

Utah State University

DigitalCommons@USU

All Graduate Plan B and other Reports

Graduate Studies

5-2013

Processing and Manipulation of Data Collected from the Educational On-Line Game Refraction

Xiaotian Dai
Utah State University

Follow this and additional works at: <https://digitalcommons.usu.edu/gradreports>



Part of the [Mathematics Commons](#)

Recommended Citation

Dai, Xiaotian, "Processing and Manipulation of Data Collected from the Educational On-Line Game Refraction" (2013). *All Graduate Plan B and other Reports*. 320.

<https://digitalcommons.usu.edu/gradreports/320>

This Report is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Plan B and other Reports by an authorized administrator of DigitalCommons@USU. For more information, please contact digitalcommons@usu.edu.



PROCESSING AND MANIPULATION OF DATA COLLECTED FROM THE
EDUCATIONAL ON-LINE GAME REFRACTION

by

Xiaotian Dai

A report submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Statistics

Approved:

Dr. Jürgen Symanzik
Major Professor

Dr. Daniel C. Coster
Committee Member

Dr. H. Taylor Martin
Committee Member

UTAH STATE UNIVERSITY
Logan, Utah

2013

ABSTRACT

Processing and Manipulation of Data Collected from the Educational On-line Game
Refraction

by

Xiaotian Dai, Master of Science
Utah State University, 2013

Major Professor: Dr. Jürgen Symanzik
Department: Mathematics and Statistics

A team of students, artists, and researchers at the Center for Game Science at the University of Washington are trying to create video games that can discover optimal pathways for learning. They have focused so far on early mathematics education, including topics such as fractions and algebra, which are some of the main bottlenecks preventing students from pursuing a career in science. As a result, the educational on-line game “Refraction” was created, which is aimed at students who start learning fraction computations. When the students are playing the game online, all the data and information, such as mouse movements and mouse clicks, are stored in datasets of different formats. In this MS report, we will develop functions in the R software environment that will allow other researchers to easily process and manipulate the data generated from this game for future statistical analyses.

(70 pages)

CONTENTS

	Page
ABSTRACT	ii
LIST OF FIGURES	v
1 INTRODUCTION	1
1.1 The Refraction game	1
1.2 JSON data	3
1.3 XML data	4
1.4 Data manipulation using R	4
1.5 Objectives	5
2 REORGANIZING REFRACTION GAME DATA FROM THE ORIGINAL JSON FORMAT INTO R DATA FRAMES	7
2.1 Structure of JSON objects	7
2.2 Programming in R	10
2.3 Input JSON data	10
2.4 Data and information in the JSON objects	13
2.4.1 General information frame	13
2.4.2 Bin_pieces data frame	14
2.4.3 Win_pieces data frame	15
2.4.4 Board_pieces data frame	16
2.4.5 Detailed information of pieces	17
3 IMPLEMENTATION OF A JSON TREE IN R	20
3.1 Introduction	20
3.2 The basic JSON tree	22
3.3 Union and intersection of JSON trees	25
4 REORGANIZING REFRACTION GAME DATA FROM THE ORIGINAL XML FORMAT INTO R DATA FRAMES	28
4.1 Structure of XML objects	28
4.2 Programming in R	30
4.3 Input XML data	31
4.4 Data and information in the XML objects	33
4.4.1 Attributes data frame	34
4.4.2 Laser objects data frames	35
4.4.3 Other keys in the XML objects	37

5 CONCLUSIONS	39
5.1 Summary of research	39
5.2 Future research	39
APPENDICES	43
APPENDIX A R CODE	44
A.1 R code for Chapter 2	44
A.2 R code for Chapter 3	52
A.3 R code for Chapter 4	60

LIST OF FIGURES

Figure		Page
1	Screen shot of the web page of Refraction on September 28th, 2013 http://centerforgamescience.org/portfolio/refraction/ . . .	2
2	JSON object showing level “43” of the Refraction game	8
3	Screen shot of level “43” of the Refraction game	9
4	JSON object showing level “43” of the Refraction game	21
5	JSON tree structure plot for level “43” of the Refraction game	24
6	Union tree of all levels of the Refraction game	26
7	Intersection tree of all levels of the Refraction game	27
8	XML object showing level “111” of the Refraction game	29

CHAPTER 1


INTRODUCTION

1.1 The Refraction game

In an effort to relieve the crisis in Science, Technology, Engineering, and Mathematics (STEM) education, a team of graduate students, artists, and researchers at the Center for Game Science (CGS) at the University of Washington have created video games focused on scientific discovery, discovering optimal learning pathways for STEM education, cognitive skill training, and games that explore collective over individual intelligence (Center for Game Science *CGS*, 2012). Their games can be accessed at <http://centerforgamescience.org/games/>. We are working with data from their Refraction game in this MS research project.

The Refraction game (see Figure 1) was designed for students who are learning early mathematics topics, especially fraction computations. Currently, there are many competing theories how best to teach these subjects, and a lack of experimental data to evaluate these teaching methods prevents the development of effective learning tools (Center for Game Science *CGS*, 2012). The rising popularity of the educational game Refraction may provide us some useful information about the optimal educational pathways.

Refraction is developed on a model of fraction understanding centered on splitting or equal partitioning idea. The objective of this game is to free animals that are trapped in spaceships. To release the animals, the spaceships require different fractions of power that are supplied by lasers (such as $1/2$, $1/3$, $1/6$, or $1/9$ of a full laser beam). A full laser beam (with power 1) originally comes out of a laser source. Students need to bend and split lasers by using the benders and splitters supplied.




Center for Game Science

[GAMES](#) [PEOPLE](#) [RESEARCH](#)

Refraction

Refraction focuses on teaching fractions and discovering optimal learning pathways for math education.

[previous / next](#)



Overview

Refraction is a game in which you bend and split lasers to free animals trapped in space.

[Play Refraction!](#)

Fig. 1: Screen shot of the web page of Refraction on September 28th, 2013
<http://centerforgamescience.org/portfolio/refraction/>

Usually, students apply combinations of $1/2$ and $1/3$ splitters to split the laser and use the benders to change the direction of the laser beam. When the required fractions of laser beams hit on the spaceships, the trapped animals will be freed and this game level is successfully passed. The splitters are the primary mathematical tool students have (Aghababyan et al., 2013). How to split the laser correctly involves simple fraction computations.

The educational games created by CGS have already become a hit in the educational community. Refraction won the Grand Prize in the Disney Learning Challenge at SIGGRAPH 2010 and Best Work in the Primary School Category at NHK Japan 2011 (Japan Broadcasting Corporation *NHK*, 2011). More and more players are visiting the Refraction game at <http://centerforgamescience.org/portfolio/refraction/>. Figure 1 shows the starting web page of the Refraction game. All of the user interactions in this game are stored, including mouse movements and mouse

clicks. Internally, these user actions are translated into game pieces placed or moved on the game board, completion of a game level, etc. We have access to some of the stored Refraction game data. These datasets are in XML and JSON formats. We will discuss how to process and manipulate these datasets in the R software environment (R Development Core Team, 2012) in this MS report.

In fact, the creators of the Refraction game have already published several research papers with focus on computational education based on the stored game data. Andersen, Liu, Snider, Szeto, Cooper and Popović (2011) argue that secondary objectives that support the primary goal of the game are consistently useful, while secondary objectives that do not support the main goal require extensive testing to avoid negative consequences. For instance, as a result, the bonus coins in the Refraction game can distract from the primary goal and cause many players to play for less time. Andersen, Liu, Snider, Szeto and Popović (2011) discuss whether aesthetic improvements such as music, sound effects, and animations can be useful for attracting and retaining players. Their research revealed that music and sound effects had little or no effect on player retention during the game, while animations caused users to play more.

1.2 JSON data

JavaScript Object Notation (JSON) is a programming language model data interchange format (Crockford, 2006). It is easy for humans to read and write. It is also easy for computers to parse and generate because JSON is a natural presentation of data. A JSON object is a collection of key and value pairs. The keys are strings and the types of values presented in JSON can be strings, numbers, booleans, object, arrays, or even NULL. A colon separates the keys from the values, and a comma separates the pairs. The pairs of keys and values are wrapped in curly braces and the

arrays are wrapped in square brackets.

The JSON format is often used for serializing and transmitting structured data over a network connection. It is used primarily to transmit data between a server and web application, serving as an alternative to XML (Crockford, 2006). There exist JSON libraries or built-in JSON support for many programming languages and systems. The JSON libraries built in R will be discussed in Chapter 2.

1.3 XML data

Extensible Markup Language (XML) describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them (Bray et al., 1997). XML files are in a format that is both human-readable and machine-readable. XML files are created to store and transport data and information. The design goals of XML emphasize simplicity, generality, and usability over the Internet (Bray et al., 1997). Similarly to JSON, the XML files have a basic format of key and value pairs and they are designed to be self-descriptive with information wrapped in tags. However, XML is designed to store data rather than displaying data. Hence, functions are written in R to process and display data stored in the XML files. More details will be discussed in Chapter 4.

1.4 Data manipulation using R

The first and often the toughest or most time-consuming task in an analytical environment for a new project is getting the data loaded into the analytical software. In addition, obtaining and storing data from various types of databases requires different data types. Fortunately, R has different internal data types such as lists, arrays, and data frames and this feature offers multiple options to efficiently handle the datasets. In many cases, information makes more sense to humans when it is presented in a

rectangular data format with rows for records and columns for variables. Many statistical analyses, both model-based and graphical, are based on data frames. However, the JSON and XML datasets for the Refraction game cannot easily be stored in data frames directly, even with the help of the existing R packages. We will discuss in this MS report how to reorganize the Refraction game data from the original formats into R data frames.

After the data are reorganized into data frames, it is also helpful to determine how the datasets are structured. The structure of a JSON object is a typical example of a hierarchical data structure. Tree representations can be useful for summarizing the structures of hierarchical data (Al-Khalifa et al., 2002). The tree representation of the JSON data structure can be referred to as an implementation of the so-called “JSON tree”. Some researches are devoted to the plotting of “JSON tree” in other programming languages, such as jQuery4u.com (2013). So far, however, “JSON tree” was not implemented in R.

The R language provides a rich environment for working with data, especially data to be used for statistical modeling or graphics (Spector, 2008). In fact, there are many useful features in R, for instances, built-in functions, vectors or matrices manipulations, and so on. So far, there are more than 4700 available packages in R software environment (<http://cran.us.r-project.org>). R has become an essential computer tool for many statistical computing projects. It continuously inspires researchers to come up with new and exciting ways of using it.

1.5 Objectives

The goal of this research is to process and manipulate data from the Refraction game that are stored in different external JSON and XML files. The data and information, once read into R, can be used for future research. As such, the objectives of

this MS research project are:

1. Reorganizing Refraction game data from the original JSON format into R data frames (Chapter 2).
2. Implementation of a JSON tree in R (Chapter 3).
3. Reorganizing Refraction game data from the original XML format into R data frames (Chapter 4).

The three objectives listed above will be accomplished using the R programming language. This can be the start of future research about the educational Refraction game.

Overall, for many researchers interested in the educational Refraction game, the R functions developed in this MS research project should make it easier for other researchers to process and manipulate data from this game for their analyses.

CHAPTER 2

REORGANIZING REFRACTION GAME DATA FROM THE ORIGINAL JSON FORMAT INTO R DATA FRAMES

2.1 Structure of JSON objects

The first goal of this research is to read in the Refraction game data stored in a JSON file and reorganize the data into a meaningful format for future processing in R. It is important to understand the structure of a JSON data file.

A JSON object is a collection of key and value pairs. A colon separates the keys from the values, and a comma separates the pairs. The keys are strings and the types of value presented in JSON can be strings, numbers, booleans, object, arrays, or even NULL (Crockford, 2006). The keys are wrapped in quote marks and they can be understood as the names of the variables. The data wrapped in curly braces or square brackets after a colon represent lower level variables included in the higher level variable, which is the key string in front of the colon. Generally, JSON is a human-readable format and a natural presentation of data.

Figure 2 shows the first object in the JSON file of the Refraction game data, representing game level “43”, and Figure 3 shows a screenshot of the game board of this level. The first pair of this object has a variable named “concepts” and an empty value. There are five other keys at the first level of the object, which are “game_id”, “bin_pieces”, “qid”, “win_pieces”, and “board_pieces”. More details about these keys will be discussed in Section 2.4.

```

"43" : {
  "concepts" : [],
  "game_id" : 11,
  "bin_pieces" : {
    "bender" : [{
      "id" : 2
    }
  ]
},
"qid" : 43,
"win_pieces" : {
  "bender" : [2]
},
"board_pieces" : {
  "source" : [{
    "id" : 0,
    "value" : {
      "denom" : 1,
      "num" : 1
    }
  }
],
  "target" : [{
    "id" : 1,
    "value" : {
      "denom" : 1,
      "num" : 1
    }
  }
],
}
}

```

Fig. 2: JSON object showing level “43” of the Refraction game

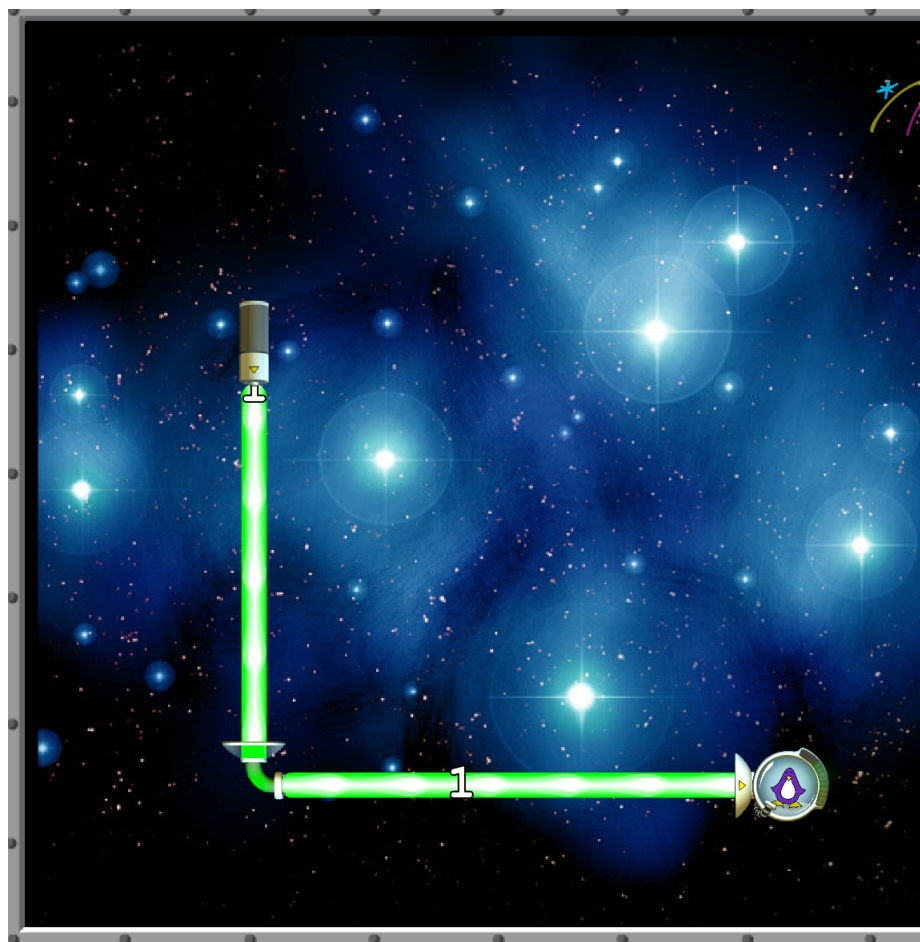


Fig. 3: Screen shot of level “43” of the Refraction game

2.2 Programming in R

R has different data types, such as lists, arrays, and data frames. These data types provide multiple options to handle the datasets. In order to manipulate the Refraction data file of JSON format, the functions described in the following sections will read in the JSON dataset and reorganize the data from its original format into R data frames. The names of the R functions are listed below:

1. **loadJSON**
2. **JSON_general**
3. **JSON_bin_pieces**
4. **JSON_win_pieces**
5. **JSON_board_pieces**
6. **pieces**

More details about these functions will be discussed in the following sections.

2.3 Input JSON data

There exist JSON libraries or built-in JSON support for many programming languages and systems. Since R is an open-source software, two packages have been contributed to the Comprehensive R Archive Network (CRAN) that contain built-in functions for data of JSON format, which are named *rjson* (Couture-Beil, 2012) and *RJSONIO* (Temple Lang, 2011). The functions included in the *RJSONIO* package are built on the C programming language, which can significantly speed up the processing

of the data. More importantly, *RJSONIO* can analyze large collections of JSON objects without unpacking the whole dataset, which is a useful feature for handling the datasets of the Refraction game. Hence, we will use the parser provided by the *RJSONIO* package to input JSON files and store the data in R as a list data type. In R, a list is a generic data structure containing other objects and the objects can be of different types or lengths. This is an essential way to store the JSON data. The first function listed above, **loadJSON**, is written to read JSON data into R. The following output is the R list produced by the **loadJSON** function based on the first JSON object (Figure 2).

```
> data = loadJSON("refraction.json")
```

```
> data[[1]]
```

```
$concepts
```

```
list()
```

```
$game_id
```

```
[1] 11
```

```
$bin_pieces
```

```
$bin_pieces$bender
```

```
$bin_pieces$bender[[1]]
```

```
$bin_pieces$bender[[1]]$id
```

```
[1] 2
```

```
$qid
```

```
[1] 43
```

```
$win_pieces
```

```
$win_pieces$bender
```

```
[1] 2
```

```
$board_pieces
```

```
$board_pieces$source
```

```
$board_pieces$source[[1]]
```

```
$board_pieces$source[[1]]$id
```

```
[1] 0
```

```
$board_pieces$source[[1]]$value
```

```
$board_pieces$source[[1]]$value$denom
```

```
[1] 1
```

```
$board_pieces$source[[1]]$value$num
```

```
[1] 1
```

```
$board_pieces$target
```

```
$board_pieces$target[[1]]
```

```
$board_pieces$target[[1]]$id
```

```
[1] 1
```

```
$board_pieces$target[[1]]$value
```

```
$board_pieces$target[[1]]$value$denom
```

```
[1] 1
```

```
$board_pieces$target[[1]]$value$num
```

[1] 1

2.4 Data and information in the JSON objects

After the JSON objects are read into R, we want to extract data and information of particular interest. There are 147 JSON objects contained in the Refraction game data file. Each JSON object is labeled by a unique “qid” value, which is an identifier for the levels of the Refraction game. The following sections will describe the R functions in detail.

2.4.1 General information frame

The keys “concepts”, “game_id”, and “qid” have a value directly after the colons and they are the general information or identifiers for one unique level of the Refraction game. The function called **JSON_general** has the purpose to extract values of these three keys for all the objects in the data file. The data frame *general_frame* is generated from the general information of the JSON objects. The first six elements of the data frame are shown below. If a JSON object has more than one value for the key “concepts”, all of the “concepts” values for an object will be stored in the same cell of the data frame and separated by commas, such as the sixth row of the data frame below.

```
> general_frame = JSON_general(data)
> head(general_frame)

  concepts game_id qid
1                11  43
2                11  44
```

3		11	46
4		11	47
5		11	48
6	p1, pu	11	50

2.4.2 Bin_pieces data frame

In order to successfully finish one level of the Refraction game, the player needs to choose appropriate pieces from the available “bin_pieces” and put them in the right places of the game board, so that the light can hit right on the target. The available “bin_pieces” are “bender”, “splitter”, “combiner”, and “converter”. The key “bin_pieces” contains information of all the available pieces for the player in one level. Of particular interest is how many “bin_pieces” there are and what the available pieces are. The function called **JSON_bin_pieces** detects names of all the available pieces and returns a data frame which contains the number of available pieces for each game level.

```
> bin_frame = JSON_bin_pieces(data)
> head(bin_frame)
```

	qid	bender	splitter	combiner	converter
2	43	1	0	0	0
3	44	4	0	0	0
4	46	2	0	0	0
5	47	3	0	0	0
6	48	3	0	0	0
7	50	1	2	0	0

As shown in the data frame above, there are four kinds of “bin_pieces” that exist in the 147 levels of the game. In the first “qid” level, there is only one “bender” available for the player, as shown in Figure 2. The **JSON_bin_pieces** function only

counts the number of the “bin_pieces” in each object. Furthermore, the function **pieces** will work on the detailed information of the pieces, such as the “id” of a single piece.

2.4.3 Win_pieces data frame

As mentioned in the last section, the player needs to choose appropriate “bin_pieces” and put them in the right places of the game board, so that the light can hit right on the target. The information of the appropriate “bin_pieces” is stored under the key “win_pieces”. More importantly, the pieces should be organized correctly to pass the game level and the order of the pieces does matter. The function **JSON_win_pieces** returns a data frame which contains the “win_pieces” for each “qid” level and their associated “id” and order.

```
> win_frame = JSON_win_pieces(data)
> win_frame[(win_frame$qid >= 43) & (win_frame$qid <= 50), ]
```

	qid	pieces	id	order
2	43	bender	2	1
3	44	bender	3	1
4	46	bender	2	1
5	46	bender	3	2
6	47	bender	2	1
7	47	bender	3	2
8	47	bender	4	3
9	48	bender	2	1
10	48	bender	3	2
11	48	bender	4	3
12	50	splitter	6	1

In the data frame above, there is information of “win_pieces” for the first six levels of the game. For the “qid” level 46, the player needs to choose two benders from the

“bin_pieces” to win this level and the bender with “id” number 2 should be put on the right spot prior to choosing the bender with “id” number 3.

2.4.4 Board_pieces data frame

In contrast to “bin_pieces”, which are the available pieces for the player, the “board_pieces” refer to the pieces that are already placed in their spots of the game board. The “board_pieces” are essential and can not be removed by the player, such as light “source” and “target”. In different levels, the designers of the game may add more “board_pieces” in order to increase the difficulty and attraction of the game. For instance, the “board_pieces” “blocker” is an object that the light can not pass through and hence, the player needs to modify the path of the light in case the light will be blocked by the “blocker”. Since the “board_pieces” for each level of the game are of particular interest, the function called **JSON_board_pieces** detects names of all the pieces under the key “board_pieces” and returns a data frame which contains the information about “board_pieces”.

```
> board_frame = JSON_board_pieces(data)
> head(board_frame)
```

	qid	source	target	blocker	multitarget
2	43	1	1	0	0
3	44	1	1	0	0
4	46	1	1	0	0
5	47	1	1	4	0
6	48	1	1	22	0
7	50	1	3	5	0

As shown in the data frame above, there are four kinds of “board_pieces” in the 147 levels of the game. In the first “qid” level, there is only one source and one target,

as can be seen in the JSON object in Figure 2.

2.4.5 Detailed information of pieces

Each piece under “bin_pieces” and “board_pieces” has more detailed information such as “id”. For instance, the piece “source” in the first JSON object shown in Figure 2 has two keys under it, which are “id” and “value”, and “value” has another two keys (“denom” and “num”) mapped onto it.

In order to extract all the useful information, the function `pieces` gets all the keys and values for each piece and presents the data in a data frame. The function takes two arguments: the JSON data and the name of the pieces of interest. For example, if we are interested in the detailed information about “source”, then we call the function as follows:

```
> piece_frame1 = pieces(data, "source")
> piece_frame1[(piece_frame1$qid >= 43) & (piece_frame1$qid <= 50), ]

  qid id denom_value num_value
1  43  0           1         1
2  44  0           1         1
3  46  0           1         1
4  47  0           1         1
5  48  0           1         1
6  50  0           1         1

> piece_frame2 = pieces(data, "target")
> piece_frame2[(piece_frame2$qid >= 43) & (piece_frame2$qid <= 50), ]

  qid id denom_value num_value
1  43  1           1         1
2  44  1           1         1
3  46  1           1         1
```

4	47	1	1	1
5	48	1	1	1
6	50	1	3	1
7	50	2	3	1
8	50	3	3	1

The piece “source” and “target” are both “board_pieces”. If we are interested in some of the pieces from “bin_pieces”, then we call the same function and obtain other data frames.

```
> piece_frame3 = pieces(data, "bender")
> piece_frame3[(piece_frame3$qid >= 43) & (piece_frame3$qid <= 50), ]
```

```
  qid id
1  43  2
2  44  2
3  44  3
4  44  4
5  44  5
6  46  2
7  46  3
8  47  2
9  47  3
10 47  4
11 48  2
12 48  3
13 48  4
14 50  4
```

```
> piece_frame4 = pieces(data, "splitter")
> piece_frame4[(piece_frame4$qid >= 43) & (piece_frame4$qid <= 50), ]
```



```
qid id value
1 50 5     2
2 50 6     3
```

In all the returned data frames above, “qid” serves as the primary identifier for the JSON objects and it always takes the first column of the data frames. The function **pieces** can also be applied to all the other “board_pieces” and “bin_pieces”. Overall, all the data contained in the JSON data file can be extracted and prepared for future analyses in R.

CHAPTER 3

IMPLEMENTATION OF A JSON TREE IN R

3.1 Introduction

In general, JSON is a human-readable data interchange format used for representing simple data structures (Crockford, 2006). Especially in the JSON format used for the Refraction game data, the JSON objects are easy to understand and quickly mastered. However, there may be some other JSON data files which are not organized so well. Hence, it is beneficial to obtain a better understanding of the structure of a JSON object.

Wilhelm (2009) mentions that tree representations can be useful for presenting hierarchical data on the screen and uses JSON objects as a typical example of hierarchical data. In fact, primitive tree-structured relationships are parent-child and ancestor-descendant, which can be referred to as hierarchical relationships. Finding all occurrences of these relationships in a dataset is a core operation for the tree representation of the data structure (Al-Khalifa et al., 2002). Wang (2011) discusses how the JSON structure can be abstracted as a multi-branch tree and how the JSON objects can be parsed by recursive traversal of its tree parsing. Overall, the JSON objects can be summarized as a tree structure. There already exist some free JSON tree viewers online, such as jQuery4u.com (2013). Currently, all of them are built in programming languages other than R, such as JavaScript. The research goal presented in this chapter is creating a tree plot in R to make a simple visual presentation of the structure of a JSON object. The first object in the JSON file from the Refraction game, shown in Figure 4, will be used as an example again to document the newly developed R functions.

```

"43" : {
  "concepts" : [],
  "game_id" : 11,
  "bin_pieces" : {
    "bender" : [{
      "id" : 2
    }
  ]
},
"qid" : 43,
"win_pieces" : {
  "bender" : [2]
},
"board_pieces" : {
  "source" : [{
    "id" : 0,
    "value" : {
      "denom" : 1,
      "num" : 1
    }
  }
],
  "target" : [{
    "id" : 1,
    "value" : {
      "denom" : 1,
      "num" : 1
    }
  }
],
}
}

```

Fig. 4: JSON object showing level “43” of the Refraction game

3.2 The basic JSON tree

As mentioned in the previous section, the structure of a JSON object can be described as a multi-branch tree, in which there are several nodes on each level of the tree object. The nodes may be terminated or they may contain more child nodes. In the JSON object shown in Figure 4, there are six keys at the first level. They are “concepts”, “game_id”, “bin_pieces”, “qid”, “win_pieces”, and “board_pieces”. Three of the keys have values and three other keys contain other key/value pairs. Each of the child nodes is only referred to by its parent node, and none of the child nodes connects with the root node directly. Hence, a tree data structure can be defined recursively as a collection of nodes starting from the root node. In order to sketch the tree structure, the function needs to find all the nodes of the tree.

The tree plot can be organized similarly to the original JSON object, in which the keys of the same level will be placed around one vertical line and the child nodes will be placed on the lower right side of its parent node.

A data frame is used to store the information of this structure temporarily and the function **JSONtree** will take in a JSON object and return such a data frame. When we test this function on the object shown in Figure 4, we will get the following data frame, called matrix:

```
> matrix = JSONtree(data[[1]])
> matrix
```

	[,1]	[,2]	[,3]	[,4]
[1,]	"concepts"	NA	NA	NA
[2,]	"game_id"	NA	NA	NA
[3,]	"bin_pieces"	NA	NA	NA
[4,]	NA	"bender"	NA	NA
[5,]	NA	NA	"id"	NA
[6,]	"qid"	NA	NA	NA

```

[7,] "win_pieces" NA      NA      NA
[8,] NA          "bender" NA      NA
[9,] "board_pieces" NA      NA      NA
[10,] NA         "source" NA      NA
[11,] NA         NA       "id"   NA
[12,] NA         NA       "value" NA
[13,] NA         NA       NA       "denom"
[14,] NA         NA       NA       "num"
[15,] NA         "target" NA      NA
[16,] NA         NA       "id"   NA
[17,] NA         NA       "value" NA
[18,] NA         NA       NA       "denom"
[19,] NA         NA       NA       "num"
attr(,"class")
[1] "JSONtree"

```

This data frame contains the location information of the keys to be placed in a plot. We just need some lines to connect the nodes and sketch a “tree”. The function **print.JSONtree** can finish this job and produce a sketch of the JSON tree. This function is built based on the S3 object-oriented programming models and the output matrix above with a class “JSONtree” can be inherited by the generic function **print()**, which means users of the function can call the function **print()** directly instead of **print.JSONtree()**.

As shown in Figure 5, the sketch of the JSON object is a simple tree-like visual presentation of the structure. The **JSONtree** function is designed to be a general purpose function that can be used for managing various datasets stored in JSON format. However, it is not an actual “graph”. The JSON tree may be implemented as a real graph using **plot()** methods in the future.

```
> print(matrix)
```

```
JSON
|
|  - - - - concepts
|  - - - - game_id
|  - - - - bin_pieces
|          |
|          |  - - - - bender
|          |          |
|          |          |  - - - - id
|  - - - - qid
|  - - - - win_pieces
|          |
|          |  - - - - bender
|  - - - - board_pieces
|          |
|          |  - - - - source
|          |          |
|          |          |  - - - - id
|          |          |  - - - - value
|          |          |          |
|          |          |          |  - - - - denom
|          |          |          |  - - - - num
|          |          |  - - - - target
|          |          |          |
|          |          |          |  - - - - id
|          |          |          |  - - - - value
|          |          |          |          |
|          |          |          |          |  - - - - denom
|          |          |          |          |  - - - - num
```

Fig. 5: JSON tree structure plot for level “43” of the Refraction game

3.3 Union and intersection of JSON trees

The function **JSONtree** is designed to sketch the tree structure of every single JSON object. There are 174 JSON objects in the JSON data file of the Refraction game. Instead of calling the **JSONtree** function 174 times, we need another function to find the union of the 174 JSON trees. We can easily see which keys are contained in the whole dataset based on the union JSON tree. The function **uniontree** returns the union of the 174 JSON trees. The different child nodes in different JSON objects are grouped under their corresponding parent nodes.

The union of all the JSON trees in Figure 6 provides us with some further information about which keys are contained in this JSON dataset. In fact, as mentioned in Section 2.1, there are six keys at the first level of all the objects, which are “concepts”, “game_id”, “bin_pieces”, “qid”, “win_pieces”, and “board_pieces”. Hence, the union tree also has these six keys at the first level and other child nodes in the dataset that do not appear in the JSON tree for level “43”, shown in Figure 5.

It is also of interest to identify the keys that are included in every JSON object. In order to achieve this goal, the function **intersecttree** returns the intersection of the 174 JSON trees. A key is included in this intersection tree only if this key is contained in every JSON object.

The intersection tree in Figure 7 shows the joint structure of the 174 JSON objects. Indeed, the six keys mentioned above are used in each level. Except for the “source” branch from “board_pieces”, no other keys from the union of all trees (see Figure 6) appears in the intersection of all trees (see Figure 7).


```

> intersecttree(data)

JSON
|
|  - - - - concepts
|  - - - - game_id
|  - - - - bin_pieces
|  - - - - qid
|  - - - - win_pieces
|  - - - - board_pieces
|          |
|          |  - - - - source
|          |  |
|          |  |  - - - - id
|          |  |  - - - - value
|          |  |  |
|          |  |  |  - - - - denom
|          |  |  |  - - - - num

```

Fig. 7: Intersection tree of all levels of the Refraction game

The tree representations of the JSON objects can easily be extended to other JSON files. The same approach can also be applied to other tree-structured data formats, or, say, data formats with a hierarchical database model, such as the XML format.

CHAPTER 4

REORGANIZING REFRACTION GAME DATA FROM THE ORIGINAL XML FORMAT INTO R DATA FRAMES

4.1 Structure of XML objects

In addition to the JSON file discussed in Chapter 2, Refraction game data are stored in another file with XML format. Both datasets contain similar information of each level of the Refraction game. However, there are differences between them since they are generated with different details. Hence, we also want to reorganize the data from the original XML format into data frames for future processing in R.

By definition, an XML document is a string of characters. The characters making up an XML document are divided into markup and content, which may be distinguished by the application of simple syntactic rules (Bray et al., 1997). Generally, strings that constitute markup begin with the character “<” and end with a “>”. These strings are called tags. There are start-tags and end-tags. The strings in the tags are the “keys”, i.e., they can be referred to as the names of the variables. The end-tags start with “/” and the strings in the end-tags must match those in the start-tags. Between the start-tag and the end-tag, there can be one value or other pairs of keys and values. If an XML object consists of key/value pairs within a start-tag, the contents within the start-tag are describing the attributes of the XML object.

```

<pipesLevel id="TheBeginning" name="The beginning" qid="111" version="1.0">
  <useSpecialLevel>1</useSpecialLevel>
  <enableMagnify>0</enableMagnify>
  <partitionLasers>1</partitionLasers>
  <partitionUnit>
    <num>1</num>
    <denom>1</denom>
  </partitionUnit>
  <laserObject>
    <className>FGLaserSink</className>
    <inputDirection>WEST</inputDirection>
    <index>88</index>
    <targetValue>
      <num>1</num>
      <denom>1</denom>
    </targetValue>
  </laserObject>
  <laserObject>
    <className>FGLaserSource</className>
    <outputDirection>SOUTH</outputDirection>
    <index>32</index>
  </laserObject>
  <laserObject>
    <className>FGLaserDivider</className>
    <inputDirection>NORTH</inputDirection>
    <outputDirection>EAST</outputDirection>
    <index>82</index>
  </laserObject>
  <concept>
    <name>bend</name>
    <level>1</level>
  </concept>
  <numMoves>1</numMoves>
</pipesLevel>

```

Fig. 8: XML object showing level “111” of the Refraction game

Figure 8 shows the first object in the XML file of the Refraction game data. The first line is the start-tag of the object, inside which there are four pairs of keys and values. Similar to Section 2.4, the keys “id”, “name”, “qid”, and “version” are the attributes of the first XML object. There are also other attributes in other XML objects, such as “type”, “time”, and “representation”. Besides the attributes, the XML objects contain various key/value pairs. More details about the XML objects will be discussed in Section 4.4.

4.2 Programming in R

As discussed in Section 2.2, R has different data types, such as lists, arrays, and data frames. These data types provide multiple options to handle the datasets. In order to manipulate the Refraction data file of XML format, the functions described in the following sections will read in the XML dataset and reorganize the data from its original format into R data frames. The names of the R functions are listed below:

1. **loadXML**
2. **XML_attrs**
3. **XML_laserObject**
4. **laserObject**
5. **XML_info**

More details about these functions will be discussed in the following sections.

4.3 Input XML data

Many programming languages have built-in functions to aid software developers with the processing of XML data. The XML package in R (Temple Lang, 2012) contains built-in functions for handling data of XML format. It provides the parser to read in XML data and store the data in R as a list data type. The first function listed above, **loadXML**, is written to input XML data. The following output is the R list produced by the **loadXML** function based on the first XML object (Figure 8).

```
> data = loadXML("FGSpecialLevelswithQids_joined.XML")
> data[[1]]

$useSpecialLevel
[1] "1"

$enableMagnify
[1] "0"

$partitionLasers
[1] "1"

$partitionUnit
$partitionUnit$num
[1] "1"

$partitionUnit$denom
[1] "1"

$laserObject
$laserObject$className
[1] "FGLaserSink"
```

\$laserObject\$inputDirection

[1] "WEST"

\$laserObject\$index

[1] "88"

\$laserObject\$targetValue

\$laserObject\$targetValue\$num

[1] "1"

\$laserObject\$targetValue\$denom

[1] "1"

\$laserObject

\$laserObject\$className

[1] "FGLaserSource"

\$laserObject\$outputDirection

[1] "SOUTH"

\$laserObject\$index

[1] "32"

\$laserObject

\$laserObject\$className

[1] "FGLaserDivider"

\$laserObject\$inputDirection

```
[1] "NORTH"
```

```
$laserObject$outputDirection
```

```
[1] "EAST"
```

```
$laserObject$index
```

```
[1] "82"
```

```
$concept
```

```
$concept$name
```

```
[1] "bend"
```

```
$concept$level
```

```
[1] "1"
```

```
$numMoves
```

```
[1] "1"
```

```
$.attrs
```

id	name	qid	version
"TheBeginning"	"The Beginning"	"111"	"1.0"

4.4 Data and information in the XML objects

After the XML objects are read into R, we want to extract data and information of particular interest. There are 174 XML objects contained in the Refraction game

data file. Each XML object is corresponding to a specific game level. The following sections will describe the R functions in detail.

4.4.1 Attributes data frame

There are pairs of keys and values within the start tags of all XML objects. These are defined as the attributes of the objects. These attributes provide general information about the XML objects. The function `XML_attrs` returns a data frame that contains the attributes for each of the XML objects.

```
> attrs_frame = XML_attrs(data)
```

```
> head(attrs_frame)
```

	id	name	qid	version	type	time	representation
1	TheBeginning	The Beginning	111	1.0	<NA>	<NA>	<NA>
2	LearningBenders	Fender Bender	112	1.0	<NA>	<NA>	<NA>
3	MultipleBends	Step Up	142	1.0	<NA>	<NA>	<NA>
4	LearningBlockers	On the Rocks	<NA>	1.0	<NA>	<NA>	<NA>
5	urn	U-Turn	<NA>	1.0	<NA>	<NA>	<NA>
6	Halfsies	Halfsies	113	1.0	<NA>	<NA>	<NA>

Recall that the “qid” value can serve as the primary identifier for the JSON objects in Chapter 2. Unfortunately, many of the XML objects miss the “qid” value in their attributes and it doesn’t make sense to use “qid” as the only identifier for XML objects any more. As can be seen in the data frame above, the “version” attribute has the same value “1.0” for all objects. Many of the objects don’t have the last three keys (“type”, “time”, and “representation”) in their attributes. The following data frames will include the attributes “id”, “name”, and “qid” as the identifiers for the XML objects.

4.4.2 Laser objects data frames

Each of the XML objects contains one or more keys that are called “laserObject”. Each “laserObject” key provides detailed information of a specific laser object in a game level. The “laserObject” keys are of particular interest since the laser objects can be referred to as the “pieces” in the JSON objects, which we have discussed in Section 2.4. The function `XML_laserObject` returns a data frame containing the counts of the laser objects in each level of the Refraction game.

```
> laserObject_frame = XML_laserObject(data)
> head(laserObject_frame)
```

	id	name	qid	FGLaserSource	FGLaserSink	FGLaserModifier			
1	TheBeginning	The Beginning	111	1	1	0			
2	LearningBenders	Fender Bender	112	1	1	0			
3	MultipleBends	Step Up	142	1	1	0			
4	LearningBlockers	On the Rocks	<NA>	1	1	0			
5	urn	U-Turn	<NA>	1	1	0			
6	Halfsies	Halfsies	113	1	2	0			
	FGLaserDivider	FGLaserRecombiner	FGLaserSubtractor	FGLaserBlocker	FGCoin	Edge			
1	1	0	0	0	0	0			
2	4	0	0	0	0	0			
3	2	0	0	0	0	0			
4	3	0	0	4	0	0			
5	2	0	0	0	0	0			
6	4	0	0	4	0	0			

As shown in the data frame above, there are nine kinds of laser objects that exist in the 174 levels of the game. These are “FGLaserSource”, “FGLaserSink”, “FGLaserModifier”, “FGLaserDivider”, “FGLaserRecombiner”, “FGLaserSubtractor”, “FGLaserBlocker”, “FGCoin”, and “Edge”. While the function `XML_laserObject`

returns how many of these objects exist in each level of the game, we need another function that can extract information about every single piece of the laser objects.

In order to extract this additional information, the function `laserObject` gets all the keys and values for each laser object and presents the data in a data frame. The function takes two arguments: the XML data and the name of the laser object of interest. For example, if we are interested in the detailed information about “FGLaserSource”, then we can call the function as follows:

```
> source_frame = laserObject(data, "FGLaserSource")
> head(source_frame)
```

	id	name	qid	className	outputDirection	index
1	TheBeginning	The Beginning	111	FGLaserSource	SOUTH	32
2	LearningBenders	Fender Bender	112	FGLaserSource	WEST	78
3	MultipleBends	Step Up	142	FGLaserSource	EAST	71
4	LearningBlockers	On the Rocks	<NA>	FGLaserSource	EAST	71
5	urn	U-Turn	<NA>	FGLaserSource	EAST	42
6	Halfsies	Halfsies	113	FGLaserSource	WEST	57

	targetValue.num	targetValue.denom
1	<NA>	<NA>
2	1	1
3	1	1
4	1	1
5	1	1
6	1	1

If we are interested in the “FGLaserBlocker”, then we call the same function and change the second argument:

```
> blocker_frame = laserObject(data, "FGLaserBlocker")
> head(blocker_frame, n = 8)
```

	id	name	qid	className	index
1	LearningBlockers	On the Rocks	<NA>	FGLaserBlocker	95
2	LearningBlockers	On the Rocks	<NA>	FGLaserBlocker	85
3	LearningBlockers	On the Rocks	<NA>	FGLaserBlocker	75
4	LearningBlockers	On the Rocks	<NA>	FGLaserBlocker	65
5	Halfsies	Halfsies	113	FGLaserBlocker	43
6	Halfsies	Halfsies	113	FGLaserBlocker	33
7	Halfsies	Halfsies	113	FGLaserBlocker	63
8	Halfsies	Halfsies	113	FGLaserBlocker	73

As shown in the data frame above, “FGLaserBlocker” shows up four times in the XML object with attributes “id”=“LearningBlockers” and “name”=“On the Rocks” and also four times in the XML object with “id”=“Halfsies” and “name”=“Halfsies”. In all the returned data frames, the keys “id”, “name”, and “qid” always take up three columns of the data frames, and together, they serve as the identifiers for the XML objects. Similar calls to the **laserObject** function can also be made with all the other laser objects.

4.4.3 Other keys in the XML objects

Except for the key “laserObject”, the XML objects don’t contain consistent information. Some of the keys are included in some of the objects, while the rest of the objects don’t contain similar information. Since all keys may be useful in the ongoing research, the function **XML_info** returns all keys that show up at least once and their values in the XML objects, except for “laserObject” keys.

```
> info_frame = XML_info(data)
```

```
> head(info_frame)
```

	id	name	qid	useSpecialLevel	enableMagnify
1	TheBeginning	The Beginning	111	1	0
2	LearningBenders	Fender Bender	112	<NA>	<NA>

3	MultipleBends	Step Up	142		<NA>	<NA>	
4	LearningBlockers	On the Rocks	<NA>		<NA>	<NA>	
5	urn	U-Turn	<NA>		<NA>	<NA>	
6	Halfsies	Halfsies	113		<NA>	<NA>	
	partitionLasers	partitionUnit.num	partitionUnit.denom	concept.name			
1	1	1	1	bend			
2	<NA>	<NA>	<NA>	bend			
3	<NA>	<NA>	<NA>	bend			
4	<NA>	<NA>	<NA>	bend			
5	<NA>	<NA>	<NA>	bend			
6	<NA>	<NA>	<NA>	<NA>			
	concept.level	numMoves	theme	type	text	concept.name.1	concept.level.1
1	1	1	<NA>	<NA>	<NA>	<NA>	<NA>
2	5	1	<NA>	<NA>	<NA>	<NA>	<NA>
3	5	1	<NA>	<NA>	<NA>	<NA>	<NA>
4	6	<NA>	<NA>	<NA>	<NA>	<NA>	<NA>
5	3	<NA>	<NA>	<NA>	<NA>	<NA>	<NA>
6	<NA>	<NA>	<NA>	<NA>	<NA>	<NA>	<NA>

Overall, all the data contained in the XML data file can be extracted and prepared for future analyses in R.

CHAPTER 5

CONCLUSIONS

5.1 Summary of research

The research presented in this report allows to manipulate data from JSON and XML format datasets from the educational Refraction game. All the research objectives described in the last three chapters were accomplished using the R programming language, since R provides a rich environment for working with data and useful tools for handling various datasets. As a result of this research, the data of the Refraction game stored in JSON and XML format are reorganized into data frames, and the data frames can be used for future analyses and visualizations (Aghababayan et al., 2013). Another task accomplished in this research is the implementation of the “JSON tree”, which provides a simple visual presentation of the structure of a JSON object. The functions built to plot JSON trees do not only work with the current Refraction JSON file, but they can also generate “tree” plots for other JSON data files.

5.2 Future research

The ultimate goal of this research is to study the effect of the Refraction game on students’ learning, and, in particular, whether students become more familiar with fraction computations or not after playing this game. The research presented here will serve as a start of future, more general research to analyze data from the Refraction game. These future studies most likely will involve more statistical techniques that have been implemented in R. Overall, this will allow us to decide whether educational games such as the Refraction game are helpful for students who are learning early mathematics topics. More importantly, this research will hopefully help the game

designers to develop advanced versions of educational games so that future generations of students will have further benefits from playing such games.

REFERENCES

- Aghababayan, A., Symanzik, J. and Martin, T. (2013), Visualization of “states” in online educational games, *in* ‘Proceedings of the 59th World Statistical Congress, Hong Kong, China, International Statistical Institute’.
- URL:** <http://www.statistics.gov.hk/wsc/CPS007-P7-S.pdf>
- Al-Khalifa, S., Jagadish, H., Koudas, N., Patel, J. M., Srivastava, D. and Wu, Y. (2002), Structural joins: A primitive for efficient XML query pattern matching, *in* ‘Data Engineering, 2002. Proceedings. 18th International Conference on’, IEEE, pp. 141–152.
- Andersen, E., Liu, Y.-E., Snider, R., Szeto, R., Cooper, S. and Popović, Z. (2011), On the harmfulness of secondary game objectives, *in* ‘Proceedings of the 6th International Conference on Foundations of Digital Games’, ACM, pp. 30–37.
- Andersen, E., Liu, Y.-E., Snider, R., Szeto, R. and Popović, Z. (2011), Placing a value on aesthetics in online casual games, *in* ‘Proceedings of the SIGCHI Conference on Human Factors in Computing Systems’, ACM, pp. 1275–1278.
- Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E. and Yergeau, F. (1997), ‘Extensible Markup Language (XML)’, *World Wide Web Journal* **2**(4), 27–66.
- Center for Game Science *CGS* (2012), ‘Refraction’.
- URL:** <http://centerforgamescience.org/portfolio/refraction/>
- Couture-Beil, A. (2012), ‘rjson: JSON for R’, *R package version 0.2.8*.
- Crockford, D. (2006), ‘JSON: The fat-free alternative to XML’, *URL:* <http://www.json.org/fatfree.html>.

Japan Broadcasting Corporation *NHK* (2011), ‘NHK Japan Prize’.

URL: *<http://www.nhk.or.jp/jp-prize/english/index.html>*

jQuery4u.com (2013), ‘Online JSON tree viewer tool’.

URL: *<http://www.jquery4u.com/demos/online-json-tree-viewer/>*

R Development Core Team (2012), *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria. ISBN 3–900051–07–0 (<http://www.R-project.org/>).

Spector, P. (2008), *Data manipulation with R*, Springer, New York, NY.

Temple Lang, D. (2011), ‘Rjsonio: Serialize R objects to JSON, JavaScript object notation’, *R package version 0.95-0* .

Temple Lang, D. (2012), ‘XML: Tools for parsing and generating XML within R and S-plus.’, *R package version 3.9-4* .

Wang, G. (2011), Improving data transmission in web applications via the translation between XML and JSON, *in* ‘2011 Third International Conference on Communications and Mobile Computing (CMC)’, IEEE, pp. 182–185.

Wilhelm, M. (2009), ‘Tree representations: Graphics libraries for displaying hierarchical data’, *Code4Lib Journal* .

URL: *<http://journal.code4lib.org>*

APPENDICES

APPENDIX A

R CODE

A.1 R code for Chapter 2

```
###  
library(RJSONIO)  
library(plyr)  
###  
  
###Input JSON file.  
loadJSON = function(json_file) {  
  data = fromJSON(json_file)  
  return(data)  
}  
  
###General information data frame.  
JSON_general = function(data) {  
  
  n = length(data)  
  concepts = list(n)  
  game_id = list(n)  
  qid = list(n)  
  concepts = sapply(data, "[", "concepts")  
  concepts = sapply(concepts, paste, collapse = ", ")  
  game_id = sapply(data, "[", "game_id")  
  qid = sapply(data, "[", "qid")  
  basic = as.data.frame(cbind(concepts, game_id, qid))
```

```

    return(basic)
}

###bin_pieces data frame
JSON_bin_pieces = function(data) {

  tempList = list(0, 0)
  nameList = list("qid", "bender")
  names(tempList) = nameList
  newFRAME = as.data.frame(tempList)
  for (i in 1:length(data)) {
    name = names(data[[i]]$"bin_pieces")
    number = list()
    for (j in 1:length(data[[i]]$"bin_pieces")) {
      number[[j]] = length(data[[i]]$"bin_pieces"[[j]])
    }
    number = c(data[[i]]$"qid", number)
    names(number) = c("qid", name)
    frame = as.data.frame(number)
    List = list(newFRAME, frame)
    newList = do.call(rbind.fill, List)
    newFRAME = as.data.frame(newList)
  }
  bin_pieces = newFRAME[-1, ]
  bin_pieces[is.na(bin_pieces[, ])] = 0

  return(bin_pieces)
}

```

```

###board_pieces data frame
JSON_board_pieces = function(data) {

  tempList = list(0, 0)
  nameList = list("qid", "source")
  names(tempList) = nameList
  newFRAME = as.data.frame(tempList)
  for (i in 1:length(data)) {
    name = names(data[[i]]$"board_pieces")
    number = list()
    for (j in 1:length(data[[i]]$"board_pieces")) {
      number[[j]] = length(data[[i]]$"board_pieces"[[j]])
    }
    number = c(data[[i]]$"qid", number)
    names(number) = c("qid", name)
    frame = as.data.frame(number)
    List = list(newFRAME, frame)
    newList = do.call(rbind.fill, List)
    newFRAME = as.data.frame(newList)
  }
  board_pieces = newFRAME[-1, ]
  board_pieces[is.na(board_pieces[, ])] = 0

  return(board_pieces)
}

###win_pieces data frame
JSON_win_pieces = function(data) {

  tempList = list(0, "bender")

```

```

nameList = list("qid", "pieces")
names(tempList) = nameList
newFRAME = as.data.frame(tempList)
for (i in 1:length(data)) {
  name = names(data[[i]]$"win_pieces")
  number = list()
  for (j in 1:length(name)) {
    order = 1
    for (k in 1:length(data[[i]]$"win_pieces"[[j]])) {
      number = list(data[[i]]$"qid", name[j],
                    data[[i]]$"win_pieces"[[j]][k], order)
      names(number) = c("qid", "pieces", "id", "order")
      order = order + 1
      frame = as.data.frame(number)
      List = list(newFRAME, frame)
      newList = do.call(rbind.fill, List)
      newFRAME = as.data.frame(newList)
    }
  }
}
win_pieces = newFRAME[-1, ]

return(win_pieces)
}

```

```

###Detailed information of each piece.

```

```

pieces = function(data, PieceName) {

  frame1 = JSON_bin_pieces(data)
  frame2 = JSON_board_pieces(data)

```



```

        [PieceName][[1]][[j]]$"id",
        data[[i]]$"board_pieces"
        [PieceName][[1]][[j]]$"value"[[1]],
        data[[i]]$"board_pieces"
        [PieceName][[1]][[j]]$"value"[[2]])
    names(list[[index]]) = c("qid", "id", "denom_value", "num_value")
    index = index + 1
  }
}
}
}
}
} else if (length(which(PieceName == names(frame1))) != 0) {
  for (i in 1:length(data)) {
    if (length(data[[i]]$"bin_pieces"[PieceName][[1]]) > 0) {
      for (j in 1:length(data[[i]]$"bin_pieces"[PieceName][[1]])) {
        if (length(data[[i]]$"bin_pieces"
          [PieceName][[1]][[j]]) == 1) {
          list[[index]] = c(data[[i]]$"qid",
            data[[i]]$"bin_pieces"
            [PieceName][[1]][[j]]$"id")
          names(list[[index]]) = c("qid", "id")
          index = index + 1
        } else if (length(data[[i]]$"bin_pieces"
          [PieceName][[1]][[j]]) > 1) {
          if (length(data[[i]]$"bin_pieces"
            [PieceName][[1]][[j]]$"value") == 1) {
            list[[index]] = c(data[[i]]$"qid",
              data[[i]]$"bin_pieces"
              [PieceName][[1]][[j]]$"id",
              data[[i]]$"bin_pieces"

```

```

                                [PieceName][[1]][[j]]$"value")
names(list[[index]]) = c("qid", "id", "value")
index = index + 1
} else if (length(data[[i]]$"bin_pieces"
                                [PieceName][[1]][[1]]$"value") > 1) {
list[[index]] = c(data[[i]]$"qid",
                                data[[i]]$"bin_pieces"
                                [PieceName][[1]][[j]]$"id",
                                data[[i]]$"bin_pieces"
                                [PieceName][[1]][[j]]$"value"[[1]],
                                data[[i]]$"bin_pieces"
                                [PieceName][[1]][[j]]$"value"[[2]])
names(list[[index]]) = c("qid", "id", "denom_value", "num_value")
index = index + 1
}
}
}
}
}
} else {
  stop("Never seen such a piece!!")
}
frame = do.call(rbind, list)

return(frame)
}

```

```
###Call the functions.
```

```
data = loadJSON("refraction.json")
```



```
general_frame = JSON_general(data)
bin_frame = JSON_bin_pieces(data)
board_frame = JSON_board_pieces(data)
win_frame = JSON_win_pieces(data)

piece_frame1 = pieces(data, "source")
piece_frame2 = pieces(data, "splitter")
piece_frame3 = pieces(data, "usu")
```

A.2 R code for Chapter 3

```

###Input JSON file.
library(RJSONIO)
data = fromJSON("refraction.json")

###Generate a matrix of JSON tree structure.
JSONtree = function(JSONlist) {

  #First level keys.
  nodes = unique(c(names(JSONlist), names(unlist(JSONlist))))
  nodeslist = strsplit(nodes, "\\.")
  first = unique(sapply(nodeslist, "[", 1))
  col = max(sapply(nodeslist, length))
  mat = matrix(, length(first), col)
  mat[, 1] = first

  #Other level keys.
  for(k in 2:col) {
    n = length(which(is.na(mat[, k - 1]) == FALSE))
    for(i in 1:n) {
      row = which(is.na(mat[, k - 1]) == FALSE)[i]
      branch = nodeslist
      for(j in 1:(k - 1)) {
        where = which(sapply(branch, "[", j) == mat[row:1, j]
                          [min(which(!is.na(mat[row:1, j]))))]
        branch = branch[where]
      }
      branch = sapply(branch, "[", k)
      [is.na(sapply(branch, "[", k)) == FALSE]
      branch = unique(branch)
    }
  }
}

```

```

    newmat = matrix(, length(branch), col)
    newmat[, k] = branch
    if(row < dim(mat)[1]) {
      mat = rbind(mat[1:row, ], newmat,
                  mat[(row + 1):dim(mat)[1], ])
    } else if(row == dim(mat)[1]) {
      mat = rbind(mat[1:row, ], newmat)
    }
  }
}

class(mat) = "JSONtree"
return(mat)
}

```

```

###Print the JSON tree.
print.JSONtree = function(matrix) {

  #Inherit output.
  if(!inherits(matrix, "JSONtree"))
    stop("Wrong Data!!")
  matrix[is.na(matrix)] = " "
  newmat = matrix(" ", dim(matrix)[1], 1)
  matrix = cbind(newmat, matrix)
  newmat = matrix(" ", 1, dim(matrix)[2])
  matrix = rbind(newmat, matrix)
  matrix[1, 1] = "ROOT"
  mat = matrix
  matrix = mat[1, ]
  newmat = matrix(" ", 1, dim(mat)[2])

```

```

for (i in 2:dim(mat)[1]) {
  matrix = rbind(matrix, newmat, mat[i, ])
}
for(i in 1:(dim(matrix)[2] - 1)) {
  where = which(matrix[, i] != " ")
  if(length(where) == 1) {
    nwhere = which(matrix[, i + 1] != " ")
    matrix[(where + 1):max(nwhere), i] = "|"
  } else if(length(where) > 1) {
    for(j in 1:(length(where) - 1)) {
      if(max(which(matrix[(where[j] + 2), ] != " ")) > i) {
        nwhere = which(matrix[where[j]:
                               where[j + 1], i + 1] != " ")
        matrix[(where[j]:where[j + 1])
                [2:max(nwhere)], i] = "|"
      }
    }
  }
  if(where[length(where)] < dim(matrix)[1]) {
    nwhere = which(matrix[where[length(where)]:dim(matrix)[1], i + 1] != " ")
    if(length(nwhere) > 0 ) {
      matrix[(where[length(where)]:dim(matrix)[1])[2:max(nwhere)], i] = "|"
    }
  }
}
}
mat = matrix[, 1]
newmat = matrix(" ", dim(matrix)[1], 1)
for (i in 2:dim(matrix)[2]) {
  mat = cbind(mat, newmat, matrix[, i])
}
for(i in 1:(dim(matrix)[2] - 1)) {

```

```

for(j in 1:dim(matrix)[1]) {
  k = 2 * i
  if(mat[j, k - 1] == "|" && mat[j, k + 1] != " " &&
      mat[j, k + 1] != "|") {
    mat[j, k] = "- - - -"
  }
}
}

write.table(format(mat, justify = "centre"),
            row.names = F, col.names = F, quote = F)
}

```

###Generate union tree.

```

uniontree = function(data) {
  nodes = unique(names(unlist(data)))
  nodes = unique(sub("[0-9]?$", "", nodes))
  nodes = unique(sub("[0-9]?$", "", nodes))
  nodeslist = strsplit(nodes, "\\.")
  nodeslist = sapply(nodeslist, "[", -1)
  first = unique(sapply(nodeslist, "[", 1))
  col = max(sapply(nodeslist, length))
  mat = matrix(, length(first), col)
  mat[, 1] = first

  for(k in 2:col) {
    n = length(which(is.na(mat[, k - 1]) == FALSE))
    for(i in 1:n) {
      row = which(is.na(mat[, k - 1]) == FALSE)[i]
      branch = nodeslist
    }
  }
}

```

```

for(j in 1:(k - 1)) {
  where = which(sapply(branch, "[", j) == mat[row:1, j]
                [min(which(!is.na(mat[row:1, j]))))]
  branch = branch[where]
}
branch = sapply(branch, "[", k)[is.na(sapply(branch, "[", k)) == FALSE]
branch = unique(branch)
newmat = matrix(, length(branch), col)
newmat[, k] = branch
if(row < dim(mat)[1]) {
  mat = rbind(mat[1:row, ], newmat, mat[(row + 1):dim(mat)[1], ])
} else if(row == dim(mat)[1]) {
  mat = rbind(mat[1:row, ], newmat)
}
}
}

class(mat) = "JSONtree"
return(mat)
}

```

```

###Generate intersection tree.
intersecttree = function(data) {
  nodes = unique(c(names(data[[1]]), names(unlist(data[[1]]))))
  nodeslist1 = strsplit(nodes, "\\.")
  col = max(sapply(nodeslist1, length))
  list = list()
  for(j in 1:col) {
    newlist = unique(lapply(nodeslist1, "[", 1:j))
    k = 1

```

```

for(i in 1:length(newlist)) {
  if(is.na(newlist[[k]][length(newlist[[k]])])) {
    newlist[[k]] = NULL
    k = k - 1
  }
  k = k + 1
}
list = c(list, newlist)
}
nodeslist = list
for(l in 1:length(data)) {
  nodes = unique(c(names(data[[l]]), names(unlist(data[[l]]))))
  nodeslist1 = strsplit(nodes, "\\.")
  col = max(sapply(nodeslist1, length))
  list = list()
  for(j in 1:col) {
    newlist = unique(lapply(nodeslist1, "[", 1:j))
    k = 1
    for(i in 1:length(newlist)) {
      if(is.na(newlist[[k]][length(newlist[[k]])])) {
        newlist[[k]] = NULL
        k = k - 1
      }
      k = k + 1
    }
    list = c(list, newlist)
  }
  nodeslist = intersect(nodeslist, list)
}
first = unique(sapply(nodeslist, "[", 1))
col = max(sapply(nodeslist, length))

```

```

mat = matrix(, length(first), col)
mat[, 1] = first

for(k in 2:col) {
  n = length(which(is.na(mat[, k - 1]) == FALSE))
  for(i in 1:n) {
    row = which(is.na(mat[, k - 1]) == FALSE)[i]
    branch = nodeslist
    for(j in 1:(k - 1)) {
      where = which(sapply(branch, "[", j) == mat[row:1, j]
                        [min(which(!is.na(mat[row:1, j]))))]
      branch = branch[where]
    }
    branch = sapply(branch, "[", k)[is.na(sapply(branch, "[", k)) == FALSE]
    branch = unique(branch)
    newmat = matrix(, length(branch), col)
    newmat[, k] = branch
    if(row < dim(mat)[1]) {
      mat = rbind(mat[1:row, ], newmat, mat[(row + 1):dim(mat)[1], ])
    } else if(row == dim(mat)[1]) {
      mat = rbind(mat[1:row, ], newmat)
    }
  }
}

class(mat) = "JSONtree"
return(mat)
}

```

###Call Functions.


```
matrix = JSONtree(data[[1]])  
print(matrix)  
uniontree(data)  
intersecttree(data)
```

A.3 R code for Chapter 4

```

###

library(XML)
library(plyr)

###

###Input XML file.
loadXML = function(filename) {
  xml_data = xmlParse(filename)
  xml_list = xmlToList(xml_data)
  return(xml_list)
}

###Attributes data frame
XML_attrs = function(data) {
  pipeList = c(llply(data[1]$pipesLevel$.attrs))
  nameList = c(llply(names(data[1]$pipesLevel$.attrs)))
  names(pipeList) = nameList
  frame = as.data.frame(pipeList)
  for (i in 2:length(data)) {
    pipeList = c(llply(data[i]$pipesLevel$.attrs))
    nameList = c(llply(names(data[i]$pipesLevel$.attrs)))
    names(pipeList) = nameList
    frame = rbind.fill(frame, as.data.frame(pipeList))
  }
  return(frame)
}

```

```

###The counts of laserObject's.
XML_laserObject = function(data) {
  xml_list = data
  templist = c(rep(list(as.factor(0)), 4), rep(list(0), 9))
  nameList = c(1:ply(names(xml_list[1]$pipesLevel$.attrs),
                    "FGLaserSource", "FGLaserSink",
                    "FGLaserModifier", "FGLaserDivider",
                    "FGLaserRecombiner", "FGLaserSubtractor",
                    "FGLaserBlocker", "FGCoin", "Edge")
  names(templist) = nameList
  newFRAME = as.data.frame(templist)

  for (i in 1:length(xml_list)) {
    laserObject = xml_list[[i]][names(xml_list[[i]]) == "laserObject"]
    pipeList = list()
    nameList = list()
    FGLaserSource = 0
    FGLaserSink = 0
    FGLaserModifier = 0
    FGLaserDivider = 0
    FGLaserRecombiner = 0
    FGLaserSubtractor = 0
    FGLaserBlocker = 0
    FGCoin = 0
    Edge = 0

    if(length(laserObject) == 0) {
      laserObject = laserObject
    } else if(length(laserObject) > 0) {
      for (j in 1:length(laserObject)) {
        if(laserObject[[j]]$className == "FGLaserSource") {

```

```

    FGLaserSource = FGLaserSource + 1
  } else if(laserObject[[j]]$className == "FGLaserSink") {
    FGLaserSink = FGLaserSink + 1
  } else if(laserObject[[j]]$className == "FGLaserModifier") {
    FGLaserModifier = FGLaserModifier + 1
  } else if(laserObject[[j]]$className == "FGLaserDivider") {
    FGLaserDivider = FGLaserDivider + 1
  } else if(laserObject[[j]]$className == "FGLaserRecombiner") {
    FGLaserRecombiner = FGLaserRecombiner + 1
  } else if(laserObject[[j]]$className == "FGLaserSubtractor") {
    FGLaserSubtractor = FGLaserSubtractor + 1
  } else if(laserObject[[j]]$className == "FGLaserBlocker") {
    FGLaserBlocker = FGLaserBlocker + 1
  } else if(laserObject[[j]]$className == "FGCoin") {
    FGCoin = FGCoin + 1
  } else if(laserObject[[j]]$className == "Edge") {
    Edge = Edge + 1
  }
}
}

pipeList = c(llply(xml_list[i]$pipesLevel$.attrs),
             FGLaserSource, FGLaserSink, FGLaserModifier, FGLaserDivider,
             FGLaserRecombiner, FGLaserSubtractor,
             FGLaserBlocker, FGCoin, Edge)
nameList = c(llply(names(xml_list[i]$pipesLevel$.attrs),
                  "FGLaserSource", "FGLaserSink", "FGLaserModifier",
                  "FGLaserDivider", "FGLaserRecombiner", "FGLaserSubtractor",
                  "FGLaserBlocker", "FGCoin", "Edge")
            names(pipeList) = nameList
frame = as.data.frame(pipeList)

```

```

List = list(newFRAME, frame)
newList = do.call(rbind.fill, List)
newFRAME = as.data.frame(newList)
}

name = c("id", "name", "qid",
         "FGLaserSource", "FGLaserSink", "FGLaserModifier",
         "FGLaserDivider", "FGLaserRecombiner", "FGLaserSubtractor",
         "FGLaserBlocker", "FGCoin", "Edge")
newFRAME = newFRAME[, names(newFRAME) %in% name]
newFRAME = newFRAME[2:dim(newFRAME)[1], ]
rownames(newFRAME) = 1:nrow(newFRAME)
return(newFRAME)
}

###Detailed information of each laser object.
laserObject = function(data, laser) {
  list = list()
  index = 1
  for (i in 1:length(data)) {
    if(length(which(names(data[[i]]) == "laserObject")) == 0) {
      index = index
    } else {
      for (j in 1:length(names(data[[i]]) == "laserObject")) {
        if(data[[i]][names(data[[i]]) == "laserObject"]
           [[j]]$"className" == laser) {
          list[[index]] = c(1lply(data[i]$pipesLevel$.attrs),
                           unlist(data[[i]]
                                   [names(data[[i]])
                                       == "laserObject"][[j]]))
        }
      }
    }
  }
}

```

```

        index = index + 1
      }
    }
  }
}

frame = as.data.frame(list[[1]])
for (i in 2:length(list)) {
  frame = rbind.fill(frame, as.data.frame(list[[i]]))
}

frame = frame[, !(names(frame) %in% c("version", "time", "representation"))]
laser = frame[, !(names(frame) %in% c("id", "name", "qid"))]
id = frame$id
name = frame$name
qid = frame$qid
frame = cbind(id, name, qid, laser)
return(frame)
}

###Other keys of each XML object.
XML_info = function(data) {
  list = list()
  for (i in 1:length(data)) {
    temp = data[[i]][names(data[[i]]) != "laserObject"]
    temp = temp[names(temp) != ".attrs"]
    list[[i]] = c(lapply(data[i]$pipesLevel$.attrs), unlist(temp))
  }
  frame = as.data.frame(list[[1]])
  for (i in 2:length(list)) {
    frame = rbind.fill(frame, as.data.frame(list[[i]]))
  }
}

```

```
frame = frame[, !(names(frame) %in% c("version", "time", "representation"))]  
return(frame)  
}
```

```
###Call the functions.
```

```
data = loadXML("FGSpecialLevelswithQids_joined.XML")
```

```
attrs_frame = XML_attrs(data)
```

```
laserObject_frame = XML_laserObject(data)
```

```
info_frame = XML_info(data)
```

```
laser1 = laserObject(data, "FGLaserSource")
```

```
laser2 = laserObject(data, "FGLaserBlocker")
```