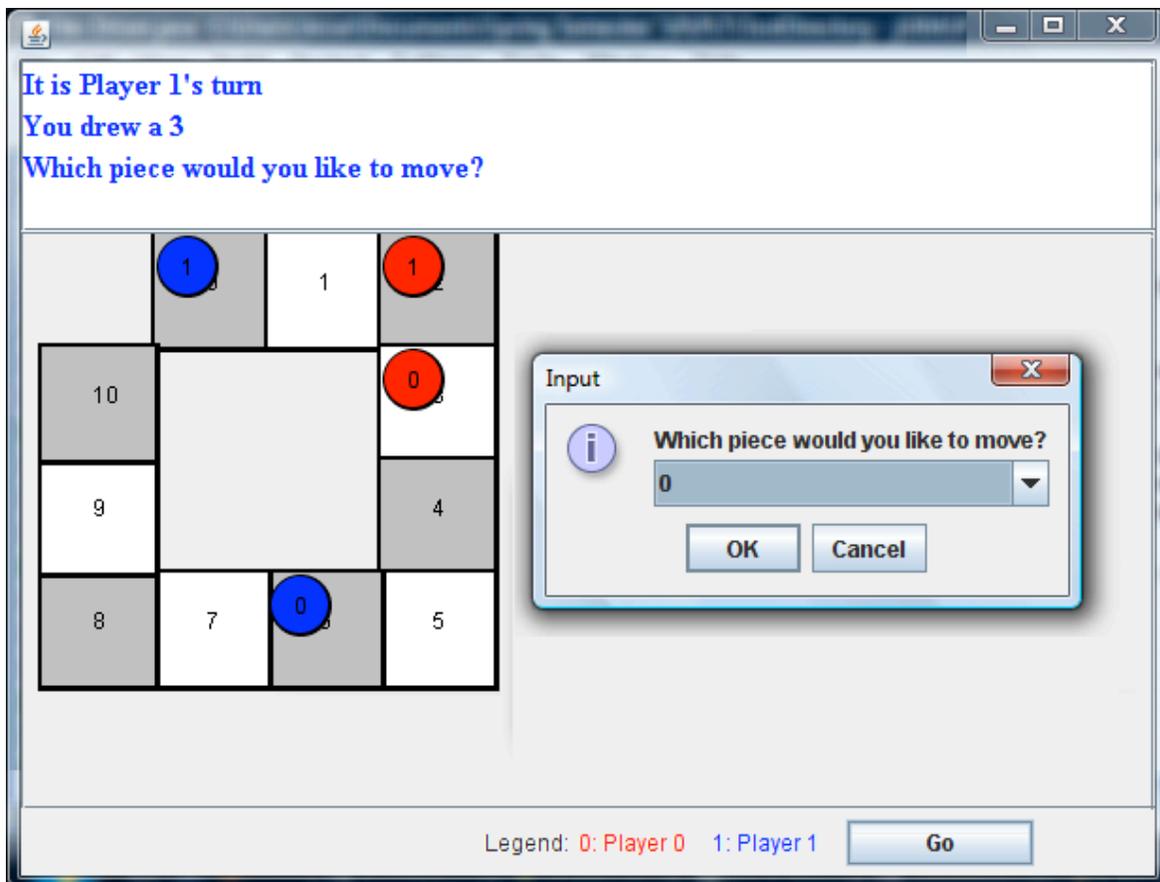
Another instance of a named function, `directionRoll`, is defined starting on line 9. Similar to `landsOn()`, a named `roll()` function allows the same definitions to be associated with several unique cards. Note that all `roll()` functions in the `Deck` block must be named functions (you may not write the unnamed one used in the `Dice` block). We now introduce the third and final input construction in `ROLL`, `promptList()`. Similar to `promptRange()`, it accepts an integer variable as its first argument whose value will be set with the input. Unlike `promptRange()`, `promptList()` has only two arguments, the second of which is an array of integer values. Each value of the array is printed out for the user to choose amongst. On line 14, when calling `promptList()`, we store the user input into a variable that the programmer creates himself, an integer variable called `choice`. In line 17 is the declaration of another integer variable called `direction`, which is used to store the direction in which the player wants to move. We see the first call of `moveReverse()` on line 27, which is just like `move()` except, as the name implies, the piece will move backwards instead of forwards. If the user chooses to go forward, however, the normal `move()` function is called (line 22).

Additionally, there is a second `roll()` function declared, called `goAgain`, in lines 31-39. This shows how you may change the `NextTurn` field (line 38) to manipulate the order of player moves. The value it is set to become the id of the player whose turn will be next. In this

game, a draw of card 3 rewards the player with a second consecutive turn; thus the value of `NextTurn` is reset to the value of the current player.

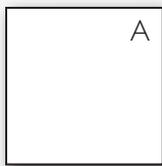
As with program 3, the three blocks of program 4 may be placed, in order, inside of a named `Game` block to construct a working program. To compile this program using the `ROLL` compiler, place your program file into the same directory that houses all of the `ROLL` compiler files. From this directory you may use the provided shell script `rollc` to compile your program by typing `rollc` followed by a space and then the filename. Then to execute it, simply type `roll`. Here is a screen shot of what program 3 might look like if you ran it:

Figure 3



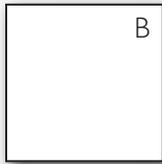
## **6. Conclusion**

At this point we have covered the essential structures and syntax of the ROLL programming language. Using these basic tools, complex and powerful board games may be created. For more detailed descriptions and examples of all the constructs in ROLL, please refer to the language reference manual. Game on!



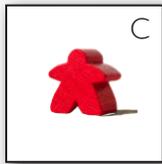
A

Introduction



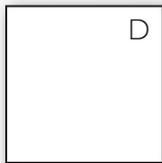
B

Language Tutorial



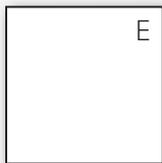
C

Primer on Organizing  
ROLL Code



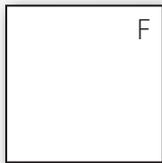
D

Language Reference  
Manual



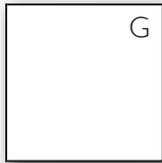
E

Project Plan



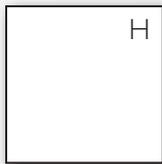
F

Language Evolution



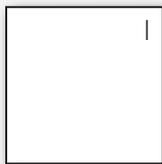
G

Translator  
Architecture



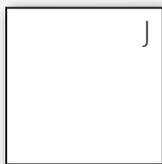
H

Development  
Environment



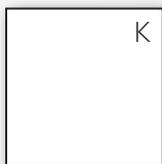
I

Test Plan



J

Conclusions



K

Appendix



# Primer on Organizing ROLL Code



## Primer on Organizing ROLL Code

The following pages contain diagrams that illustrate how the code in a ROLL program is organized.

This chapter is meant to be a high-level overview of how a ROLL program is structured. Please refer to the page numbers throughout the glossary below and tree diagrams to find the page in the Language Reference Manual where each is discussed in more detail.

### Glossary of terms used in the ROLL language and in the diagrams

#### 1. *Blocks*

These are the four major sections of the ROLL program. These include the `Players`, `Board`, `Dice`, and `Deck` blocks. They represent the things you would find in a board game box you buy at the store.

See page D - 13 of the Language Reference Manual for details about *blocks*.

#### 2. *Block-level components*

The lines of code you see in a block fall into one of three statement categories: *block-level fields*, *object data types*, and *functions*. We will go into more detail of each category in the following glossary items.

Note that you have to keep your statement categories written in the same order we just mentioned. You should initialize your *fields*, make your *objects*, and define your *functions* in that order when you write a *block*.

#### 3. *Block-level fields*

These *fields* are the properties of the *block* they belong to. They are variables that are defined by the ROLL language itself because they are important to the functioning of the *block*.

You can find *block-level fields* in three flavors. They are always one of the following: integers (most fields are of this category), integer arrays (the only ones are `StartOn` and `FinishOn` in the `Players` *block*), or arrays of *objects*, which we will discuss in the next glossary item.

See page D - 14 of the Language Reference Manual for details about *fields*.

#### 4. *Object data types*

These are items of a board game that you would find as part of a *block*. The *object data types* in ROLL are `Player`, `Tile`, `Die`, and `Card`. These are *objects* that exist as multiple copies in any board game session.

Each *object* is in charge of holding information about itself. This information, which we called *object attributes*, will be discussed in the next glossary item.

With the exception of `Player`, the ROLL programmer creates objects using the `make` command.

See page D - 18 of the Language Reference Manual for details about *objects*.

#### 5. *Object attributes*

This is the information that is stored in the *object*.

You can find *object attributes* in three flavors. They are always one of the following: integers (most attributes are of this category), names of functions (so you can attach a named *function* to the object), or, in one case, an array of more objects (we're talking about `PieceList`, which belongs to `Player`).

See page D - 18 of the Language Reference Manual for details about *attributes*.

## 6. *Functions (Dynamic Functions)*

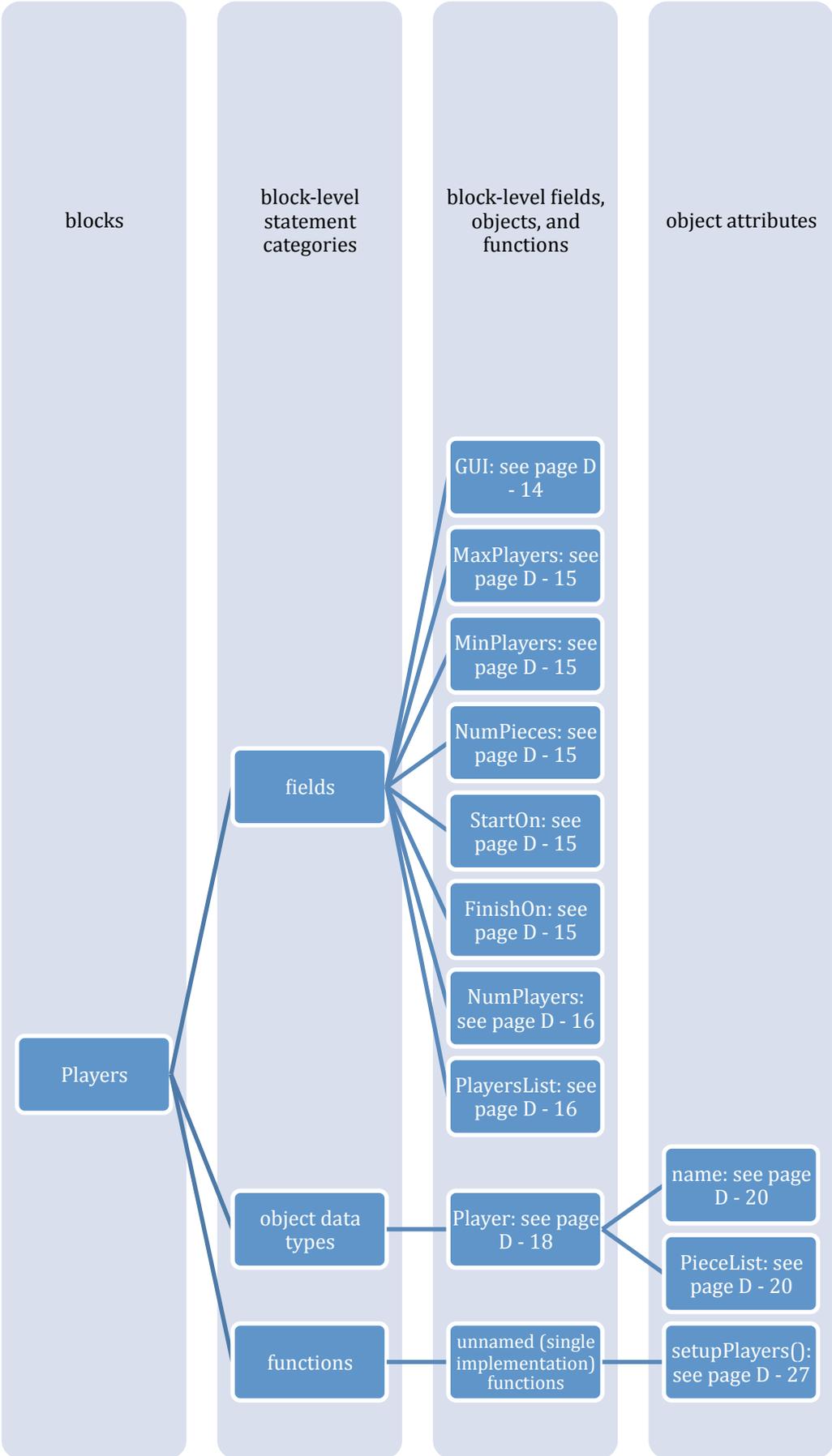
These are the methods in the ROLL language. *Functions* are the last of the three statement categories that go into writing a *block*. There are two different kinds of functions in ROLL, unnamed and named functions.

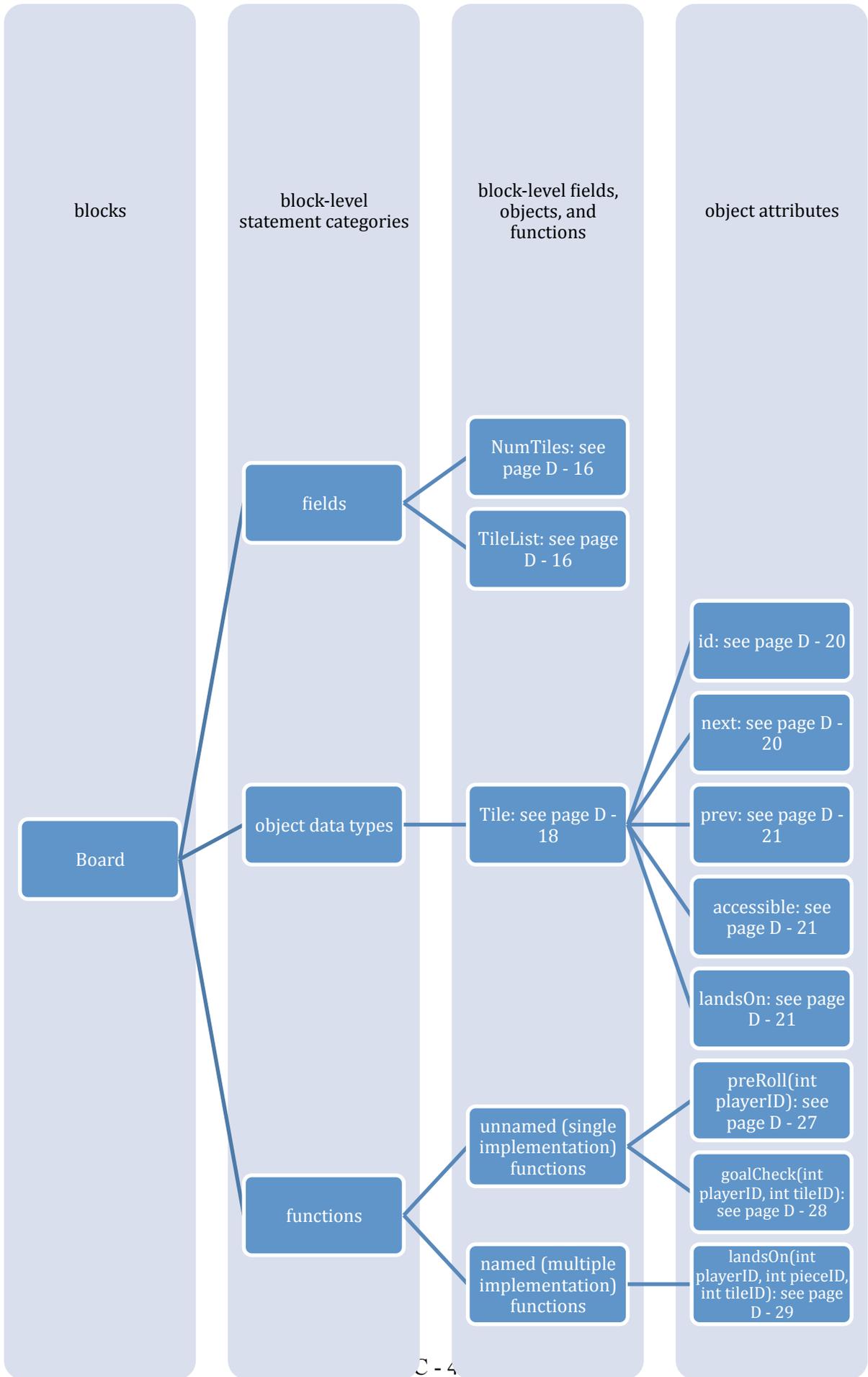
See page D - 25 of the Language Reference Manual for details about *functions*, and to learn about the difference between unnamed and named functions.

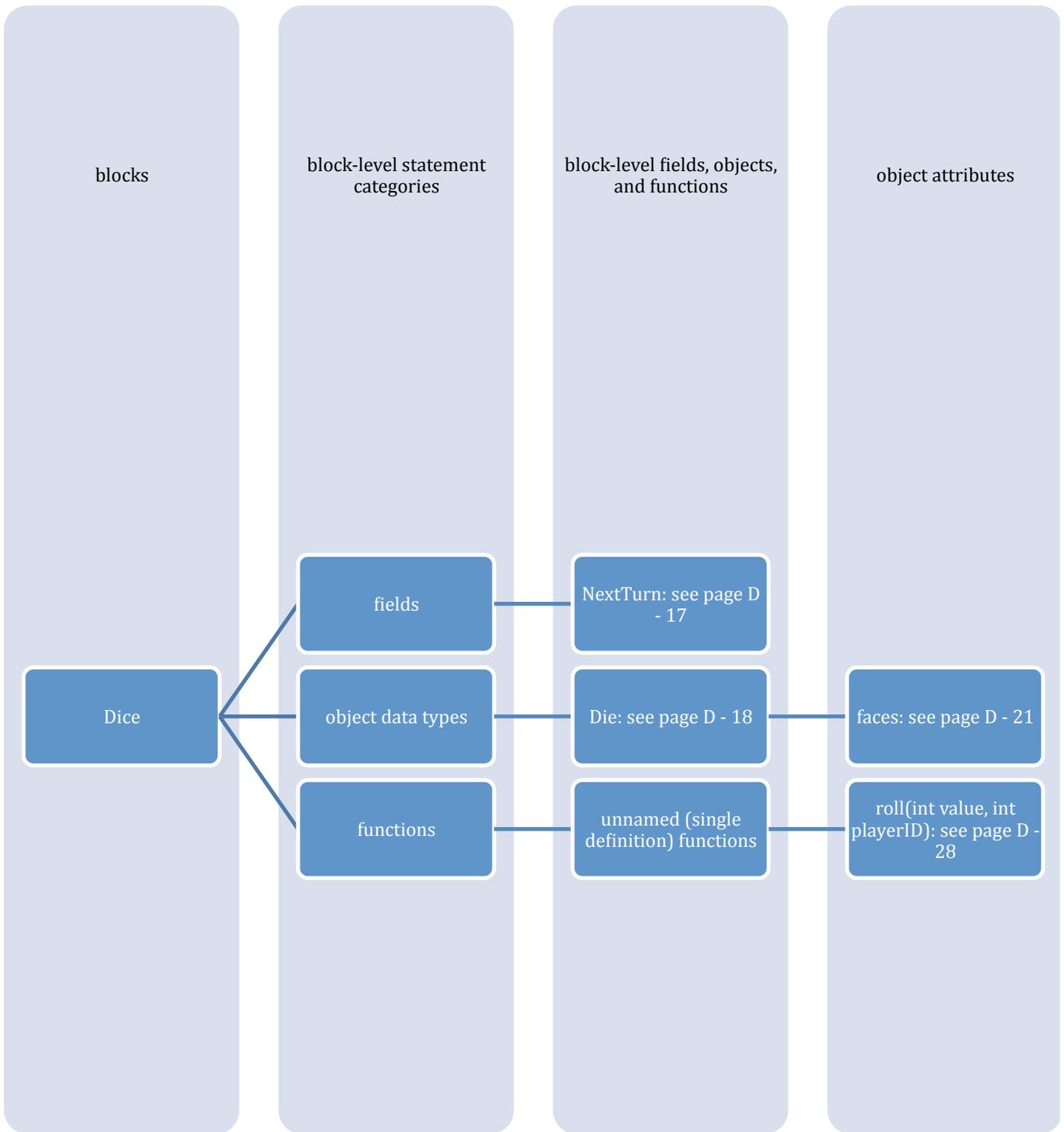
### **How to organize your ROLL program**

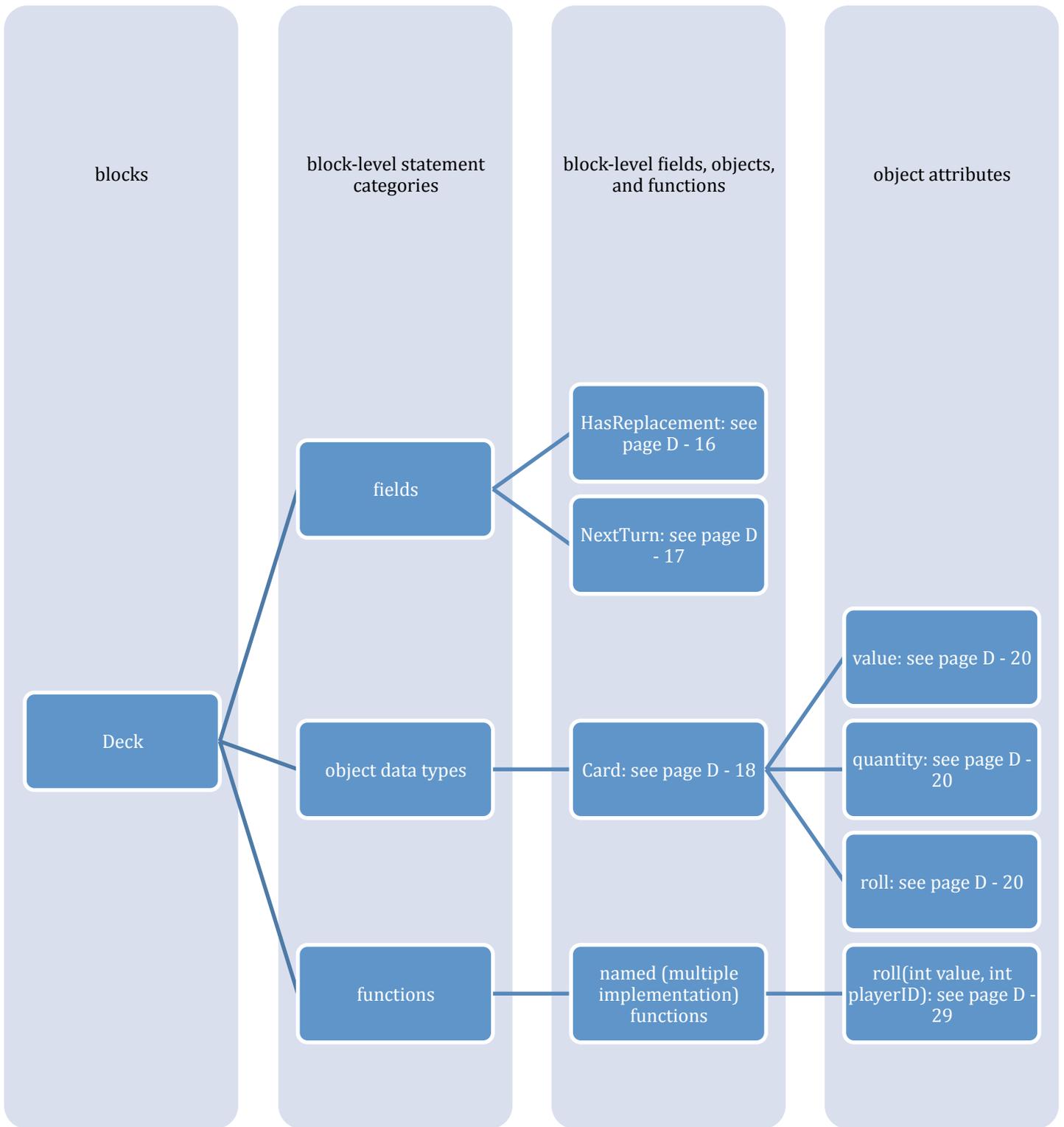
The elements of a ROLL program have to be written in the order of a top-to-bottom traversal of the trees on the next four pages. In other words, if an element A comes on top of element B in tree diagrams, element A should come before element B in your ROLL source code.

If you choose to leave out any element in your ROLL program, the elements you do choose to implement have to come in the order shown in the diagrams.

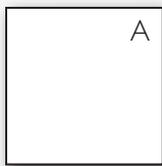






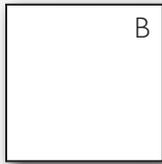






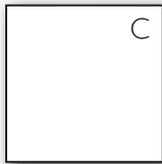
A

Introduction



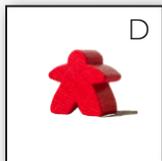
B

Language Tutorial



C

Primer on Organizing  
ROLL Code



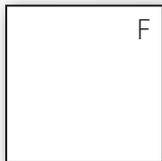
D

Language Reference  
Manual



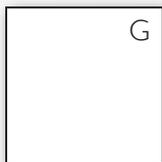
E

Project Plan



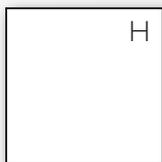
F

Language Evolution



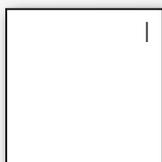
G

Translator  
Architecture



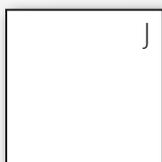
H

Development  
Environment



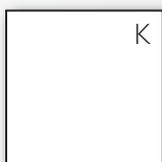
I

Test Plan



J

Conclusions



K

Appendix

# Language Reference Manual



# ROLL Language Reference Manual

Jesse Bentert - jrb2137  
John Graham - jwg2116  
Yipeng Huang - yh2315  
Lauren Pully - lep2128  
Dan Wilkey - dgw2109

1. Introduction.....	3
2. Syntax .....	3
a. Line Termination.....	3
b. Whitespace.....	3
c. Comments .....	4
3. Primitive Types.....	4
a. Introduction .....	4
b. Integers.....	4
c. Text .....	4
d. Arrays.....	5
e. Casting .....	6
4. User-Initialized Variables .....	6
a. Overview .....	6
b. Local Integer Variables.....	7
c. Local Integer Arrays .....	7
d. Global Integer Variables .....	8
5. Operators & Logic .....	8
a. Arithmetic Operators.....	8
b. Logical Comparisons .....	9
c. Concatenation Operator .....	10
d. Conditionals .....	10
e. For-each Loops.....	12
6. How a ROLL program is organized.....	13
7. Block-Level Fields.....	14
a. Introduction .....	14
b. Scope.....	14
c. Field List .....	14
8. Object Data Types.....	17
a. Introduction .....	17
b. Object Data Type List.....	18
9. Attributes of Object Data Types .....	18
a. Specifying Attributes .....	18
b. Accessing Attributes .....	18
c. Attribute List .....	19
10. Functions.....	22
a. Introduction .....	22
b. Functions Already Defined by the ROLL Language (Static Functions) .....	22

i. Static Function List.....	22
c. Function Templates That Can Be Defined By The Programmer (Dynamic Functions) .....	25
i. Single Definition vs. Multiple Definition Functions .....	26
ii. Function Naming Mechanism.....	26
iii. Argument Enumeration .....	26
iv. Dynamic Function List (Single Definition).....	27
v. Dynamic Function List (Multiple Definition).....	29
11. Reserved Words.....	31

## 1. Introduction

ROLL is an easy-to-use programming language designed to facilitate the implementation of children's board games. Aspiring ROLL developers should have some programming experience with a high-level language like Java or C#. A creative and unique idea for a game will also come in handy, though not entirely necessary. Well-implemented games in ROLL contain a custom game board, multiple players, an element of chance, and choices available to the players that influence the outcome of the game. A program written in ROLL could serve as the back-end of a complete game program that may include a graphical user interface implemented in another language.

The following Language Reference Manual serves to introduce aspiring roll programmers both to the syntax of the language and details on how to most effectively program in ROLL.

## 2. Syntax

Elements of our language will be formatted according to the following convention:

Element Name	Format	Example
Functions	Lowercase followed by camel case	aMethod
Local Variables	Lowercase	next
Block-level Fields	Uppercase followed by camel case	StartOn
User Defined Global Variables	Dollar sign, \$, followed by uppercase, followed by camel case	\$MyGlobal
Object Types	Uppercase followed by camel case	TileList
Language Keywords	Lowercase	make

### a. Line Termination

All lines are terminated with semicolons (;)  
Methods and type definitions are an exception – their start and end is delimited with curly braces ({}), so they are not followed by semicolons.

### b. Whitespace

Whitespace has no meaning in our language other than to increase code readability.

Example:

```
Game aGame {  
  Players {}  
  Board {}  
  Dice {} // or Deck {}  
}
```

Is equivalent to:

```
Game aGame { Players{} Board{} Dice{} }
```

### c. Comments

Comments in the source code are ignored by the compiler and are a useful tool in writing readable code. Only line commenting is allowed in ROLL. Line comments begin with two successive forward slashes and continue through to the end of the current line (without exception). The following is an example:

```
int pizza; //this is a comment
```

## 3. Primitive Types

### a. Introduction

In an effort to minimize the number of data types a ROLL programmer must be familiar with, ROLL uses a small set of primitive data types including integers and text.

### b. Integers

ROLL accepts any signed integer. They can range in value from -2,147,483,648 to +2,147,483,647. Positive integers do not need to be indicated with a (+) sign.

Example:

```
183  
-9238484  
9
```

### c. Text

Text is defined as any combination of all printable ASCII characters placed in between a set of double quotes.

Within text, the ASCII character ‘”’ and the sequence of two “//” are not allowed. These characters are reserved for other constructs in the ROLL language.

Example:

```
"Enter a name:"  
"Adam"
```

Note that programmer may not define their own text variables. They can only be used in a call to `print()` (see functions) or as a player’s name attribute (see object attributes).

#### d. Arrays

ROLL uses arrays of various types of objects in the implementation of a game. `TileList`, `PlayerList`, and a player’s `PieceList` are, respectively, arrays of tiles on the board, players in the game, and pieces of the players. The programmer cannot create new arrays of these three types from scratch.

Arrays of integers are also available in ROLL. Some arrays (like `StartOn`) are already specified in the language and need only be set by the programmer. Arrays fields can be created using the following syntax:

```
StartOn = {1, 2, 3, 4};
```

Here, `StartOn` is assigned an integer array of length four, storing values 1, 2, 3, and 4.

Arrays can also be set using an integer variable:

```
int aValue = 7;  
StartOn = {1, 2, 3, aValue};
```

Alternatively, arrays can be created using the following shorthand syntax, placing a tilde between the upper and lower bounds of the array. The array will be filled with all integers between the upper and lower bounds. Integers, variables, or complex arithmetic expressions involving both integers and variables can be on both sides of the tilde for shorthand array declaration. The following array will be instantiated with all values between its upper and lower bounds:

```
StartOn = {1 ~ 4};
```

Here, `StartOn` also is assigned an integer array of length four storing values 1, 2, 3, and 4.

Note that in using the special tilde syntax, the upper bound must be greater than the lower bound. `ROLL` accepts arrays of integers up to size 2,000,001.

The user may also define custom arrays. See user-initialized variables, below.

### e. Casting

`ROLL` does not allow the programmer to deliberately cast one type to another. Casting only occurs from integers to text when an integer is concatenated with text. The concatenation operator, `|`, will be discussed in a later section.

Example:

```
print("It is player" | playerID | "'s  
turn.");
```

Output:

```
It is player3's turn.
```

The integer `playerID` is cast to a text representation of its numerical value.

## 4. User-Initialized Variables

### a. Overview

Programmers using `ROLL` may instantiate three kinds of variables: local integers, local integer arrays, and globally visible integer variables.

Naming Convention

Variable names are case sensitive in `ROLL`. A variable name can be any combination of upper or lowercase characters, numerical digits (0 through 9), and underscore (`_`) characters. By convention, the fields of the `ROLL` language that the programmer may set are camel case, starting with a capital. Variables that the programmer defines himself are camel case, starting with a lowercase letter.

Instantiation & Assigning an Initial Value

To instantiate a variable the programmer uses the “int” keyword. To assign a value to a variable a programmer uses the '=' assignment operator.

Example:

```
int myInteger = 15;
```

## b. Local Integer Variables

By convention, local integer names begin with a lowercase character followed by camel case.

The scope of a variable that the programmer defines himself is limited to the function level, block level, or game level where the variable is instantiated.

To reassign a value to the local variable, use the '=' assignment operator.

Example:

```
int myInteger = 15;
myInteger = anotherInteger;
```

## c. Local Integer Arrays

The names of local integer arrays must be preceded by a '#'. By convention, local integer array names begin with a lowercase character followed by camel case.

The scope of a variable is limited to the function level, block level, or game level where the array is instantiated.

There are three ways to create an integer array:

```
int #myArray[size]; //size is the size of the
array
int #myArray = {1, 2, 3, 4}; //in-place array
instantiation
int #myArray = {1 ~ 4};
```

To reassign a value to an array element, use the '=' assignment operator.

Examples:

```
int #myArray[4];
```

```
int #myArray = {5, 6, 7, 8};
#myArray[2] = 7;
```

#### d. Global Integer Variables

Integer variables that are not specified in any of the four main blocks are part of the Game level. All fields that are declared at this level can be modified by any of the four main blocks. Global fields must be declared before any of the blocks are implemented.

They can be useful to games that require integer variables to be visible in all areas of the game.

The names of global integer variables must be preceded by a '\$'. By convention, global integer variable names begin with an uppercase character followed by camel case.

Examples:

```
int $MyGlobalVariable = 7000;
$MyGlobalVariable = $MyGlobalVariable + 1000;
```

Note that only initialization of a global variable is done before the blocks are implemented. Expressions that use these global variables, such as in the second line of the example, must appear *within* the blocks.

## 5. Operators & Logic

### a. Arithmetic Operators

Arithmetic operations can only be used on integers. They can be used to assign a value to a new integer or to an existing integer. The four basic operations are:

Operator	Name	Description
+	Addition	Adds the left operand to the right operand.
-	Subtraction	Subtracts the right operand from the left operand (Also used to indicate a negative number)
*	Multiplication	Multiplies the left operand by the right operand.

Operator	Name	Description
/	Division	Divides the left operand by the right operand. (Truncates non-zero decimals in the quotient)

ROLL accepts arithmetic expressions in infix notation. The standard order of operations is preserved. Expressions are evaluated from left to right with multiplication and division computed before addition and subtraction. To eliminate ambiguity when *reading* ROLL code, you may surround arithmetic computations by parentheses.

Precedence Specification	Name	Description
( )	Parentheses	Operations contained within parentheses will be computed first, regardless of the order of precedence. Parentheses must contain a left operand, followed by an operator, followed by a right operand.

Example:

```
int x = 0;
int y = 11;
x = y + 2;           // x is now equal to 13
y = x / 2;          // y is now equal to 6
int z = 100 - x * y; // z is now equal to
22
```

Example:

The following lines of code are *equivalent*:

```
int z = 100 - x * y;
int z = 100 - (x * y);
int z = -(x * y) + 100;
```

But they are *not* equivalent to:

```
int z = (100 - x) * y;
```

## b. Logical Comparisons

Logical comparison operators are used to compare integers by their value. There are several logical operators in ROLL:

<b>Conditional Operator</b>	<b>Description</b>
==	equals
!=	not equals
!	not*
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

Examples of logical comparisons between integers will be shown in a later section, in conjunction with the conditional operators that are used to compute the return values of logical comparisons.

\*note that the ! (not) operator can only be used on complete *expressions* that are inside parentheses. Expressions must contain an operator between 2 values.

### c. Concatenation Operator

Text fields and integers can be appended to each other using the | operator.

<b>Operator</b>	<b>Name</b>	<b>Description</b>
	Concatenation	Appends the right operand to the end of the left operand. The right and left operands can be any combination of integers, texts, and variables.

### d. Conditionals

These represent blocks of code that will only be computed if a certain expression evaluates to true. The best way to learn these is by example.

#### If Conditionals

Evaluate a given condition. If the condition evaluates to true (non-zero), it will execute the block of code immediately following the condition.

```
if (condition) {
    //execute this block of code
}
```

#### Else If Conditionals

Evaluate the conditions sequentially. Upon evaluating the first TRUE condition, it will execute that condition's preceding block of code and discontinue evaluating the following conditions.

```
if (condition_1) {
    //execute condition_1's block of code
}
else if (condition_2) {
    //execute condition_2's block of code
}
else if (condition_3) {
    //execute condition_3's block of code
}
```

#### Else Conditionals

This is the final case in a group of if and else if statements. If all the preceding conditions are false, then the else condition is reached and its following block of code is executed.

```
if (condition_1) {
    //execute condition_1's block of code
}
else if (condition_2) {
    //execute condition_2's block of code
}
else if (condition_3) {
    //execute condition_3's block of code
}
else {
    //execute this block of code if all
    // previous conditions were false
}
```

#### Examples:

Given the following integers, conditional statements are evaluated below.

```
int x = 5;
int y = 12;
int z = 16;
```

#### Example 1:

```
if (x < y) {
    // this block will be executed
}
```

Example 2:

```
if (y >= 20) {
    // this block will be ignored
}
else if (z == x) {
    // this block will be ignored
}
else if ( !(x != 5) ) {
    // this block will be executed
}
else if (y != z) {
    // this block will be ignored
}
```

Example 3:

```
if (z <= y) {
    // this block will be ignored
}
else if (x == y) {
    // this block will be ignored
}
else {
    // this block will be executed
}
```

## e. For-each Loops

These are used to iterate through an array and perform an action on every value in that array. For-each loops can only iterate through an array of integers (you may not iterate through the `PlayerList`, `TileList`, or a player's `PieceList`). For-each loops may *not* add values into or remove values from the array.

Examples:

The following examples print each value in the array. You can use any of the three types of arrays in a for-each loop. The three examples below all print the same output:

Output:  
the next element is: 5

```
the next element is: 6
the next element is: 7
the next element is: 8
the next element is: 9
```

```
1.
for ( int i : { 5, 6, 7, 8, 9 } ) {
    //print("the next element is: " | i );
}

2.
for ( int i : { 5 ~ 9 } ) {
    print("the next element is: " | i );
}

3.
int #myArray = {5 ~ 9};
for ( int i : #myArray ) {
    print("the next element is: " | i );
}
```

Furthermore, the integer that is the index used for the iteration can only be instantiated as part of the for-each statement.

Example:

```
ROLL programmers may write for (int i : {1 ~ 5})
{...}, but they may not write int i = 0; for (i : {1 ~
5}) {...}.
```

## 6. How a ROLL program is organized

A program written in ROLL is organized into three major blocks: the `Players` block, the `Board` block, and a choice of either the `Dice` block or the `Deck` block. Each block contains block-level fields, object data types, and functions that are closely associated with the functioning of that block. These three statement types must appear in this order in a block (fields, objects, functions). Descriptions of these are in the following sections.

The major blocks must be arranged, in the case of a program using the `Dice` block, in the order of `Players`, `Board`, and then `Dice`. For programs that use the `Deck` block, they should be in the order of `Players`, `Board`, then `Deck`. Any block can be omitted, in which case the language default is used for that block.

## 7. Block-Level Fields

### a. Introduction

Within ROLL's main blocks are fields that can be set by the programmer. With the exception of one called `NextTurn`, they all can be set only once. They all may be omitted if the programmer wishes to use a default value that we provide. This section enumerates all the programmable block-level fields in ROLL, the block in which they can be set (parent block), the block(s) in which they can be accessed, and the values they default to if they are omitted by the ROLL programmer.

### b. Scope

All block-level fields follow the same rule with regard to the textual regions of the program in which they may be accessed. All block-level fields may be accessed within dynamic functions that belong to the same block *or a later block*. This means that all of the fields of the `Players` block are accessible in the `Players`, `Board`, and `Dice/Deck` blocks. All of the fields of the `Board` block are accessible in the `Board` and `Dice/Deck` blocks. All of the fields of the `Dice/Deck` are accessible only in the `Dice/Deck` block.

Note that the fields must be set in the order in which they appear below (although as stated, any can be omitted). Unless otherwise specified, the field can only be set explicitly by the user and may not be used in a call to `promptList` or `promptRange` (see functions), which allow the user to set a variable.

### c. Field List

Field	Parent Block	Description	Default Value
<b>GUI</b>	<code>Players</code>	An integer flag, specified in the <code>Players</code> block, that determines if the output of a program should be displayed in a GUI (set value to 1) or to the command line (set value to 0).	0

<b>MaxPlayers</b>	Players	An integer representing the maximum number of players in the game. Accessible by all blocks.	6
<b>MinPlayers</b>	Players	An integer representing the minimum number of players in the game. Accessible by all blocks.	2
<b>NumPieces</b>	Players	An integer representing the number of Pieces that each player will have in the game. Accessible by all blocks.	1
<b>StartOn</b>	Players	An integer array that indicates the starting tile's ID of each players' pieces. Accessible by all blocks.  If the programmer specifies a <code>StartOn</code> array that has fewer values than there are players, then for the players with an ID number greater than the number of specified values, their pieces will start by default on the 0 <sup>th</sup> tile.	{0, 0, 0, 0, 0, 0}
<b>FinishOn</b>	Players	An array that indicates the victory condition tile ID of each player's pieces. Accessible by all blocks. This field can be set by prompting the user (see <code>promptRange</code> and <code>promptList</code> functions)  If the programmer specifies a <code>FinishOn</code> array that has fewer values than there are players, then for the players with an ID number greater than the number of specified values, their pieces will have a <code>FinishOn</code> value of 9.	{9, 9, 9, 9, 9, 9}
<b>NumPlayers</b>	Players	An integer representing the number of players that will play the game. This field can be set by prompting the user (see <code>promptRange</code> and <code>promptList</code> functions)	2

<b>PlayerList</b>	Players	An array of all the Players. Accessible by all blocks. You can access an individual player object (discussed later) by providing an integer in square brackets after <code>PlayerList</code> . Eg: <code>PlayerList[3]</code> will return the player whose ID is 3. <b>Note that this cannot be explicitly set to any values; it is automatically generated behind the scenes.</b>	N.A.
<b>NumTiles</b>	Board	The number of Tiles on the Board. Accessible by the Board and Dice/Deck blocks.	10
<b>TileList</b>	Board	An array of all the tiles. Accessible by the Board and Dice/Deck blocks. You can access an individual tile object (discussed later) by providing an integer in square brackets after <code>TileList</code> . Eg: <code>TileList[10]</code> will return the tile whose ID is 10.  <b>Note that this is not set to any values the way the previous fields are. Each tile is created using the special “make” command (see section 9a)</b>	N.A.
<b>HasReplacement</b>	Deck	An integer value that indicates whether a deck has replacement or not. More specifically: set to 0 if cards have an equal probability of being drawn on every roll and set to 1 if all cards in the deck must be selected before any repeats may occur. Once all cards have been drawn in the second scenario, the deck is randomized or ‘shuffled’ to ensure a new ordering the next time through. This field is accessible only by the Deck block.	0

<b>NextTurn</b>	Dice/Deck	<p>An integer that the programmer can set to force the game to give the next turn to a specific player. Usually written as part of a <code>roll()</code> function definition.</p> <p>This field is accessible by just the Dice/Deck block.</p> <p>Note that unlike the previous block-level fields, <code>NextTurn</code> can be set to a value more than once.</p> <p>This field can be set by prompting the user (see <code>promptRange</code> and <code>promptList</code> functions)</p>	<p>Normally, the ID of the player who goes next is one plus the ID of the player who is currently going (until you get to the player with ID <code>NumPlayers - 1</code>; then the next player is he who has ID 0).</p>
-----------------	-----------	---	---

## 8. Object Data Types

### a. Introduction

The ROLL language has five object data types: `Tile`, `Die`, `Card`, `Player`, and `Piece`. The first four of these are each associated with one of the four main blocks. `Piece` is associated with a `Player` object data type. With the exception of `Player` and `Piece`, these data types can be instantiated by the user using the `make` command with the following syntax:

```
make Object(attribute1:value1,
attribute2:value2);
```

For example, a `Card` might be created by typing the following:

```
make Card(value:6, quantity:4);
```

Each object data type has associated attributes, which is discussed in the following section.

## b. Object Data Type List

Object Data Type	Parent	Attributes	Example Instantiation
<b>Player</b>	Players Block	PieceList, Name, occupiedTileID	Players (and their pieces) in ROLL are instantiated automatically and cannot be created explicitly.
<b>Tile</b>	Board Block	ID, next, prev, accessible, landsOn	make Tile(id:4, next:5, prev:3, accessible:{1, 7}, landsOn:jumpToStart);
<b>Die</b>	Dice Block	faces	make Die(faces:6);
<b>Card</b>	Deck Block	value, quantity	make Card(value:4, quantity:6, roll:splitRoll);
<b>Piece</b>	Player Object	occupiedTileID	Pieces in ROLL are instantiated automatically and cannot be created explicitly.

## 9. Attributes of Object Data Types

### a. Specifying Attributes

Each object data type has the attributes listed in the table above. A list of these attributes in more detail appears in the table that follows. For the object data types that are created using the make command (Tile, Die, and Card) the attributes must be set in the order in which they appear in the following table. For example, when creating a Tile, you can write

```
make Tile(id:4, next:5, prev:3);
```

but you *cannot* write

```
make Tile(next:5, id:4, prev:3);
```

Some of these attributes may be omitted when using the make command; this is specified in the “can omit?” column.

For the Player object data type, which does not use the make command, the only attribute that can be set by the programmer is name. This must be set using the promptName function, which is discussed later.

No attributes of the Piece object data type may be set by the programmer.

### b. Accessing Attributes

Many of these attributes serve a purpose in and of themselves (eg: the number of faces on the die determines the amount rolled on each turn), but

other attributes only make sense if they can be accessed by the programmer (eg: the `accessible` array of a `Tile`). It is possible in ROLL to access some of the attributes of `Player` data types, `Tile` data types, and `Piece` data types. However, none of the attributes of the `Die` and `Card` data types may be accessed.

To access the attributes of an object data type, you must be able to access the object data type itself; you can then refer to its attribute by appending `.attributeName` to its end. Thus, to access the attributes of a `Player` or a `Tile`, you must be able to access a `Player` data type itself and a `Tile` data type itself. These can be accessed from the `PlayerList` field of the `Players` block and from the `TileList` field of the `Board` block, which are arrays of `Player` objects and `Tile` objects, respectively. To specify an individual object, place the ID of that object in square brackets after the list name.

For example, `PlayerList[3]` returns the player object whose ID is 3, and `PlayerList[3].name` returns his name attribute.

To access an attribute of a `Piece`, you must first obtain the `PieceList` of a player. Since the `PieceList` is actually an attribute of a `Player` object data type, you can refer to the `PieceList` by writing `PlayerList[playerID].PieceList`, and to a piece's attribute as: `PlayerList[playerID].PieceList[pieceID].attributeName`.

Since no attributes of the `Die` or `Card` data type can be accessed, you cannot access a `Die` or `Card` data type (and hence, there is no such thing as a *DieList* or a *CardList*).

Here is a list of object attributes and their associated properties. The “can be accessed” column specifies whether the attribute is accessible by the programmer.

### c. Attribute List

Attribute	Parent Object Data Type	Description	Can omit?	Default value if it can be omitted	Can be accessed?

<b>name</b>	Player	A text field of the Player that indicates the player's name.	N.A. (a <b>Player</b> is not created using make command)	N.A.	Yes
<b>PieceList</b>	Player	An array of the pieces that a given Player has. You can get an individual piece by providing an integer in square brackets after <b>PieceList</b> .  Eg: <code>PlayerList[4].PieceList[2]</code> will return the third piece object (ID 2) of the player data type whose ID is 4.	N.A. (a <b>Player</b> is not created using make command)	N.A.	Yes
<b>occupiedTileID</b>	Piece	A field of a piece inside the <b>PieceList</b> of a Player. This local variable can be accessed in the following manner:  <code>PlayerList[int_type].PieceList[int_type].occupiedTileID</code>	N.A. (a <b>Piece</b> is not created using make command)	N.A.	Yes
<b>id</b>	Tile	A local field of Tile. The ID number of a given tile. These are successive integers from zero to the number of tiles - 1.	No	N.A.	No
<b>next</b>	Tile	A local field of Tile. The ID number of the next tile that a Piece will move forward to when traversing the tiles sequentially in a <code>move()</code> call (function calls are discussed later).	No	N.A.	Yes

<b>prev</b>	Tile	A local field of Tile. The ID number of the previous tile that a piece will move backwards to when traversing the Tiles sequentially in a call to <code>moveReverse()</code> .	No	N.A.	Yes
<b>accessible</b>	Tile	A local field of Tile. An array of IDs of Tiles that a Piece can move to from a given Tile.	Yes	An empty array	Yes
<b>landsOn</b>	Tile	A local field of Tile. A field within tile that indicates which <code>landsOn()</code> function to call when a Piece lands on that Tile.	Yes	The default <code>landsOn</code> function	No
<b>faces</b>	Die	An integer field within Die that specifies how many faces are on the given Die. When a player rolls the die, it can roll any value from 1 to the number of faces specified in this value.	No	N.A.	No
<b>value</b>	Card	A local field of Card. The number on the Card and the default number of tiles a piece will move forward when a Player draws that Card.	No	N.A.	No
<b>quantity</b>	Card	A local field of Card. Indicates the number of one type of a Card found in the Deck.	No	N.A.	No
<b>roll</b>	Card	A local field of Card. A field within Card that indicates which <code>roll()</code> function to call when that Card is drawn. The <code>roll()</code> function then executes the player's turn.	Yes	The default <code>roll()</code> function	No

## 10. Functions

### a. Introduction

Functions are the methods in ROLL. They are groupings of instructions that are common to many board games. Functions can have a name and a set of arguments that are passed into the function.

### b. Functions Already Defined by the ROLL Language (Static Functions)

Many functions in ROLL are already defined and only need to be called. We call these static functions. They are called simply by passing the necessary function arguments to the function. These functions are `promptText()`, `promptList()`, `promptRange()`, `print()`, `declareWinner()`, `move()`, `moveReverse()`, and `jump()`. Note that the programmer is not allowed to define these functions himself.

#### i. Static Function List

##### **print(text textToPrint)**

**Called in:**

Can be called anywhere in the program.

**Arguments:**

`text textToPrint` — the text that will be printed to the console.

**Action:**

The program prints out the `text` passed in through the function argument.

##### **promptName(text nameToBeChanged)**

**Called in:**

Can be called anywhere in the program, but is usually found as a function call in the `setupPlayers()` functions.

**Arguments:**

`text nameToBeChanged` — the text variable that a player's name attribute will be set to.

**Action:**

The program waits for the next line entered by the player, terminated by a carriage return, and stores the entered text in the name field passed in by the programmer.

### **promptList(int intToBeChanged, int[] options)**

#### **Called in:**

Can be called anywhere in the program, but is usually found as a function call in the `setupPlayers()` and `roll()` functions.

#### **Arguments:**

`int intToBeChanged` — the `int` variable that will store the user's selection

`int[] options` — the array of integers that are valid options.

#### **Action:**

The program prints out the list of integers enumerated in options. The program then waits for the next line entered by the player, terminated by a carriage return, and determines if the entry is an integer in the list of allowed integers. If it is, the program stores the integer in the variable passed in by the programmer (`intToBeChanged`). Otherwise, the program waits until a valid integer is entered.

### **promptRange(int intToBeChanged, int lowerBound, int upperBound)**

#### **Called in:**

Can be called anywhere in the program, but is usually found as a function call in the `setupPlayers()` and `roll()` functions.

#### **Arguments:**

`int intToBeChanged` — the `int` variable that will store the user's selection

`int lowerBound` — the lower bound for valid inputs

`int upperBound` — the upper bound for valid inputs

#### **Action:**

The program prints out the range of valid integers. The program then waits for the next line entered by the player, terminated by a carriage return, and determines if the entry is an integer in the range of allowed integers. If it is, the program stores the integer in the variable passed in by the programmer. Otherwise, the program waits until a valid integer is entered.

### **declareWinner(int playerID)**

#### **Called in:**

Can be called anywhere in the program, but is usually found as a function call in the `goalCheck()` function in the Board block.

**Arguments:**

`int playerId` — the ID of the player that will be declared the winner of the game.

**Action:**

The program will print to the console that the specified player won the game and the game terminates.

**declareWinner()****Called in:**

Can be called anywhere in the program, but is usually found as a function call in the `goalCheck()` function in the Board block.

**Arguments:**

<none>

**Action:**

The game terminates. Notice that the `declareWinner()` function is overloaded. The version with a `playerID` argument declares a winner, while this version with no arguments declares no winner.

**move(int playerId, int pieceID, int distance)****Called in:**

Can be called in the `roll()` functions of the Dice or the Deck blocks or from the `landsOn()` function of the Board block.

**Arguments:**

`int playerId` — the ID of the player whose piece is to be moved.

`int pieceID` — the ID of the piece to be moved.

`int distance` — the number of squares the piece will be moved forward.

**Action:**

When the `move()` function is called, the current player's piece (specified by `playerID` and `pieceID` in the argument list) arguments moved forward by the distance specified in the argument.

**moveReverse(int playerId, int pieceID, int distance)****Called in:**

Can be called in the `roll()` functions of the Dice or the Deck blocks or the `landsOn()` function of the Board block.

**Arguments:**

`int playerID` — the ID of the player whose piece is to be moved.  
`int pieceID` — the ID of the piece to be moved.  
`int distance` — the number of squares the piece will be moved backward.

**Action:**

When the `moveReverse()` function is called, the current player's piece (specified by `playerID` and `pieceID` in the argument list) arguments moved backward by the distance specified in the argument.

**jump(int playerID, int pieceID, int tileID)**

**Called in:**

Can be called from the `roll()` functions of the `Dice` and the `Deck` blocks. Can also be called from the `landsOn()` function of `Tiles` in the `Board` block.

**Arguments:**

`int playerID` — the ID of the player whose piece is to be moved.  
`int pieceID` — the ID of the piece to be moved.  
`int tileID` — the ID of the destination tile the piece will be moved to.

**Action:**

When the `jump()` function is called, the current player's piece (specified by `playerID` and `pieceID` in the argument list) argument is moved directly to the tile with `tileID` specified in the argument.

**c. Function Templates That Can Be Defined By The Programmer (Dynamic Functions)**

ROLL also has a number of function templates that can be *defined* with code that lays out what happens when the function is called. We call these dynamic functions. These are `setupPlayers()`, `preRoll()`, `goalCheck()`, `landsOn()`, and `roll()`. The programmer cannot call these functions at any point in his/her code. These functions are called automatically at specific times during the flow of a game.

The arguments that these functions take are unchangeable. Thus, when implementing one of these functions, the programmer must write the function signature exactly as it appears in the LRM below. For example, when defining the `roll()` method, the programmer

must write `define roll(int amountRolled, int playerID)` exactly as is, and then may place his/her desired code afterwards, within braces.

Also note that the programmer does not *have* to define all of these functions. Any unimplemented function will be replaced with the language's default definition of that function.

### **i. Single Definition vs. Multiple Definition Functions**

Some dynamic functions, namely `setupPlayers()`, `preRoll()`, and `goalCheck()`, make sense only when there is a unique definition of how the function is carried out, so these functions can only have one definition in the program.

Two other dynamic functions, `landsOn()` and `roll()`, can have more than one definition. This allows these functions to have more than one way to carry out its purpose. For example, it would make sense for a program to have more than one `landsOn()` function to allow different tiles to have different reactions when a piece lands on that tile. The programmer would then write two separate definitions of the `landsOn()` function.

### **ii. Function Naming Mechanism**

When the programmer provides more than one definition of a function, each definition must be given a different name. The syntax for naming a function starts with the word `function`, followed by the name. The programmer then states which general function template he is implementing by typing `= define` and then the name of the function template. For example, here the programmer creates a new function named `directionRoll`, which is based off of the `roll()` function template:

```
function directionRoll = define roll(int
amountRolled, int playerID) {
    //Function implementation goes here
}
```

\*Note that new `roll()` or `landsOn()` functions may not be named `default`, as it is a reserved word in ROLL and will cause errors.

### **iii. Argument Enumeration**

The argument list of a function template enumerates the variables that are available to the programmer when he writes the function

definition. For example, the `roll(int amountRolled, int playerID)` function template provides for the two function variables `amountRolled` and `playerID`, whose values are the amount that was rolled on the dice, and the ID of the player whose turn it is. These two variables exist only in the definition of the `roll()` function. Note that the programmer cannot change these arguments. Each function has a specified argument list that the programmer cannot change; the reason for including them in the function definition is so the programmer knows what variables he is allowed to refer to. Variables the programmer would like to refer to that are not in the argument list would have to be either block-level fields in scope (such as `NumPlayers` and `NumPieces`), or global or local variables he has defined himself.

Return Types:

The functions in the ROLL language do not have return types. The `promptName()`, `promptList()`, and `promptRange()` functions ask for a variable integer or text as one of their arguments to store the value entered by the user.

#### iv. Dynamic Function List (Single Definition)

##### **setupPlayers()**

**Defined in:**

`Players` block

**Time called:**

This is a function not explicitly called by the programmer. It is called automatically once at the beginning of the game when the board is set up.

**Arguments:**

<none>

**Action:**

This function is called at the beginning of the game after the board is set up. This function assists with determining the run time variables such as the number of players and the names of the players. All the code that is necessary to determine the properties of players should be inside this function. The programmer cannot name this function.

**Default action if no function is written:**

Determines the number of players and their names

##### **preRoll(int playerID)**

**Defined in:**

Board block

**Time called:**

This is a function not explicitly called by the programmer. It is called automatically multiple times over the course of the game, once before a player's turn begins.

**Arguments:**

`int playerID` — the ID of the player whose turn is about to begin

**Action:**

This function is called once every time a player is about to begin a turn. The programmer should include in this function's definition all the code that should be executed when a player turn begins. Usually this includes printing to the screen a message indicating whose turn it is. The programmer cannot name this function.

**Default action if no function is written:**

Prints whose turn it is

**goalCheck(int playerID, int tileID)**

**Defined in:**

Board block

**Time called:**

This is a function not explicitly called by the programmer. It is called automatically multiple times over the course of the game, once after each player's turn ends.

**Arguments:**

`int playerID` — the ID of the player whose turn just ended

`int tileID` — the ID of the tile onto which the player just moved a piece

**Action:**

This function is called once every time a player ends a turn. The programmer should include in this function's definition all the code necessary to determine if a player has won the game. The programmer cannot name this function.

**Default action if no function is written:**

Checks if the piece that was just moved has reached the tile with ID `NumTiles - 1`

**roll(int amountRolled, int playerID)**

**Defined in:**

The Dice block

**Time called:**

The `roll()` function is called each time the dice are rolled.

**Arguments:**

`int amountRolled` — The value of the card that was selected.  
`int playerID` — The player who drew a card.

**Action:**

The code in the definition of the `roll()` function carries out the actions that take place during a player turn.

**Default action if no function is written:**

Moves the player's piece with ID 0 forward the number that was rolled on the dice.

## v. Dynamic Function List (Multiple Definition)

### **landsOn(int playerID, int pieceID, int tileID)**

**Defined in:**

Board block

**Time called:**

This is a function not explicitly called by the programmer. Each `Tile` data type is linked to a named definition of the `landsOn` function. The `landsOn` function is called automatically when a piece lands on the tile via the `move()`, `moveReverse()`, or `jump()` function.

**Arguments:**

`int playerID` — the owner of the piece that just landed on the tile

`int pieceID` — the piece that just landed on the tile

`int tileID` — the tile onto which a piece just landed

**Action:**

This function is called once automatically every time a piece moves onto a tile. The programmer can write multiple definitions for this function template and provide each definition a different name. The `landsOn` function is linked to a `Tile` data type by specifying the named definition of the function template in the `make Tile` command.

**Default action if no function is written:**

Prints a message saying that the piece landed on that tile

### **roll(int amountRolled, int playerID)**

**Defined in:**

The Deck block

**Time called:**

This is a function not explicitly called by the programmer. Each `Card` data type in the Deck block is linked to a definition of the `roll()` function template. The `roll()` function is called each time a card is drawn from the deck.

**Arguments:**

`int amountRolled` — The value of the card that was selected.

`int playerId` — The player who's turn it is.

**Action:**

The code in the definition of the `roll ( )` function carries out the actions that take place during a player turn.

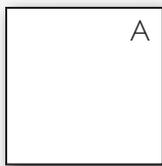
**Default action if no function is written:**

Moves the player's piece with ID 0 forward by the value of the card that was drawn

## 11. Reserved Words

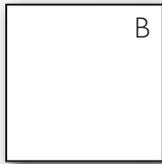
ROLL contains several words that are unique to the language. Using these words out of their context (i.e. as a user-defined variable) will result in a compile-time error. These words include the block names, block-level fields, and object data types seen throughout the LRM. In addition, since ROLL source code is translated into Java code, all Java reserved words are reserved in ROLL as well. The following is the list of ROLL reserved words:

abstract	faces	move	protected
accessible	final	moveReverse	public
assert	finally	name	quantity
Board	FinishOn	native	return
Boolean	FinishOn	new	roll
break	float	next	short
byte	for	NextTurn	StartOn
Card	function	NumPieces	static
case	Game	NumPlayers	strictfp
catch	goto	NumTiles	super
char	GUI	NumTiles	switch
class	HasReplacement	occupiedTileId	synchronized
const	id	occupierID	this
continue	if	package	throw
Deck	implements	Piece	throws
declareWinner	import	PieceList	Tile
default	instanceof	Player	TileList
define	int	PlayerList	transient
Dice	interface	Players	try
Die	jump	prev	value
do	landsOn	print	void
double	long	private	volatile
else	make	promptList	while
enum	MaxPlayers	promptName	
extends	MinPlayers	promptRange	



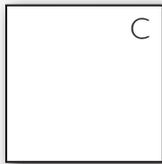
A

Introduction



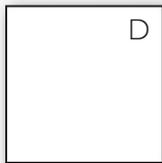
B

Language Tutorial



C

Primer on Organizing  
ROLL Code



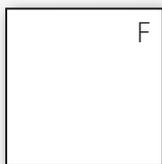
D

Language Reference  
Manual



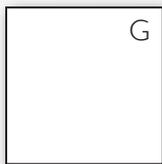
E

Project Plan



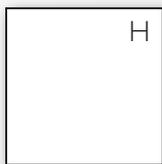
F

Language Evolution



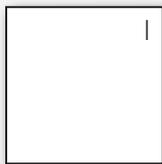
G

Translator  
Architecture



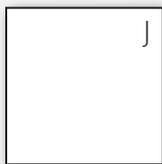
H

Development  
Environment



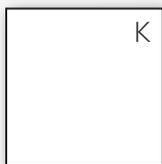
I

Test Plan



J

Conclusions



K

Appendix

# Project Plan

*Lauren Pully,  
Project Manager*



# Project Plan

## Lauren Pully

### *Project Manager*

#### *Process*

We used an incremental development approach when creating the ROLL translator – we broke the project into small milestones and made sure that we tracked the progress every step of the way.

We met the first weekend of the semester to brainstorm language ideas. After two long brainstorming sessions we finally settled on ROLL, a language for creating and playing board games. We used weekly status meetings as a chance to update each other on the tasks completed throughout the previous week. We would then spend time determining our next steps and finally iron out a schedule for tasks to be completed during the following week.

#### *Responsibilities*

Our official team roles are divided as:

Lauren Pully (Project Manager)  
Jesse Bentert (Tools & Language Guru)  
John Graham (System Architect)  
Daniel Wilkey (System Integrator)  
Yipeng Huang (Tester & Validator)

In reality, however, definitive team rolls blended together as our team collaborated on several parts of the project, despite our various distinct roles. After writing our whitepaper together we split into two groups. Jesse, John, and Lauren started to design the Java backend while Dan and Yipeng started outlining ROLL programs. Once we had a basic outline of both a ROLL program and the Java backend, we took a break from code design to write our Language Reference Manual and Tutorial. After completion, we resumed code design with new team roles. Jesse and Dan worked on the grammar and parser while John coded sample ROLL programs. Lauren worked on the GUI. Yipeng authored and ran the entire testing suite.

#### *Implementation Style Sheet*

We formatted our code using a basic style sheet similar to that of Eclipse. To a Java programmer, this should appear familiar. A screen shot of sample ROLL code is included below:

```
Game SampleProgram {
    int $randomVar = 1;
    Players {
        GUI = 1;
        StartOn = {0,5,2};
    }
}
```

```

    FinishOn = {9,9,9};

    define setupPlayers(){
        print(StartOn[2]);
    }
}

Board{
    NumTiles = 10;
    ...
}

```

### *Timeline*

The following timeline includes our team’s project meetings and weekly milestones:

<i><b>Date</b></i>	<i><b>Milestone</b></i>
1/21	Formed Team
1/31	First Team Meeting, Discussed Possibilities for a Language
2/7	Finalized idea for Language, Began discussing “Buzz Words”
2/13	Wrote whitepaper as a team
2/21	Finished whitepaper, split up back-end and ROLL code responsibilities
2/24	Whitepaper due
2/27	Discussed ROLL language ideas, basic ROLL programs, and backend design
3/6	Divided up responsibilities for Language Reference Manual and Tutorial
3/19	Compiled first draft of documentations
3/20	Met to discuss documentations
3/22	Finished documentations
3/28	Split up responsibilities for finishing compiler
4/3	Discussed progress on compiler roles
4/7	Rough draft of grammar complete, started parsing
4/10	Discussed grammar and parsing changes, divided up responsibilities to complete the Java backend, discussed presentation
4/18	Discussed error handling, parsing progress, and backend progress, rehearsed presentation
4/21	Discussed parsing and GUI progress, rehearsed presentation
4/23	Rehearsed presentation
4/24	Rehearsed presentation
4/25	Rehearsed presentation
4/26	Presentation in Class
5/3	Created a final to do list for our project and distributed the work left to be done
5/7	Finished documentation

### *Project Log*

The following project log divides our group’s major milestones into individual timelines.

### **Whitepaper**

The whitepaper gave our team an opportunity to really get to know one another while

brainstorming our ideas for the semester. We spent our first meeting throwing out ideas and narrowed it down to a couple of choices by the end of two hours. The next week we agreed on a language idea and started to talk about features that we wanted it to have. We spent the next couple of weeks solidifying these ideas and completing our whitepaper.

<i>Date</i>	<i>Milestone</i>
1/21	Formed Team
1/31	First Team Meeting, Discussed Possibilities for a Language
2/7	Finalized idea for Language, Began discussing “Buzz Words”
2/13	Wrote whitepaper as a team
2/21	Finished whitepaper
2/24	Whitepaper due

### **Language Syntax Design**

One of our first milestones was to design the ROLL syntax. We tried to keep in mind that ROLL was designed to be simple, and that it should be easy to understand from our whitepaper. We started by writing both sample constructs and basic programs.

<i>Date</i>	<i>Milestone</i>
2/21	Started talking about ROLL syntax
2/24	Whitepaper due
2/27	Discussed ROLL language ideas and basic ROLL programs
3/6	Wrote short ROLL programs
5/7	Wrote more complex ROLL programs for our testing suite

### **Java Backend Design**

As we designed the ROLL code we discussed features that our Java backend would need to include. We started working on the basic versions of our Java backend to support these features. As we continued to add features to our ROLL language we added corresponding support in our backend. We also decided to add a GUI.

<i>Date</i>	<i>Milestone</i>
2/21	Started discussing Java backend
2/27	Discussed sample programs in backend
3/6	Finished basic Java, command line backend
4/18	Updated Java backend to support new ROLL features, began working on GUI
4/21	Discussed GUI progress
5/3	Generalized GUI Complete
5/7	Customized GUI complete

### **Language Reference Manual / Language Tutorial**

Writing the language reference manual provided our team an opportunity to put all our concrete ideas onto paper.

<i>Date</i>	<i>Milestone</i>
-------------	------------------

3/6	Divided up responsibilities for Language Reference Manual and Tutorial
3/19	Compiled first draft of documentations
3/20	Met to discuss documentations
3/22	Finished documentations
5/7	Finished documentation

### **Grammar and Parsing**

After writing some sample ROLL programs for our documentation, we had a good idea of what the grammar should look like. We started writing out parts of the grammar and then began transferring those into Lex and Yacc for parsing.

<i>Date</i>	<i>Milestone</i>
3/28	Started writing grammar and parsing it
4/3	Discussed progress on compiler roles
4/7	Rough draft of grammar complete, started parsing and testing
4/10	Discussed grammar and parsing changes
4/18	Discussed error handling and parsing progress
4/21	Discussed parsing progress
5/7	Updated grammar to reflect parsing errors found in testing

### **Presentation**

The class presentation gave our team a chance to step back from coding our compiler and present our accomplishments from the semester.

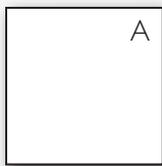
<i>Date</i>	<i>Milestone</i>
4/10	Discussed grammar and parsing changes, divided up responsibilities to complete the Java backend, discussed presentation
4/18	Discussed error handling, parsing progress, and backend progress, rehearsed presentation
4/21	Discussed parsing and GUI progress, rehearsed presentation
4/23	Rehearsed presentation
4/24	Rehearsed presentation
4/25	Rehearsed presentation
4/26	Presentation in Class
5/3	Created a final to do list for our project and distributed the work left to be done
5/7	Finished documentation

### **Final Report**

The final report was an opportunity to revise and compile all of the documentation we had created throughout the semester.

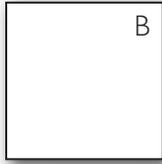
<i>Date</i>	<i>Milestone</i>
2/24	Whitepaper due
3/22	Language Reference Manual and Tutorial Due
5/7	Finished final report





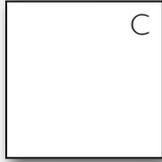
A

Introduction



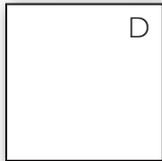
B

Language Tutorial



C

Primer on Organizing  
ROLL Code



D

Language Reference  
Manual



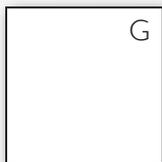
E

Project Plan



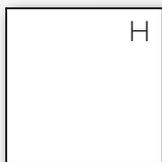
F

Language Evolution



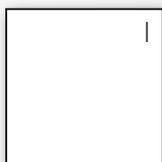
G

Translator  
Architecture



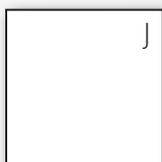
H

Development  
Environment



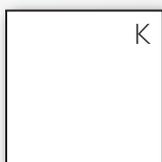
I

Test Plan



J

Conclusions



K

Appendix

# Language Evolution

*Jesse Bentert,  
Tools & Language Guru*



## Language Evolution

**Jesse Bentert**

*System Architect*

Our motive for ROLL was to design a syntax that was emblematic of the constructs found in an actual board game. We figured that this would make programming in ROLL simpler and more intuitive. We realized that there are three items common to all board games: the board, the dice, and the players. We thus decided to create one section of code, dubbed blocks, for each of these. In accordance with this initial decision, we tried to incorporate every subsequent language construct into one of the three abstractions to maintain a clear divide between the component blocks.

In addition, we concluded that there are many actions common to all board games, like moving a piece or rolling a die. We realized that we could code these generic constructs ourselves and shield them from the ROLL programmer. In this way, a programmer who writes in our language would only have to focus on the features specific to his own game and would not have to write code for those things common to all board games. Thus, every line of code that a ROLL programmer writes is specific to their game.

To make it easy for the programmer to code these specifics, we decided to give each block an associated set of attributes that the programmer can vary. We came up with three different forms for these attributes: fields that could be set, objects that could be created, and functions that the programmer could define.

Fields are the board game constructs that vary from game to game whose values can be specified by the programmer. For example, we created a variable NumTiles whose value can be

set in the board block by including `NumTiles = <someNumber>;`, where `<someNumber>` can be any positive integer. The fields related to a board are set in the board block; likewise, the fields related to players are set in the players block.

Instantiating objects allows gives the programmer more functionality than setting fields. For example, although setting the number of tiles on the board is as easy as giving `NumTiles` a value, we also allow the programmer to give each tile a specific behavior. We made this possible by allowing the programmer to explicitly create each tile object, specifying the attributes that he wanted each one to have. For example, to create a tile, you could write “make `Tile(id: 3, next: 4, prev: 2).`”

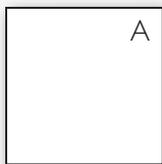
We realized that programmers might want to give their board games behavior that they could not specify by only setting fields and creating objects. We thus created a set of functions for each block that the programmer would define himself. These functions are automatically called at appropriate times throughout the course of the game, meaning the programmer only has to specify their desired action and with which tile they’re associated. For example, the `preRoll()` function allows the programmer to specify the action carried out before each player’s turn.

One challenge in developing ROLL was ensuring that, for each new construct we added, we maintained the features we agreed upon in our original language proposal. We periodically evaluated our language against that proposal. One of our main goals was that the programmer need not provide the code common to all games. But after adding functions to our language, we realized that this was not the case. For example, ROLL programmers can write a `setupPlayers()` function that retrieves information about the players in the game. Most board game creators are probably going to do the same thing in this function: determine how many players there are and get their names as input. We did not want to get rid of this function entirely in case the

programmer *did* want to implement some custom behavior, but we did not want to *force* them to write it, since this behavior is so common. Thus we developed the concept of defaulting. The programmer can omit any function that he wants, in which case it is replaced with a default that we provide. If the programmer is satisfied with just asking for the number of players and their names, he can leave out the `setupPlayers()` function. If, instead, he wants to do something more complex, like give each player some money to start, then he can write the `setupPlayers()` method to do so.

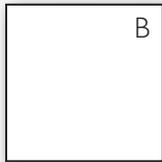
We then realized we could extend this idea to fields and blocks. Programmers can omit most of the block fields, or even entire blocks. Doing so allows them to use our default implementation for that field or block, thus granting programmers the opportunity to customize their game while leaving out any code common to all games

In order for all team members to be informed of language changes as we developed ROLL, we had to be in constant communication. Our main method of communication was email. Anytime a new syntactic construct was created, an email was sent to all team members. However, email cannot replace in person meetings; therefore, we had biweekly meetings in which we discussed any changes that had been made since the last meeting. It was sometimes difficult for everyone to be up to speed with our evolving syntax, but being in constant contact enabled us to resolve any conflicts.



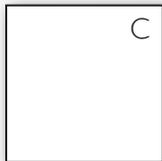
A

Introduction



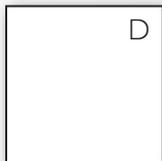
B

Language Tutorial



C

Primer on Organizing  
ROLL Code



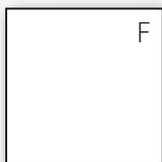
D

Language Reference  
Manual



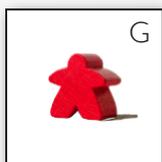
E

Project Plan



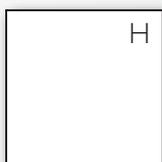
F

Language Evolution



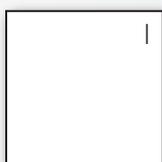
G

Translator  
Architecture



H

Development  
Environment



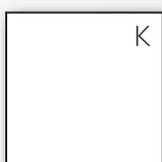
I

Test Plan



J

Conclusions



K

Appendix

# Translator Architecture

*John Graham,  
System Architect*



# Translator Architecture

**John Graham**

*System Architect*

ROLL compilation begins with our compilation script, `rollc`. Using the command `rollc <game-name>`, a ROLL source file is translated to executable code. Although this is all the application programmer will see for an errorless program, there is a great deal going on behind the scenes between these two states of the specified board game.

Our lexical analyzing is done using Flex and both our syntactical and semantic analysis are handled by Yacc. Both Flex and Yacc compile to C source code and can be subsequently compiled and linked together using `gcc`. We refer to this generated file simply as the ROLL compiler. Passing ROLL source code to this compiler comprises the first step of our translation architecture. In practice though, the ROLL compiler may be generated before compilation of a ROLL source program begins.

The translation of ROLL source code actually begins when this code reaches our lexical analyzer, `roll.l`. This Flex file uses a series of regular expressions to recognize valid tokens from the source code and generate a token stream. Any unrecognized characters are echoed back to the standard out to indicate that they are invalid tokens, though compilation proceeds as normal. It is possible for an error to be thrown at this stage if a string from the source file with valid characters cannot be mapped to a token. In all other events, the token stream generated by the lexical analyzer is given as input to our syntax analyzer, `roll.y`.

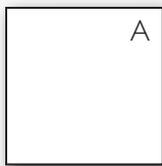
The goal of the ROLL syntax analyzer is to use a syntax directed translation scheme to convert ROLL source code to syntactically correct and semantically equivalent Java source code.

We use purely bottom-up parsing productions to turn roll syntax into four Java classes, which integrate with our backend framework. Our translation is achieved through a series of helper functions that perform simple C string manipulation. For each parsing action, the specifics of the current production are concatenated with the synthesized attributes of the production's children (either may be the empty string). Because we do not inspect the attributes of a production's parents or siblings, no top-down parsing is needed (though Yacc does provide this capability). The majority of type and syntax errors are caught at this point in compilation and thrown back to the programmer. Once the Java files have been generated and output from the ROLL compiler, control is passed to the Java compiler for the next stage of translation.

The four output Java files (`GlobalVariables.java`, `PlayersImplementation.java`, `BoardImplementation.java`, and `DiceImplementation.java`) created for each ROLL source program are compiled together with their backend counterparts using `javac` to yield an intermediate representation of the board game (i.e. a set of Java class files). It is possible at this stage to receive Java errors for such things as undefined variables. Two of the more attractive features of using Java for our backend are its robust error checking and the portability of its generated class files. After this stage of compilation, the working board game may be ported to any machine that has the JVM installed.

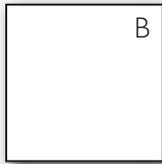
Finally, we include a second script with our language entitled `roll` that executes the Java code. Using Java's just-in-time compiler, the board game is finally converted to machine language at runtime and the game can be played. As a very last line of defense, runtime errors may be thrown for problems such as use of null values or array indices out of bounds, which may slip by both compilers. As you can see, the 'compilation' of a ROLL program actually involves

several sub-compilations before the end user may finally play a game like Sorry or Chutes and Ladders.



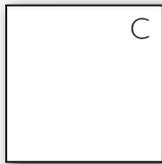
A

Introduction



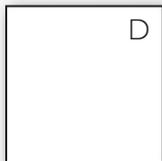
B

Language Tutorial



C

Primer on Organizing  
ROLL Code



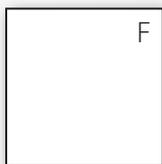
D

Language Reference  
Manual



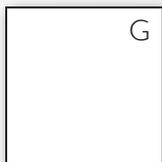
E

Project Plan



F

Language Evolution



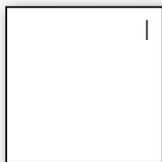
G

Translator  
Architecture



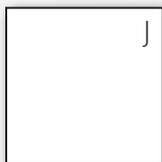
H

Development  
Environment



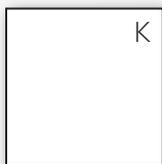
I

Test Plan



J

Conclusions



K

Appendix

# Development Environment

*Daniel Wilkey,  
System Integrator*



## **Development Environment**

**Daniel Wilkey**

*System Integrator*

Development of ROLL can be broken down into two stages. In the first stage, our primary goal was to create the outline of a Java backend framework. Our initial concept for the translator was to define syntax for specifying board game rules and compile those rules into Java files. These Java files would integrate with our backend framework to construct a complete, working program. During this phase, we developed using the Eclipse IDE because of its exceptional Java support. In order to facilitate version control, we chose to use the Eclipse Subversion plug-in, Subclipse. Finally, we integrated Subclipse with GoogleCode, where our files were actually stored and managed. This environment worked well for the initial phase because we did not have to think twice about editing any file at any time. If conflicts arose, they could be merged painlessly using the Subclipse GUI. This enabled us to rapidly develop our initial backend using Eclipse.

The issues with Eclipse began to arise when we transitioned to working on the compiler front end. Our choices of lexical analyzer, Flex, and syntax analyzer, Bison Yacc, do not integrate well with the Eclipse IDE. These languages use C for all helper functions, which, given the diversity of platforms on which we were developing, proved difficult to integrate with Eclipse. For example, Mac and Linux machines come with built-in C compilers, but when developing on Windows, both third party software and careful integration are required to handle C files. Due to these complications of working with others, we decided to move all front-end development to the local platform. The negative ramifications of this decision were the

newfound need for manual version control and file management. The upside, however, was that our entire front end could be compiled and executed using a single Makefile. Each team member used either a Linux environment (Mac or Ubuntu) or remoted to one (such as a Clic machine) in order to work on the files. Using Linux: C, Java, Flex, and Yacc compilers can all be seamlessly installed into the same path. Using the Makefile command, `make test<N>` (where N is replaced with the number of the test), we were able to build the lexer, the parser, link the two into a single C executable, run the compiler with the specified test file, and generate the Java backend.

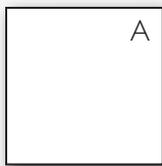
Though this strategy worked exceptionally well, we still faced one more glaring problem. Now that our frontend and backend were being developed in different environments, testing the entire translator (once we reached the point of merging the two ends) was not easy. This required manually copying all of the latest Java files from the Eclipse source folder into the frontend workspace before testing. Eventually the choice was made to abandon Eclipse entirely, which begs the question of how we handled version control.

Version control for our project was handled delicately. We decided to use Dropbox, which is simple freeware designed for general-purpose file sharing. This made it easy to import the latest files to each of our machines on command; however it offered no assistance for merging conflicted files. We implemented two major process solutions to avoid conflicts. First, we partitioned the work. Each group member (or pair of two) was assigned a subset of the files and modules on which to work. In theory, this would avoid all possibility of conflicts. However, we were not naïve enough to believe that the work one group was doing would never spill over into that of another group. Our policy for working on files outside of your assigned subset was to contact whoever was in charge of those files. Once you have ensured said person

does not have any uncommitted changes, you are allowed to edit the files yourself and notify the affected parties immediately after committing your updates. Given the complexity of this system, we astounded even ourselves with the pace at which we were able to develop the translator.

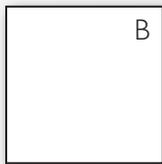
Inevitably, there were a few major conflicts and messy merges, but all in all, development progressed smoothly. A huge advantage of not using version control software is avoiding the overhead that goes along with it. Checking out, editing, testing, and committing changes could all be done very quickly using our self-made system. When development of our translator was nearly completed, we addressed the problem of what sort of compiler tools to ship to the user.

Instead of requiring the programmer to use our messy testing Makefile in the finished product, we instead wrote a very simple shell script, which compiles the ROLL source code (both frontend and backend) and generates an executable. A second script runs the compiled game. Partitioning the compilation strategies between a Makefile and scripts offers the end programmer a choice. They have the option to use our development Makefile, which allows for more informative testing and debugging but is more difficult to understand and customize. Alternatively, the user has the option to compile and run using our very simple and intuitive scripts without ever worrying about the nuts and bolts of the compiler architecture. Thus is the story of developing the best little language for implementing custom board games: ROLL.



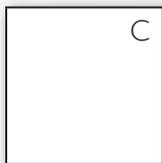
A

Introduction



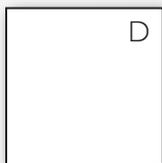
B

Language Tutorial



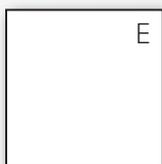
C

Primer on Organizing  
ROLL Code



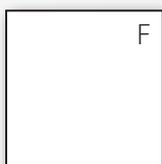
D

Language Reference  
Manual



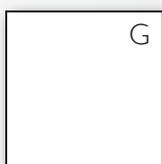
E

Project Plan



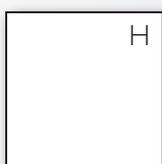
F

Language Evolution



G

Translator  
Architecture



H

Development  
Environment



I

Test Plan



J

Conclusions



K

Appendix

# Test Plan

*Yipeng Huang,  
Tester & Validator*



# Test Plan

**Yipeng Huang**

*Tester & Validator*

## Overview

Testing the ROLL compiler was done in three phases:

- 1) Automated regression test, which was done as new productions were added to the Yacc parser. This test ensured that changes to the compiler did not interfere with functionality that had already been implemented and tested.
- 2) Quality control test, which was done after the compiler front end was complete to check for proper translation of language constructs. This test ensured that all the features promised in the Language Reference Manual were delivered and that the compiler was robust.
- 3) Whole program behavior test, which was done after each new game was written in ROLL to check that the game framework handled the rules correctly and generated a game that behaved according to the programmer's intent. This test ensured that generated programs were semantically equivalent to the ROLL source.

## Automated regression test

This test was done after each new production was added to the Yacc parser to make sure the new language feature did not interfere with translating any other language construct. As each language feature was added, a new ROLL program would be written to specifically test that new functionality. These small ROLL programs accumulated and were compiled again every time a later feature was added.

## Automation using makefile

A makefile automated the compilation of this set of ROLL programs:

```
#makefile for the ROLL compiler

#c compilation
CC = gcc
CFLAGS = -g
LFLAGS = -ly -ll

#java compilation
JFLAGS = -g
JC = javac
.SUFFIXES: .java .class
.java.class:
    $(JC) $(JFLAGS) $*.java

#compiler source files
rlex = roll.l
ryacc = roll.y
tyacc = my.y
clex = lex.yy.c
cyacc = y.tab.c
name = game.r
```

```

OUTPUTS_TO_CLEAN = BoardImplementation.java DiceImplementation.java
PlayersImplementation.java y.tab.h

#test files
t1 = Empty.roll
t2 = NewHello.roll
t3 = test3.roll
t4 = test4.roll
t5 = ArrayTest.roll
t6 = Statements.roll
t7 = test7.roll
t8 = test8.roll
t9 = Default.roll
t10 = advancedChutes.roll
t11 = test11.roll
t12 = IncrementalTest.roll
t13 = Sorry.roll

#backend source files
CLASSES = \
    Board.java \
    Dice.java \
    Driver.java \
    Game.java \
    Player.java \
    Players.java \
    Piece.java \
    Tile.java \
    Die.java

#make the lexer
lexer: $(rlex)
    flex $(rlex)

#make the syntax analyzer
parser: $(ryacc)
    yacc -d $(ryacc)

#make the test parser
tparser: $(tyacc)
    yacc -d $(tyacc)

#make the compiler
compiler: clean lexer parser
    $(CC) $(CFLAGS) -o $(name) $(cyacc) $(clex) $(LFLAGS)

#make the test compiler
tcompiler: clean lexer tparser
    $(CC) $(CFLAGS) -o $(name) $(cyacc) $(clex) $(LFLAGS)

#run all tests
test: test1 test2 test3 test4 test5 test6 test7 test8 test9 test10 test11 test12 test13

#default test
test1: compiler $(t1)
    cat $(t1) | ./game.r

#hello world test
test2: compiler $(t2)
    cat $(t2) | ./game.r

#dice block test file
test3: compiler $(t3)
    cat $(t3) | ./game.r

#deck block test file
test4: compiler $(t4)
    cat $(t4) | ./game.r

```

```

#in place array test file
test5: compiler $(t5)
      cat $(t5) | ./game.r

#simple statements test
test6: compiler $(t6)
      cat $(t6) | ./game.r

#prompt test
test7: compiler $(t7)
      cat $(t7) | ./game.r

#setupplayers test
test8: compiler $(t8)
      cat $(t8) | ./game.r

#default test
test9: compiler $(t9)
      cat $(t9) | ./game.r

#advancedChutes test
test10: compiler $(t10)
       cat $(t10) | ./game.r

#global variable reassignment test
test11: compiler $(t11)
       cat $(t11) | ./game.r

#IncrementalTest test
test12: compiler $(t12)
       cat $(t12) | ./game.r

#Sorry game test
test13: compiler $(t13)
       cat $(t13) | ./game.r

#make the backend
classes:
    javac Driver.java

run: classes
    java Driver

#clean output files
clean:
    -rm -f $(clex) $(cyacc) $(name) *.class *~ $(OUTPUTS_TO_CLEAN)

default: compiler

all: clean compiler

```

### The test suite

The ROLL programs included in the regression test suite covered a broad range of language elements, including syntax, comments, types, variable instantiation, initialization, and reassignment, arithmetic operators, logical operators, and text operators, and larger constructs such as conditionals and for-each loops.

## Quality control test

The automated regression tests described in the previous section guaranteed that the compiler accepted existing programs as we added new features, but there was minimal guarantee from regression testing that the semantics of ROLL programs were properly preserved in the generated code.

Since the compiler front end and the backend Java framework evolved rapidly during compiler development, it was difficult to test for correct semantics using just the automated regression test. It was therefore necessary to test for the overall quality of the compiler once the compiler became set in stone.

The quality control test took place after the compiler productions were fully implemented and after the semantics of fundamental ROLL language features were well preserved, even after many more layers of productions were added to the compiler.

## Verifying features promised by the Language Reference Manual

To conduct the quality control test, we developed a single ROLL source file and incrementally added code that utilized features mentioned in the Language Reference Manual. These series of tests were organized in the same fashion as the Language Reference Manual, guaranteeing that everything mentioned in the Manual was incorporated. In the end, this test covered language constructs as small as ASCII character recognition to higher-level constructs like conditionals and for-each loops.

The ROLL source code used for quality control testing follows:

```
...

    int
$myGlobalVariable1234567890qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM@#$$%^&_\`
= -2147483648;
    int $myGV1 = -2147483648;
    int $myGV2 = 2147483647;
    int $myGVChange = 600;

...

        StartOn = {-1000000 ~ 1000000};

        define setupPlayers()
        {

            // print and comment tests
            // print("This shouldn't print.");
            print("This // should print.");
            print("This should print as well."); // end of line comment

            // text test
            print("All ASCII printable special characters:
~!@#$$%^&*()_+`1234567890-={|[]:;'<>?,./\"");

            // global variable test
            print("My ridiculous global variable = " |
$myGlobalVariable1234567890qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM@#$$%^&_\` )
;

            // integer bounds test
            print("lower bound integer = " | $myGV1);
            print("upper bound integer = " | $myGV2);
```

```

// global variable reassignment test
print("$myGVChange = " | $myGVChange);
$myGVChange = 700;
print("reassignment: $myGVChange = " | $myGVChange);

// array size test
print("StartOn[2000000] = " | StartOn[2000000]);

// arithmetic test
print("(10/2) = " | (10/2));
print("(10/3) = " | (10/3));
print("(2147483647/3)*3 = " | (2147483647/3)*3);
print("(1+1+1+1)*2 = " | (1+1+1+1)*2);
print("(1+1-1-1) = " | (1+1-1-1));
print("(1--1) = " | (1--1));
print("(2+12)/(3+1+3) = " | (2+12)/(3+1+3));
print("7+9 = " | 7+9);
//print("1/0 = " | 1/0);

// ++ and -- test
int i = 20;
print ("int i = " | i);
print ("int i++ = " | i++);
print ("int i = " | i);
print("(1--1) = " | (1--1));

// relop and logical operators test
int a = 7;
int b = 9;
if ((a==7)&&(a==a)&&(b>a)&&(a<b)&&(a!=b)&&!((a>b)|| (b<a)|| (a==b)))
{
    print("This line should print");
}
if
(!((a<=b)&&(a<=a)&&(b>=a)&&(b<=b)&&(a!=b)&&!((b<=a)|| (a>=b)|| (a==b)))) {
    print("This line should NOT print");
}

// if-else statements test
if (a==a) {
    if (b==b) {
        if (a==b) {
            print("IF ELSE TEST: This should not
print.");
        } else if (a<a) {
            print("IF ELSE TEST: This should not
print.");
        } else {
            if (a==a) {
                print("IF ELSE TEST: This should
print.");
            }
        }
    }
}

// array bounding test
print("***Array bounding using variables test");
int bound = 70;
int #myArray = {bound ~ $myGVChange};
print("myArray's upper bound = " | #myArray[69]);
int #myArray2 = {2, 3, bound};
print ("myArray2[0] = " | #myArray2[0]);
print ("myArray2[1] = " | #myArray2[1]);
print ("myArray2[2] = " | #myArray2[2]);

print("int bound's value is now = " | bound);

```

```
//empty print statement test
print("empty print statement test follows");
print("");
print("empty print statement test complete");
```

...

The results of this test can be found in the section, *ROLL Validation Documentation*, in the appendix.

### **Stress testing**

As part of quality control testing, the test case was written to stress test the compiler and the Java backend framework. The test case used very large integer values, arrays of very large size, and highly nested conditionals and for-each loops that iterated many times to test that the generated programs executed at reasonable speed.

### **Whole program test**

The quality control test was suitable for testing language features that are relatively low-level, such as data types, arithmetic, and loops. To test that the compiler preserved the semantics for larger language constructs such as block-level fields, object data types, user IO, and functions, we switched the testing methodology to test the compiler with whole games.

### **Testing of larger language constructs**

In the process of developing the compiler and preparing for various checkpoints, the team created a number of complete ROLL games ranging from the relatively simple default game, to more complex games such as *Chutes and Ladders*, and ultimately to a highly complex game, *Sorry!*.

Since the code for these games draws on a wide variety of language constructs, fields, object data types, and functions, testing for correct behavior in the generated games from these ROLL programs was a strong test on the validity of the compiler and the backend framework.

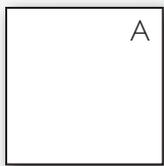
### **Testing behavior of games**

In designing a game's test suite, we took note of what fields and functions each individual game utilized, and made sure that the all fields, functions, object data types, and blocks were utilized among the set of test games.

For each game, we verified that the game preserved the semantics of the ROLL source code by checking for correct game behavior at all stages of the game.

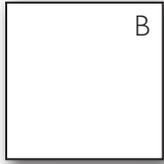
The results of this test can be found in the section, *ROLL Validation Documentation*, in the appendix.





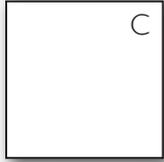
A

Introduction



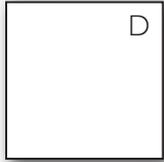
B

Language Tutorial



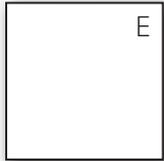
C

Primer on Organizing  
ROLL Code



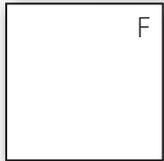
D

Language Reference  
Manual



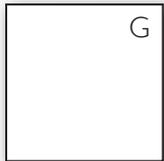
E

Project Plan



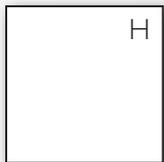
F

Language Evolution



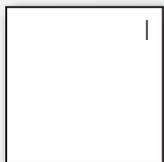
G

Translator  
Architecture



H

Development  
Environment



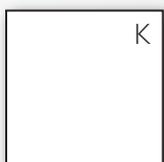
I

Test Plan



J

Conclusions



K

Appendix

# Conclusions



# CONCLUSIONS

## Lessons learned as a team

Working on a team is a very different from working solo. Coordinating schedules, making sure that code isn't omitted with shared files, and verifying that others' work is submission-ready are some of the issues we've run into as a team. That being said though, we came into this class as a group of five friends. We knew we were working together and we set early deadlines. Even despite these efforts to finish in a timely manner, the programming lesson we learned from Prof. Cannon in 1004 remains: "Multiply your expected time by 4, then multiply that by 10 to get your actual coding time."

However it has been a truly successful class, and we all agree that it has been one of the best classes we've ever taken at Columbia, if not the best. How many students can say they *wrote* their own programming language? And then actually write a program in it! How many students can put Flex and Yacc on their resume? Seeing that final board game show up, after all those intermediate compiling steps, was by far one of the most rewarding programs we've ever created. Rarely do you get a chance to write 1000+ line programs in a *custom* language.

It is also indescribably helpful to have a professor that is so genuinely helpful. He is beyond an expert in this field and knows how difficult the material and project are. Professor Aho knows what students have difficulty with, and really wants to see them succeed.

## **Lessons learned by each team member**

### **Lauren Pully**

#### ***Project Manager***

As project manager, it is important to make sure that the group stays on task. Although the semester seems like a lot of time, it will go by quickly. When everyone becomes excited about the details, it was important to help the team focus on the goal at hand – finishing our compiler. As project manager I had two approaches to this. First, it was important to make sure that everybody left every meeting with a concrete list of next steps. More importantly, however, it was necessary not to quell the enthusiasm at meetings when we talked about more high-level ideas. Instead I learned to wait these “deviations” out – it was during some of these deviations from the day’s stated goal that we accomplished the most. Once we had solidified some of our new ideas we were able to get back on track. Working on a group project with five people is not easy, but it can be fun!

### **John Graham**

#### ***System Architect***

As the system architect, I had a lot of trouble grasping all the steps of a complete translator. Paying attention in class and talking concepts through with other group members is very helpful in understanding theory. As a team member, I learned that you have to be proactive in your work. You can’t wait for someone to tell you that something needs to be done, especially if something needs to be done on your specific job function.

Getting caught up to speed is easier said than done! It helps to work with people that have a similar work ethic and are equally interested in the project (i.e. willing to stay up on a Friday night until 4:00am, but still passionate about the project). Stay on top of your work, and don't let you or any team members slip behind. Make use of positive feedback, it really promotes good work!

**Jesse Bentert**

***Language Guru***

As language guru, one thing I have learned from this project is that after the semester-long process of coming up with design decisions for our language, it is a challenging task to then present what we have designed in written form. Unlike my team members and me, our readers are unfamiliar with our language and have not been around it for an entire semester. What seems obvious to us is probably very confusing to them. Thus, we realized it was imperative to figure out how we could present our language in a clear and logical way, so that someone unfamiliar with our language could easily pick it up and understand how everything works. I was surprised at how time-consuming this task was; I assumed the majority of our time for this project would be devoted to actually coding. This goes to show that to be a computer scientist, it is not enough to be great at coding algorithms; one must be able to clearly communicate his/her ideas to the world.

**Daniel Wilkey**

***System Integrator***

First and foremost, I learned the importance of setting and keeping to a specific, actionable schedule. I have never before worked on a project of this scale and time frame and being constantly conscious of one's progress and both short-term and long-term goals is invaluable. There is no worse question to address upon completion of an assigned piece of work than 'what should I do next?' It is much easier to remain efficient when you have too much work than too little. Initially I believe that we were too preoccupied with the titles of our team roles and not the overall task at hand. The next lesson I learned was to start small. During the first few meetings of the semester, it is easy to start proposing as many features as pop into your head, but delivering on all of them in the long run is a much different story. I have found that it is much more satisfactory not to promise a feature at the onset and if, by chance, it ships with the finished product, then the user may be impressed by the added functionality. If the opposite occurs, then a natural reaction on the part of the end-user would be disappointment. Finally, I have learned it is important to stay grounded as a team. Every team member should have a good idea of the bigger picture and keep this image engrained throughout the process. If one team member is working to meet one goal and the rest are working towards another, it is easy to end up with an incongruous project at a late stage. When direction is clear, as long as team members are not working tangentially, progress can be achieved.

## **Yipeng Huang**

### ***Tester & Validator***

As compiler tester and validator, I had first-hand experience facing the challenges of formally testing the language and verifying that the features we promised in the Language Reference Manual are properly implemented in the language. Staying in constant contact with my teammates was the most important step in successfully testing the language. Because our team updated each other on language features changes, I was able to guarantee that all aspects of the language were tested and make appropriate bug reports and documentation changes to the Manual. In preparing and language documentation and the slides, I learned how important it is to establish clear terminology. Once we had agreed on the nomenclature of the various fields, functions, and other features of the language, ensuring that the presentation and the documentation were consistently worded became relatively easy. This project was most certainly a case where communication was the key to success, and I am thankful that my team was well equipped in this skill.

### **Advice for future teams**

The success of our project depended largely upon the fact that our group worked well together. Try to work in a group with people that you know – if you do not know anyone in the class then find a group early on and get to know them in a setting other than this

project. This allows you to work together as much as possible and enjoy meetings – you will be having a lot of them.

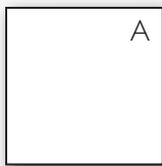
It is also important to start early. The end of the semester will come sooner than you think. A good goal is to have a working compiler for the presentation – though you will not be completely finished, this means that you do not have to spend your entire reading week working on the compiler.

Finally, enjoy your language and have fun with it! Make sure you pick something appealing to your entire team – you will spend a lot of time working with it.

### **Suggestions for the instructor on what topics to keep, drop, or add in future courses**

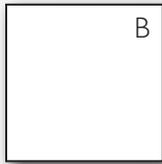
Programming Languages and Translators is unlike any class we've ever taken. The classes were interesting and well structured. The downloadable lecture outlines were very useful as a guide for taking notes in class. Both of the guest speakers were very interesting – it was great to hear people talk about the projects that they have worked on. We would definitely recommend inviting the guest speakers back. It would have been nice to have problem sets due throughout the semester (perhaps take the best 20/24 homework grades?). Though the practice questions were a good guideline for keeping up with work, the practice problems that we turned in after the semester were very helpful because we received relatively instant feedback on our performance. Overall we are all very happy that we took this class and definitely would recommend it to our peers.





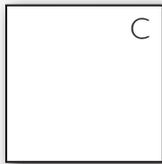
A

Introduction



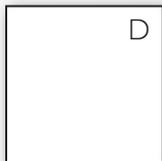
B

Language Tutorial



C

Primer on Organizing  
ROLL Code



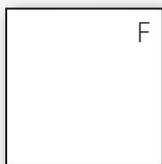
D

Language Reference  
Manual



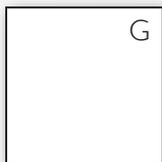
E

Project Plan



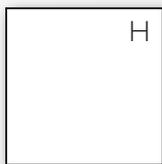
F

Language Evolution



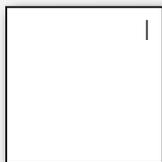
G

Translator  
Architecture



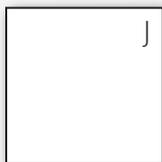
H

Development  
Environment



I

Test Plan



J

Conclusions



K

Appendix

# Appendix

K.1 - List of Source Code and Authors

K.2 - Language Grammar

K.3 - Validation Documentation



<b>File Name</b>	<b>Type</b>	<b>Author</b>
Makefile	Makefile	Dan
roll	script	Dan/Jesse
rollc	script	Dan/Jesse
images	jpg	Lauren
advancedChutesBoardComponent.java	Java	Lauren
Board.java	Java	Jesse
BoardComponent.java	Java	Lauren
BoardImplementation.java	Java	Jesse
Card.java	Java	Jesse
ChutesTile.java	Java	Lauren
Deck.java	Java	Jesse
Dice.java	Java	Jesse
DiceImplementation.java	Java	Jesse
Die.java	Java	Jesse
Driver.java	Java	Jesse
Game.java	Java	Jesse
GlobalVariables.java	Java	Jesse
GridTile.java	Java	Lauren
NextTurnListener.java	Java	Lauren
PerimeterTile.java	Java	Lauren
Piece.java	Java	Lauren
Player.java	Java	Lauren
Players.java	Java	Lauren
PlayersImplementation.java	Java	Jesse
ScrollPane.java	Java	Lauren
SnakeTile.java	Java	Lauren
SorryBoardComponent.java	Java	Lauren
SorryTile.java	Java	Lauren
SpecialBoardComponent.java	Java	Lauren
Tile.java	Java	Jesse
TileLocationReturner.java	Java	Lauren
Window.java	Java	Lauren
roll.l	Flex	Dan/Jesse
advancedChutes.roll	ROLL	Dan/Jesse
ArraySetTest.roll	ROLL	Dan/Jesse
ArrayTest.roll	ROLL	Dan/Jesse
customArrayTest.roll	ROLL	Dan/Jesse
Default.roll	ROLL	Dan/Jesse
Empty.roll	ROLL	Dan/Jesse
Hello.roll	ROLL	Dan/Jesse
IncrementalTest.roll	ROLL	Yipeng
NewHello.roll	ROLL	Dan/Jesse
ScopeTest.roll	ROLL	Yipeng
ScopeTestDeck.roll	ROLL	Yipeng
ScopeTestDice.roll	ROLL	Yipeng
SimpleChutes.roll	ROLL	Dan/Jesse

Sorry.roll	ROLL	John
Statements.roll	ROLL	Dan/Jesse
test3.roll	ROLL	Dan/Jesse
test4.roll	ROLL	Dan/Jesse
test7.roll	ROLL	Dan/Jesse
test8.roll	ROLL	Dan/Jesse
test11.roll	ROLL	Dan/Jesse
my.y	Yacc	Dan/Jesse
roll.y	Yacc	Dan/Jesse

# HIGH LEVEL GRAMMAR

NON-TERMINAL:

PRODUCTION:

program	GAME	IDEN	LEFTBRACE	game_def	RIGHTBRACE
game_def	global_vars	players_block	board_block	dice_deck_block	
board_block	BOARD	LEFTBRACE	board_def	RIGHTBRACE	
	E				
dice_deck_block	dice_block				
	deck_block				
	E				
dice_block	DICE	LEFTBRACE	dice_definition	RIGHTBRACE	
deck_block	DECK	LEFTBRACE	deck_definition	RIGHTBRACE	
players_block	PLAYERS	LEFTBRACE	players_def	RIGHTBRACE	
	E				
global_vars	global_dec	global_vars			
	E				
global_dec	INT	GLOBALIDEN	ASSIGN_OP	rhs	SEMICOLON

# PLAYERS BLOCK GRAMMAR

NON-TERMINAL: PRODUCTION:

players_block	PLAYERS E	LEFTBRACE	players_def	RIGHTBRACE			
player_def	play_decs play_funs						
play_decs	gui_def max_player_ef min_player_def num_pieces_def start_on_def finish_on_def num_players_def						
max_player_def	MAXPLAYERS E	ASSIGN_OP	rhs	SEMICOLON			
min_player_def	MINPLAYERS E	ASSIGN_OP	rhs	SEMICOLON			
num_pieces_def	NUMPIECES E	ASSIGN_OP	rhs	SEMICOLON			
start_on_def	STARTON E	ASSIGN_OP	in_place_array	SEMICOLON			
play_funs	DEFINE E	SETUPPLAYERS	LEFTBRACE	stmt_list	RIGHTBRACE		
gui_def	GUI E	ASSIGN_OP	rhs	SEMICOLON			
finish_on_def	FINISHON E	ASSIGN_OP	in_place_array	SEMICOLON			
num_players_def	NUMPLAYERS E	ASSIGN_OP	rhs	SEMICOLON			

# BOARD BLOCK GRAMMAR

NON-TERMINAL:	PRODUCTION:					
board_block	BOARD E	LEFTBRACE	board_def	RIGHTBRACE		
board_def	tile_def tile_def board_funs E	board_funs				
tile_def	qty qty	tile_list				
tile_list	tile tile	tile_list				
qty	NUMTILES	ASSIGNOP	rhs	SEMICOLON		
board_funs	pre_roll goal_check pre_roll goal_check lands_on_fun pre_roll pre_roll lands_on_fun lands_on_fun goal_check goal_check lands_on_fun pre_roll pre_roll lands_on_fun goal_check	goal_check pre_roll	lands_on_fun goal_check pre_roll goal_check pre_roll lands_on_fun			
tile	MAKE	TILE	LEFTPARENS	tile_args	RIGHTPARENS	SEMICOLON
tile_args	index	nxt	prv	optional		
index	ID	COLON	rhs	COMMA		
nxt	NEXT	COLON	rhs	COMMA		
prv	PREV	COLON	rhs			
acc	COMMA E	ACCESSIBLE	COLON	in_place_array		

lands_on		COMMA	LANDSON	COLON	IDEN					
goal_check		DEFINE	GOALCHECKFUNCTION	LEFTBRACE	stmt_list	RIGHTBRACE				
pre_roll		DEFINE	PREROLLFUNCTION	LEFTBRACE	stmt_list	RIGHTBRACE				
lands_on_fun		FUNCTION	IDEN	ASSIGNOP	DEFINE	LANDSONFUNCTION	LEFTBRACE	stmt_list	RIGHTBRACE	lands_on_fun
		FUNCTION	IDEN	ASSIGNOP	DEFINE	LANDSONFUNCTION	LEFTBRACE	stmt_list	RIGHTBRACE	
optional		acc	lands_on							
		acc								
		lands_on								
		E								



# DECK BLOCK GRAMMAR

NON-TERMINAL:	PRODUCTION:							
deck_block	DECK	LEFTBRACE	deck_definition	RIGHTBRACE				
deck_definition	has_rep	card_defs	roll_funs					
has_rep	HASREPLACEMENT	ASSIGNOP	rhs	SEMICOLON				
card_defs	card_definition	card_defs2						
card_defs2	card_definition	card_defs2						
card_definition	MAKE	CARD	LEFTPARENS	card_args	RIGHTPARENS	SEMICOLON		
card_args	card_val	card_quantity	card_fun					
card_val	VALUE	COLON	rhs					
card_quantity	COMMA	QUANTITY	COLON	rhs				
card_fun	COMMA	ROLL	COLON	IDEN				
roll_funs	named_roll_function	roll_funs						
named_roll_function	FUNCTION	IDEN	ASSIGNOP	DEFINE	ROLLFUNCTION	LEFTBRACE	stmt_list	RIGHTBRACE



expr2	rhs	RELOP	rhs			
	LEFTPARENS	expr	RIGHTPARENS			
	NOT	LEFTPARENS	expr	RIGHTPARENS		
list	int_list					
	PLAYERLIST	LEFTBRACKET	rhs	RIGHTBRACKET	PERIOD	PIECELIST
int_list	in_place_array					
	TILELIST	LEFTBRACKET	rhs	RIGHTBRACKET	PERIOD	ACCESSIBLE
	TILELIST	LEFTBRACKET	rhs	RIGHTBRACKET	PERIOD	NEXT
	TILELIST	LEFTBRACKET	rhs	RIGHTBRACKET	PERIOD	PREV
	STARTON					
	FINISHON					
	ARRAYIDEN					
in_place_array	LEFTBRACE	standard	RIGHTBRACE			
	LEFTBRACE	short_hand	RIGHTBRACE			
standard	standard2					
	E					
standard2	rhs	COMMA	standard2			
	rhs					
short_hand	rhs	TILDA	rhs			
pre_def_var	MAXPLAYERS					
	MINPLAYERS					
	NUMPIECES					
	NUMPLAYERS					
	NUMTILES					
	NEXTTURN					
	GUI					

# STATEMENTS GRAMMAR

NON-TERMINAL:	PRODUCTION:									
stmt_list	stmt	stmt_list								
	E									
stmt	declaration									
	assignment									
	if_stmt									
	for_stmt									
	funcall									
declaration	INT	IDEN	ASSIGN_OP	rhs	SEMICOLON					
	INT	IDEN	SEMICOLON							
	INT	ARRAYIDEN	LEFTBRACKET	rhs	RIGHTBRACKET	SEMICOLON				
	INT	ARRAYIDEN	ASSIGN_OP	in_place_array	SEMICOLON					
assignment	IDEN	ASSIGNOP	rhs	SEMICOLON						
	GLOBALIDEN	ASSIGNOP	rhs	SEMICOLON						
	NEXTTURN	ASSIGNOP	rhs	SEMICOLON						
	ARRAYIDEN	LEFTBRACKET	rhs	RIGHTBRACKET	ASSIGN_OP	rhs	SEMICOLON			
funcall	print									
	winner									
	move_funcall									
	move_reverse_funcall									
	jump_funcall									
	prompt_list_funcall									
	prompt_range_funcall									
	prompt_name_funcall									
winner	DECLAREWINNER	LEFTPARENS	rhs	RIGHTPARENS	SEMICOLON					
	DECLAREWINNER	LEFTPARENS	RIGHTPARENS	SEMICOLON						
prompt	PROMPTLIST	LEFTPARENS	IDEN	COMMA	list	RIGHTPARENS	SEMICOLON			
	PROMPTRANGE	LEFTPARENS	IDEN	COMMA	rhs	COMMA	rhs	RIGHTPARENS	SEMICOLON	
	PROMPTTEXT	LEFTPARENS	IDEN	RIGHTPARENS	SEMICOLON					
print	PRINT	LEFTPARENS	text	RIGHTPARENS	SEMICOLON					
move	MOVE	LEFTPARENS	rhs	COMMA	rhs	COMMA	rhs	RIGHTPARENS	SEMICOLON	
	MOVEREVERSE	LEFTPARENS	rhs	COMMA	rhs	COMMA	rhs	RIGHTPARENS	SEMICOLON	
	JUMP	LEFTPARENS	rhs	COMMA	rhs	COMMA	rhs	RIGHTPARENS	SEMICOLON	
if_stmt	IF	LEFTPARENS	expr	RIGHTPARENS	LEFTBRACE	stmt_list	RIGHTBRACE	else_stmt		
else_stmt	ELSE	IF	LEFTPARENS	expr	RIGHTPARENS	LEFTBRACE	stmt_list	RIGHTBRACE	else_stmt	
	ELSE	LEFTBRACE	stmt_list	RIGHTBRACE						
	E									
for_stmt	FOR	LEFTPARENS	INT	IDEN	COLON	int_list	RIGHTPARENS	LEFTBRACE	stmt_list	RIGHTBRACE
move_funcall	MOVE	LEFTPARENS	rhs	COMMA	rhs	COMMA	rhs	RIGHTPARENS	SEMICOLON	
move_reverse_funcall	MOVEREVERSE	LEFTPARENS	rhs	COMMA	rhs	COMMA	rhs	RIGHTPARENS	SEMICOLON	

jump_funcall	JUMP	LEFTPARENS	rhs	COMMA	rhs	COMMA	rhs	RIGHTPARENS	SEMICOLON
prompt_list_funcall	PROMPTLIST	LEFTPARENS	variable	COMMA	in_place_array	RIGHTPARENS	SEMICOLON		
	PROMPTLIST	LEFTPARENS	variable	COMMA	ARRAYIDEN	RIGHTPARENS	SEMICOLON		
prompt_range_funcall	PROMPTLIST	LEFTPARENS	variable	COMMA	rhs	COMMA	rhs	RIGHTPARENS	SEMICOLON
prompt_name_funcall	PROMPTNAME	LEFTPARENS	player_name	RIGHTPARENS	SEMICOLON				
player_name	PLAYERLIST	LEFTBRACKET	rhs	RIGHTBRACKET	PERIOD	NAME			
variable	IDEN								
	GLOBALIDEN								
	pre_def_var								
	array_var								
array_var	ARRAYIDEN	LEFTBRACKET	rhs	RIGHTBRACKET					
	STARTON	LEFTBRACKET	rhs	RIGHTBRACKET					
	FINISHON	LEFTBRACKET	rhs	RIGHTBRACKET					

# ROLL Validation Documentation Excerpt

## System Integration Test

### Purpose:

This test case is a systems integration test to ensure that the compiler environment is correctly set up and that all components function together correctly. This test case uses the empty ROLL source code to test:

1. functionality of MakeFile
2. the creation of the lexer created by Flex upon compiling the lexer specification roll.l
3. the creation of the parser created by Yacc upon compiling the parser specification my.y. Note that my.y captures each token returned by the lexer and prints the token name to the console. No changes are made to the default ROLL game framework.
4. the creation of the parser created by Yacc upon compiling the parser specification roll.y
5. the creation of the Java class files created by javac upon compiling the java source files that compose the ROLL game framework.
6. the validity of the Java class files generated by the compiler. The class files should be properly integrated with each other and should be able to run without errors.
7. the validity of the compiler executable when run in a memory checking environment, in this case Valgrind

### Test results:

Test subcases	Test description	ROLL source code used in test	Expected result	Actual result	Special notes
Token recognition test	Use the lexer generated by the lexer specification roll.l, in conjunction with the parser generated by the parser specification my.y, to test that the lexer returns the correct tokens in the correct order.	Game Default {}	The lexer should return, in order, the token for the reserved word "Game", the identifier for the name of the game "Default", and a set of opening and closing braces.	The tokens returned in order: GAME IDEN, yylval: Default LEFTBRACE RIGHTBRACE	Passed
Token response test	Use the lexer generated by the lexer specification roll.l, in conjunction with the parser generated by the parser specification roll.y, test that the parser responds to the tokens as expected and modifies the Java game framework as expected.	Game Default {}	The parser should generate concrete Java source files for BoardImplementation, DiceImplementation, PlayersImplementation that are blank other than the class signature, a constructor, and a call to the superclass constructor.	a. Generated code for BoardImplementation.java: public class BoardImplementation extends Board { public BoardImplementation(Player[] players) { super(players); } } b. Generated code for DiceImplementation.java:	Passed

				<pre> public class DiceImplementation extends Dice{     public     DiceImplementat ion(Player[] players){          super(playe rs);     } } </pre> <p>c. Generated code for PlayersImplementation.java:</p> <pre> public class PlayersImplementation extends Players{ } </pre>	
Memory integrity test	Use the lexer generated by the lexer specification roll.l, in conjunction with the parser generated by the parser specification roll.y, and execute the compiler executable game.r under valgrind.	Game Default {}	The output from Valgrind should report that the only memory leaks are those deliberately left by Flex in order to communicate with Yacc.		Passed

### Status and recommendations:

The blank concrete classes were generated as expected. The functionality of the completed game is governed by the abstract classes that are part of the ROLL game framework. No overrides were provided by the compiler.

### Language Features Incremental Tests: Syntax

#### Purpose:

This set of test cases tests the language features described in the Language Reference Manual, under the Syntax section. Features mentioned in other sections of the Language Reference Manual are required to run this set of tests, and they are assumed to operate correctly for the purposes of this test.

The test will validate the functionality of the following features:

1. The set of ASCII characters that are valid for use in variable names
2. Ability of the compiler to properly remove white space in the form of spurious spaces, new lines, and tabs.
3. The proper functioning of in line comments

#### Test results:

Test subcases	Test description	ROLL source code used in test	Expected result	Actual result	Special notes
Whitespace recognition and elimination test	Test that the compiler can recognize and ignore all spurious white space in the form of spaces, new lines, and tabs	IncrementalTest.roll, which is based off of Default.roll.  White space was added around various language constructs.	Compiler should be able to compile even with the additional confusing white space.	The compiler successfully parsed and ignored white space.	Passed.

Comment elimination test	Test that the compiler recognizes // as a cue to ignore the rest of the line.  In all other contexts, specifically, within text strings surrounded by “ ”, the character pair // should not cause the rest of the line to be ignored.	IncrementalTest.roll, which is based off of Default.roll.  // print("This shouldn't print."); print("This // should print.");  print("This should print as well."); // end of line comment	The java program should, upon execution, display:  This // should print. This should print as well.	The program displayed the lines as expected.	Passed.  It is safe to include // within text strings surrounded by “ ”. No escape characters are necessary to suppress the behavior of line commenting.
--------------------------	---	--	--	--	--

### Status and recommendations:

The basic syntax constructs mentioned in the Language Reference Manual are all correctly implemented in the ROLL compiler.

### Language Features Incremental Tests: Primitive Types

#### Purpose:

This set of test cases tests the language features described in the Language Reference Manual, under the Primitive Types section. Features mentioned in other sections of the Language Reference Manual are required to run this set of tests, and they are assumed to operate correctly for the purposes of this test.

The test will validate the functionality of the following features:

1. The integer type and its valid bounds in the ROLL language
2. Text type and the acceptable characters in text
3. Array instantiation using comma separated values
4. Array instantiation using the short hand notation specifying the lower and upper bounds of the array.
5. Array instantiation using comma separated values or short hand notation using any combination of global and local variables as the initial values.
6. Automatic casting of integers to text in print statements.

#### Test results:

Test subcases	Test description	ROLL source code used in test	Expected result	Actual result	Special notes
Integer type test	Tests the upper and lower bounds of allowable integers in the ROLL language. This confirms that the acceptable integer values in ROLL are the same as those in Java.	IncrementalTest.roll, which is based off of Default.roll.  \$myGV1 = -2147483648;  \$myGV2 = 2147483647;	The Java game program should display these two values accurately.	The values are displayed to the console without error.	Passed.
Text type test	Tests that all ASCII characters can be accepted by the ROLL compiler and can be validly used in user defined texts, such as those in print statements.	IncrementalTest.roll, which is based off of Default.roll.  The following print statement is included:  print("All ASCII printable characters: ~!@#\$\$%^&*() +`1234567890-	The Java game should print all the ASCII characters correctly, with the exception of the character “ ”, which indicates the end of a text string.	The character \ is illegal in ROLL text strings as well. Passing on \ to the Java compiler within a text string cues the Java compiler to anticipate an escape character, which leads to unpredictable behavior.	Passed.  All ASCII characters are allowable in ROLL text with the exception of “ ” and \

Regular array instantiation test	Tests that regular array instantiation using comma separated values behaves as expected.	<pre>={}[[]; '&lt;?;/'); IncrementalTest.roll, which is based off of Default.roll.  The following instantiation of the StartOn array is included:  StartOn = {1,2,3,4,5,6};  The sixth value of the array is printed to console to test that the array was accurately instantiated.</pre>	The sixth value in the array, 6, should be printed correctly.	The console prints the expected result:  StartOn[5] = 6	Passed.
Shorthand array instantiation test	Tests how big integer arrays can safely be in the ROLL language.	<pre>IncrementalTest.roll, which is based off of Default.roll.  The following instantiation of the StartOn array is included:  StartOn = {-1000000 ~ 1000000};  The last element of the array is printed to the console:  print("StartOn[2000000] = "   StartOn[2000000]);</pre>	The program should safely display the value of the last item in the array, 1000000.	The program prints 1000000 correctly.	<p>Passed.</p> <p>Forcing the compiler to generate an array of size 2147483647, the largest integer value in Java, failed.</p> <p>The compiler is guaranteed to behave accurately for arrays up to size 2000001.</p>
Regular and shorthand array instantiation using variables as values	Tests that regular array instantiation can also be done passing in local variables, global variables, and values from other arrays.	<pre>int #myArray3 = {800, 900};  int #myArray4 = { #myArray3[0], #myArray3[1] };  int #myArray5 = { #myArray3[0] ~ #myArray3[1] };  print("#myArray4[0] = "   #myArray4[0]);  print("#myArray4[1] = "   #myArray4[1]);  print("#myArray5[1] = "   #myArray5[1]);  print("#myArray5[98] = "   #myArray5[98]);</pre>	The compiler should be able to handle instantiating and initializing values for arrays using any combination of local variables, global variables, and integers stored in arrays.	The compiler instantiated and initialized values correctly.	Passed.
Integer to text automatic casting test	Test the functionality of automatic casting from integers to text within print() statements.	No separate test source code is necessary.	The program should be able to display integers as if it were strings.	The automatic casting of integers to text functions correctly.	Passed.

### Status and recommendations:

The above series of tests establishes the bounds and limitations of primitive types in the ROLL language. It is established that the ROLL program can accurately handle arrays of up to size 2,000,001, and that the ASCII characters ‘\’ and “ are forbidden within text types. No escape sequences are incorporated into the language design. These limitations are established to not hinder the ability of a ROLL programmer to design games in any significant way.

## Language Features Incremental Tests: User-Initialized Variables

### Purpose:

This set of test cases tests the language features described in the Language Reference Manual, under the User-Initialized Variables section. Features mentioned in other sections of the Language Reference Manual are required to run this set of tests, and they are assumed to operate correctly for the purposes of this test.

The test will validate the functionality of user-defined global integer variables.

### Test results:

Test subcases	Test description	ROLL source code used in test	Expected result	Actual result	Special notes
User-defined local integer variable test	Test that user-defined local integer variables behave correctly.	This functionality is tested simultaneously in the section about for-each loops.	Local variables should be properly instantiated, read from, and written to.	The local variables are accurately instantiated, displayed to the console, and reassigned a value.	Passed.
User-defined global integer variable test	Test that user-defined global variables function correctly.	IncrementalTest.roll, which is based off of Default.roll.  The following lines were added to test the functionality of instantiating, reading from, and reassigning values to a user-defined global variable:  <code>\$myGVChange = 600;</code>  <code>print("\$myGVChange = "   \$myGVChange);</code>  <code>\$myGVChange = 700;</code>  <code>print("reassignment: \$myGVChange = "   \$myGVChange);</code>	Global variables should be instantiated properly, read from, and written to.	The global variable was correctly instantiated, displayed to the console, and reassigned a value.	Passed.
User-defined local integer array test	Test that user-defined local integer arrays behave correctly.	The code used in testing array instantiation in the previous section serves as an adequate test case for this functionality.	User-defined local integer arrays should be properly instantiated, read from, and written to.	The local integer array was correctly instantiated, its values were correctly displayed to the console and reassigned values.	Passed.

### Status and recommendations:

The functionality of global variables reassignment originally did not work and was reported as a bug in the compiler. Its functionality is now verified to be complete.

With the completion of this test suite, it is verified that ROLL programmers have access to three types custom-defined variables or arrays: 1. local integer variables, 2. globally-visible integer variables, 3. local integer arrays. For instantiating arrays, there are two forms of syntax for initializing values, and variables can be used as the arguments for initializing arrays.